

Queensland University of Technology  
Faculty of Science and Technology

# Debugging with Declarative Transformation Languages

**Mark T. Hibberd**

Supervisors:  
Professor Kerry Raymond  
Faculty of Science and Technology, QUT

Dr Michael Lawley  
EHealth Research, CSIRO

August, 2009

## Statement Of Originality

I wrote this sucker.

## Acknowledgements

Thanks to everyone, Cheers and Beers.

## Abstract

– Brief overview of problem and proposed solution. —

Debugging is a key aspect to any pragmatic development approach. This thesis presents an approach for forensic debugging of transformations in model-driven development. Combining research from the model-driven development community with traditional and declarative debugging techniques, the presented approach utilises model-driven developments inherent traceability to provide post-hoc analysis of model transformations.

– continue on, describing trace enhancements and problem identification aspects –

[from forensic debugging paper](#)

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.

[from declarative debugging paper](#)

Declarative model transformation languages provide a powerful, expressive mechanism to define model-to-model transformations. However, this high-level approach to model transformations is currently limited by its dependence on traditional low-level and imperative debugging techniques. This paper proposes that high-level approaches to model-transformation, require high-level debugging techniques. Critical to achieving higher-level debugging is being able to highlight the developer's incorrect assumptions and misinterpretation of the execution model. We exploit data and execution trace to present new visualisation techniques that are able to communicate a more accurate understanding of the transformation execution to the developer. Using these techniques we show how developer's intentions and assumptions can be verified in order to localize model transformations bugs. The approaches are implemented using the Eclipse Modelling Framework and the Tefkat declarative model transformation engine.

[from confirmation](#)

In software, models are abstractions used to define a system in a way that aims to be easier to understand, communicate and reason about. Model-driven engineering (MDE) utilises models as first-class artifacts in the software development process. Developers are able to define applications using modeling languages, both graphical and textual, that are more expressive and better at capturing domain knowledge without having to deal with technical concerns.

MDE has a wide range of potential benefits, which, depending on the mode of MDE used, can lead to higher quality, productivity, maintainability or some combination of these key measures of software development. In an *ideal* world the benefits of MDE over traditional code-centric approaches would be clear and easily measurable. However, the world of software development is far from perfect; people make mistakes; bugs happen.

Mature software development languages realise and embrace this fact. Testing and debugging support are integral features for any modern language. Without this support, the barrier to adoption and effective use of a new language would be far too high. MDE is faced with this same challenge. Its adoption and effectiveness is currently limited by the unresolved issues involved with handling the imperfect nature of software development - that is, bugs. How to tell if there is a bug? How to find the bug? And, how to fix the bug? If these questions can be answered effectively, MDE will be significantly closer to realising its full potential.

This research investigates these questions in more detail, drawing upon traditional debugging techniques in order to introduce new debugging approaches specific to, and optimised for, MDE. The aim of this research is to **define** the debugging issues that MDE faces, **analyse** current MDE practices in order to identify improvements to debugging support and **develop** pragmatic, generalized solutions for debugging in MDE.

# Contents

0.1	FIXES . . . . .	1
<b>1</b>	<b>Introduction</b>	<b>9</b>
<b>2</b>	<b>Research Question</b>	<b>11</b>
2.1	Bug Lifecycle . . . . .	11
2.2	Research Questions . . . . .	13
2.3	old stuff . . . . .	13
2.3.1	Research Problem . . . . .	13
2.3.2	Research Objectives . . . . .	14
2.3.3	Research Question . . . . .	15
<b>3</b>	<b>Literature Review</b>	<b>18</b>
3.1	Logic Debugging . . . . .	18
3.2	Traditional Debugging . . . . .	18
3.3	MDD Debugging . . . . .	18
3.4	OLD-WORK-DUMPING-GROUND . . . . .	19
3.4.1	Introduction . . . . .	19
3.4.2	Model-Driven Development . . . . .	20
3.4.3	Software Verification . . . . .	24
3.4.4	Software Debugging . . . . .	26
3.4.5	Conclusion . . . . .	27
3.5	Mental Model References . . . . .	29
<b>4</b>	<b>Declarative Model Transformations and Tefkat</b>	<b>31</b>
4.1	Tefkat . . . . .	31
4.1.1	Meta-Models . . . . .	31
4.1.2	Evaluation Order . . . . .	32
4.1.3	Trace . . . . .	32
4.1.4	Data structures . . . . .	32
4.1.5	Optimisation . . . . .	32
4.2	Running Example . . . . .	33
4.2.1	Meta-Models . . . . .	33
4.2.2	Person to Male-Female Transformation . . . . .	35
4.2.3	Male-Female to Person Transformation . . . . .	38
4.2.4	Factored Male-Female to Person Transformation . . . . .	40
4.2.5	Evaluation Order . . . . .	41
4.2.6	FROM . . . . .	41

<b>5</b>	<b>Bugs, Observations and Expectations</b>	<b>42</b>
5.1	What is a bug? . . . . .	42
5.2	Cognitive Error . . . . .	45
5.2.1	Cognitive Errors in Model Transformations . . . . .	49
5.2.2	Mental Models . . . . .	50
5.3	Implementation Error . . . . .	50
5.4	Observation Error . . . . .	52
5.5	Expectations . . . . .	56
5.5.1	Explicit vs Implicit Expectations . . . . .	58
5.5.2	Output vs Program Expectations . . . . .	58
5.5.3	Positive Programming Philosophy . . . . .	58
5.5.4	Expectations and Focus . . . . .	58
5.5.5	Expectations and the Definition of a Bug . . . . .	58
5.6	Origins of a bug . . . . .	58
5.7	Left-over diagrams . . . . .	59
<b>6</b>	<b>Symptoms and Questions</b>	<b>62</b>
6.1	Traits of a Failure . . . . .	62
6.1.1	Location . . . . .	62
6.1.2	Failure Mode . . . . .	63
6.1.3	Type . . . . .	64
6.2	Debugging Questions . . . . .	64
6.2.1	Detail Questions . . . . .	64
6.2.2	Structural Questions . . . . .	65
6.2.3	Trace Questions . . . . .	65
6.2.4	Execution Questions . . . . .	66
6.2.5	Smells . . . . .	66
6.3	Old Debugging Questions Section — IGNORE BELOW - REF- ERENCE ONLY . . . . .	66
6.3.1	Classes of Bugs . . . . .	69
6.4	Notes . . . . .	70
<b>7</b>	<b>Debugging Techniques</b>	<b>71</b>
7.1	Traditional Debugging Approaches . . . . .	71
7.2	Forensic Debugging . . . . .	71
7.2.1	Identifying Potential Causes of Problem XYZ . . . . .	72
7.2.2	Pinpointing The Problem . . . . .	72
7.2.3	Initial Discussion . . . . .	72
7.2.4	Forensic Debugging with Enhanced Trace . . . . .	72
7.2.5	Trace Enhancements . . . . .	72
7.2.6	Utilisation in Forensic Debugging . . . . .	72
7.2.7	Lint . . . . .	72
7.2.8	OLD-GEAR-DUMPING-GROUND . . . . .	72
7.2.9	Localisation of Model Transformation Bugs . . . . .	72
7.2.10	Analysis . . . . .	72
7.2.11	Re-enactment . . . . .	74
7.3	Live Debugging . . . . .	77
7.3.1	Term Coverage . . . . .	77
7.3.2	Data Profiling . . . . .	77

<b>8</b>	<b>Observation and Verification Techniques</b>	<b>78</b>
<b>9</b>	<b>Advanced Example of Debugging Techniques</b>	<b>79</b>
9.1	UML to Reqlational . . . . .	79
9.2	Example 2 . . . . .	83
<b>10</b>	<b>Proof of Concept Wrapup</b>	<b>84</b>
<b>11</b>	<b>Discussion</b>	<b>86</b>
11.1	Interactions with live debugging . . . . .	86
11.2	Generalisation . . . . .	86
11.2.1	Model Transformations and Other Declarative Programming Models . . . . .	86
11.2.2	Imperative Programming Models . . . . .	86
11.3	Future Work . . . . .	86
<b>12</b>	<b>Conclusion</b>	<b>88</b>
<b>13</b>	<b>Dumping ground...</b>	<b>90</b>
13.1	confirmation . . . . .	90
13.2	models 2007 - accepted - forensic debugging . . . . .	152
13.3	models 2008 - submission - declarative debugging . . . . .	170
13.4	initial sketch of trace paper . . . . .	186
	<b>References</b>	<b>90</b>
	<b>Appendices</b>	<b>189</b>
<b>A</b>	<b>Person to Male-Female Transformation Listing</b>	<b>189</b>
<b>B</b>	<b>Male-Female to Person Transformation Listing</b>	<b>191</b>
<b>C</b>	<b>Factored Male-Female to Person Transformation Listing</b>	<b>192</b>
<b>D</b>	<b>Trace Enhancement Code</b>	<b>193</b>
<b>E</b>	<b>Forensic Debugging Code / Transformations</b>	<b>194</b>
<b>F</b>	<b>Problem Identification Transformation</b>	<b>195</b>

# List of Tables

# List of Figures

2.1	Bug lifecycle. . . . .	12
2.2	Old bug lifecycle. . . . .	12
4.1	Trace Meta-model . . . . .	32
4.2	Unisex meta-model, representing people in the running example. . . . .	34
4.3	Example input for unisex meta-model. . . . .	34
4.4	Gender meta-model, representing people in the running example. . . . .	35
4.5	Example input for gender meta-model. . . . .	35
5.1	Linear view of program. . . . .	42
5.2	Naive view of a bug. . . . .	43
5.3	Linear view of program emphasis on translations between cognitive and concrete aspects. . . . .	43
5.4	Shared model for development elements. . . . .	44
5.5	$A \cap C$ , the set of correct programs. . . . .	44
5.6	$A \cap B \cap C$ , the set of ideal programs. . . . .	45
5.7	$A \Delta B$ , Cognitive bugs. . . . .	46
5.8	$B \Delta C$ , Implementation bugs. . . . .	46
5.9	Cognitive bugs. . . . .	47
5.10	Implementation bugs. . . . .	51
5.11	Observation bugs. . . . .	52
5.12	A simple c program that prints blocks of output. . . . .	53
5.13	A simple c program that prints blocks of output with summary line. . . . .	54
5.14	Developers observations of the output. . . . .	54
5.15	The actual output of the program. . . . .	55
5.16	A simple c program with bug. . . . .	55
5.17	Expectations. . . . .	57
5.18	Overlapping models. . . . .	59
5.19	Overlapping models. . . . .	60
5.20	High level views of a program. . . . .	60
5.21	Reality vs Cognitive vs Concrete views. . . . .	61
6.1	Question categories. . . . .	69
7.1	Simple Tefkat rule containing a bug. . . . .	74
7.2	Simple UML meta-model. . . . .	75
7.3	Head-first predicate switching. . . . .	75
7.4	Tail-first predicate switching. . . . .	76

7.5	Handling multiple branches. . . . .	77
9.1	Meta-model representing our simplified UML example. . . . .	80
9.2	Meta-model representing our relational model example. . . . .	80
9.3	Source to target relations of increasing detail. . . . .	81

FIX: glossary

FIX: abbreviations

## 0.1 FIXES

auto-generated list of fixes:

- `advanced_example.tex:205`: Scope out rddl
- `symptoms_and_questions.tex:3`: common language
- `symptoms_and_questions.tex:22`: not sure if it is worth looking at
- `symptoms_and_questions.tex:24`: this needs to link back into the expectations
- `symptoms_and_questions.tex:37`: This is not quite what I had in mind
- `symptoms_and_questions.tex:105`: link back to expectations for concrete
- `symptoms_and_questions.tex:109`: again link back to expectations for
- `symptoms_and_questions.tex:129`: describe question phrasing
- `symptoms_and_questions.tex:131`: bake in cross cutting concerns
- `symptoms_and_questions.tex:136`: sort question references
- `symptoms_and_questions.tex:156`: Need more
- `symptoms_and_questions.tex:201`: type check
- `symptoms_and_questions.tex:219`: extract the rest of the questions from lint impl
- `thesis.tex:81`: Links and xrefs
- `researchq.tex:7`: restructure pivot point
- `researchq.tex:32`: simple definition of bug
- `researchq.tex:34`: need to go into the identify
- `researchq.tex:41`: first paragraph of chapter
- `researchq.tex:46`: should we flip the diagram for the bug lifecycle
- `thesis-20100916-diff.tex:204`: glossary
- `thesis-20100916-diff.tex:211`: abbreviations
- `thesis-20100916-diff.tex:217`: ES
- `thesis-20100916-diff.tex:341`: Links and xrefs
- `thesis-20100916-diff.tex:377`: restructure pivot point
- `thesis-20100916-diff.tex:400`: simple definition of bug
- `thesis-20100916-diff.tex:895`: We kind of dropped it from the fore during restructure discussions

- thesis-20100916-diff.tex:918: Scope out rddl
- thesis-20100916-diff.tex:948: typo
- thesis-20100916-diff.tex:958: first paragraph introduces the bias that often makes bugs appear
- thesis-20100916-diff.tex:978: better
- thesis-20100916-diff.tex:1035: consider rename of categories
- thesis-20100916-diff.tex:1040: more rework todo after here
- thesis-20100916-diff.tex:1056: refs
- thesis-20100916-diff.tex:1061: TODO: need to dive into the cognitive area a bit
- thesis-20100916-diff.tex:1089: at blub for conceptual bug - just not what the programmer intended
- thesis-20100916-diff.tex:1111: not sure this is actually true
- thesis-20100916-diff.tex:1169: deeper definition of bug
- thesis-20100916-diff.tex:1368:
- thesis-20100916-diff.tex:1371: observations
- thesis-20100916-diff.tex:1374: observations
- thesis-20100916-diff.tex:1381: This has been pulled in as a part of restructure
- thesis-20100916-diff.tex:1402: I am missing something here
- thesis-20100916-diff.tex:1532: common language
- thesis-20100916-diff.tex:1536: common language
- thesis-20100916-diff.tex:1561: not sure if it is worth looking at
- thesis-20100916-diff.tex:1568: this needs to link back into the expectations
- thesis-20100916-diff.tex:1587: This is not quite what I had in mind
- thesis-20100916-diff.tex:1593: This is not quite what I had in mind
- thesis-20100916-diff.tex:1617: trace symptoms
- thesis-20100916-diff.tex:1631: output symptoms
- thesis-20100916-diff.tex:1642: source symptoms
- thesis-20100916-diff.tex:1670: transform symptoms
- thesis-20100916-diff.tex:1716: complete magnitude symptoms
- thesis-20100916-diff.tex:1721: complete structure symptoms

- thesis-20100916-diff.tex:1726: complete details symptoms
- thesis-20100916-diff.tex:1730: complete balance symptoms
- thesis-20100916-diff.tex:1732: link back to expectations for concrete
- thesis-20100916-diff.tex:1737: link back to expectations for concrete
- thesis-20100916-diff.tex:1740: complete coverage symptoms
- thesis-20100916-diff.tex:1741: again link back to expectations for
- thesis-20100916-diff.tex:1744: again link back to expectations for
- thesis-20100916-diff.tex:1787: describe question phrasing
- thesis-20100916-diff.tex:1790: describe question phrasing
- thesis-20100916-diff.tex:1793: bake in cross cutting concerns
- thesis-20100916-diff.tex:1796: bake in cross cutting concerns
- thesis-20100916-diff.tex:1807: sort question references
- thesis-20100916-diff.tex:1808: sort question references
- thesis-20100916-diff.tex:1846: Need more
- thesis-20100916-diff.tex:1854: Need more
- thesis-20100916-diff.tex:1953: type check
- thesis-20100916-diff.tex:1954: type check
- thesis-20100916-diff.tex:1987: extract the rest of the questions from lint impl
- thesis-20100916-diff.tex:1990: extract the rest of the questions from lint impl
- thesis-20100916-diff.tex:2218: This chapter will be disproportionately large
- thesis-20100916-diff.tex:2224: better discussion of granularity
- thesis-20100916-diff.tex:2225: better discussion of granularity
- thesis-20101118-diff.tex:183: glossary
- thesis-20101118-diff.tex:190: abbreviations
- thesis-20101118-diff.tex:196: ES
- thesis-20101118-diff.tex:314: Links and xrefs
- thesis-20101118-diff.tex:350: restructure pivot point
- thesis-20101118-diff.tex:375: simple definition of bug
- thesis-20101118-diff.tex:377: need to go into the identify

- thesis-20101118-diff.tex:385: first paragraph of chapter
- thesis-20101118-diff.tex:390: should we flip the diagram for the bug life-cycle
- thesis-20101118-diff.tex:968: We kind of dropped it from the fore during restructure discussions
- thesis-20101118-diff.tex:991: Scope out rddl
- thesis-20101118-diff.tex:1030: note that I need to cover off the
- thesis-20101118-diff.tex:1047: better
- thesis-20101118-diff.tex:1146: need another example here
- thesis-20101118-diff.tex:1219: where does this fit into this chapter now
- thesis-20101118-diff.tex:1250: this needs to fit in heresomewhere
- thesis-20101118-diff.tex:1416: consider rename of categories
- thesis-20101118-diff.tex:1421: more rework todo after here
- thesis-20101118-diff.tex:1437: refs
- thesis-20101118-diff.tex:1442: TODO: need to dive into the cognitive area a bit
- thesis-20101118-diff.tex:1469: at blub for conceptual bug - just not what the programmer intended
- thesis-20101118-diff.tex:1479: not sure this is a good reflection on what I am
- thesis-20101118-diff.tex:1739: Pull in some of the mental-model things here
- thesis-20101118-diff.tex:1741: observations
- thesis-20101118-diff.tex:1762: This has been pulled in as a part of restructure
- thesis-20101118-diff.tex:1783: I am missing something here
- thesis-20101118-diff.tex:1792: common language
- thesis-20101118-diff.tex:1811: not sure if it is worth looking at
- thesis-20101118-diff.tex:1813: this needs to link back into the expectations
- thesis-20101118-diff.tex:1826: This is not quite what I had in mind
- thesis-20101118-diff.tex:1894: link back to expectations for concrete
- thesis-20101118-diff.tex:1898: again link back to expectations for
- thesis-20101118-diff.tex:1918: describe question phrasing

- thesis-20101118-diff.tex:1920: bake in cross cutting concerns
- thesis-20101118-diff.tex:1925: sort question references
- thesis-20101118-diff.tex:1945: Need more
- thesis-20101118-diff.tex:1990: type check
- thesis-20101118-diff.tex:2008: extract the rest of the questions from lint impl
- thesis-20101118-diff.tex:2229: This chapter will be disproportionately large
- thesis-20101118-diff.tex:2231: better discussion of granularity
- tefkat.tex:92: diagram of trees
- localisation\_techniques.tex:5: This chapter will be disproportionately large
- localisation\_techniques.tex:7: better discussion of granularity
- expectations\_and\_observations.tex:435: TODO: need to dive into the cognitive area a bit
- expectations\_and\_observations.tex:458: not sure this is a good reflection on what I am
- expectations\_and\_observations.tex:550: observations
- expectations\_and\_observations.tex:729: note that I need to cover off the
- expectations\_and\_observations.tex:940: consider rename of categories
- expectations\_and\_observations.tex:959: at blub for conceptual bug - just not what the programmer intended
- thesis-20101201-diff.tex:183: glossary
- thesis-20101201-diff.tex:190: abbreviations
- thesis-20101201-diff.tex:196: ES
- thesis-20101201-diff.tex:351: Links and xrefs
- thesis-20101201-diff.tex:391: restructure pivot point
- thesis-20101201-diff.tex:416: simple definition of bug
- thesis-20101201-diff.tex:418: need to go into the identify
- thesis-20101201-diff.tex:425: first paragraph of chapter
- thesis-20101201-diff.tex:430: should we flip the diagram for the bug life-cycle
- thesis-20101201-diff.tex:1008: We kind of dropped it from the fore during restructure discussions
- thesis-20101201-diff.tex:1033: Scope out rddl

- thesis-20101201-diff.tex:1073: :
- thesis-20101201-diff.tex:1365: need another example here
- thesis-20101201-diff.tex:1403: TODO: need to dive into the cognitive area a bit
- thesis-20101201-diff.tex:1428: not sure this is a good reflection on what I am
- thesis-20101201-diff.tex:1543: observations
- thesis-20101201-diff.tex:1752: consider rename of categories
- thesis-20101201-diff.tex:1758: more rework todo after here
- thesis-20101201-diff.tex:1785: note that I need to cover off the
- thesis-20101201-diff.tex:1791: refs
- thesis-20101201-diff.tex:1801: TODO: need to
- thesis-20101201-diff.tex:1840: at blub for conceptual bug - just not what the programmer intended
- thesis-20101201-diff.tex:1852: not sure this is a good reflection on what I am
- thesis-20101201-diff.tex:2105: consider rename of categories
- thesis-20101201-diff.tex:2252: Pull in some of the mental-model things here
- thesis-20101201-diff.tex:2256: at blub for conceptual bug - just not what the programmer intended
- thesis-20101201-diff.tex:2261: observations
- thesis-20101201-diff.tex:2272: This has been pulled in as a part of restructure
- thesis-20101201-diff.tex:2416: I am missing something here
- thesis-20101201-diff.tex:2428: common language
- thesis-20101201-diff.tex:2447: not sure if it is worth looking at
- thesis-20101201-diff.tex:2449: this needs to link back into the expectations
- thesis-20101201-diff.tex:2462: This is not quite what I had in mind
- thesis-20101201-diff.tex:2530: link back to expectations for concrete
- thesis-20101201-diff.tex:2534: again link back to expectations for
- thesis-20101201-diff.tex:2554: describe question phrasing
- thesis-20101201-diff.tex:2556: bake in cross cutting concerns

- thesis-20101201-diff.tex:2561: sort question references
- thesis-20101201-diff.tex:2581: Need more
- thesis-20101201-diff.tex:2626: type check
- thesis-20101201-diff.tex:2644: extract the rest of the questions from lint impl
- thesis-20101201-diff.tex:2865: This chapter will be disproportionately large
- thesis-20101201-diff.tex:2867: better discussion of granularity
- glossary.tex:1: glossary
- glossary.tex:8: abbreviations

FIX: Links and xrefs. All messed up from the cut-and-paste and restructuring massacres.

# Chapter 1

## Introduction

“Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth.” Arthur Conan Doyle, Sr / Sherlock Holmes

Concise introduction to introduce and frame research problem. Requires different approach than original, naive, literature review introductions. Concentrate more on research problem and less on introducing mdd. Still need to go through mda, transforms etc... just a bit terser. Also cover why/how declarative solution fits problem and why the debugging story of such a solution is a significant/important problem.

Identify the whole debugging mentality, aligning a developers mental/expected execution model with the actual execution model.

Old introduction from confirmation/articulation.

Model-driven development (MDD) is a modern software development paradigm where applications are specified at a more abstract level (i.e. a model) and then, using transformation techniques, one or more concrete implementations are generated. Through concentrated research efforts, MDD has become an accepted and widely applied development approach providing a number of benefits over more traditional, code oriented, software development. Through the utilisation of MDD, software developers are able to separate specific domain knowledge from technology concerns. Formal models and transformations allow for the automated application of architectural patterns and optimisations, where parallels can be drawn to the lower level use of compilation with third-generation languages. Traceability between models and their target implementations provide consistent, reliable links through system requirements, design, implementation and potentially testing.

Similar to the criticisms faced by software development as an engineering discipline, MDD requires resolution of a number of constraints relating to the productivity of developers, including: quality, reliability, performance and reproducibility of solutions; flexibility in use through both new and existing systems; extensibility and maintainability over extended product life-cycles; and the identification and traceability of problems back to their sources in the case

of failure.

NEED TO FIND A HOME: data slice vs program slice.

## Chapter 2

# Research Question

FIX: restructure pivot point, needs to play into structure, forward links, etc.

structure

1. bug lifecycle
2. bug lifecycle figure
3. discussion, scoping
4. research question/thesis, in terms of bug lifecycle figure
5. anti-questions/thesis

must make note of:

1. did not articulate research questions in a vacume
2. not necessarily chronological
3. forward reference literature review

### 2.1 Bug Lifecycle

FIX: simple definition of bug. See chapter 5 for a more detailed explanation.

FIX: need to go into the identify/localise/fix description. clearly spell out that testing is not debugging, but does play a key role in the bug lifecycle. This should lead into next section and make it easier to explain the slice of the bug lifecycle. Maybe a new section “what is debugging?”, probably would belong in next chapter.

FIX: first paragraph of chapter 5 (expectations) introduces the bias that often makes bugs appear 'obvious' after the fact, even if they were extremelly difficult to identify in the first instance. This needs to be linked and x-referenced throughout.

FIX: should we flip the diagram for the bug lifecycle, make sure expectation and observation are at the top?

See figure 2.1

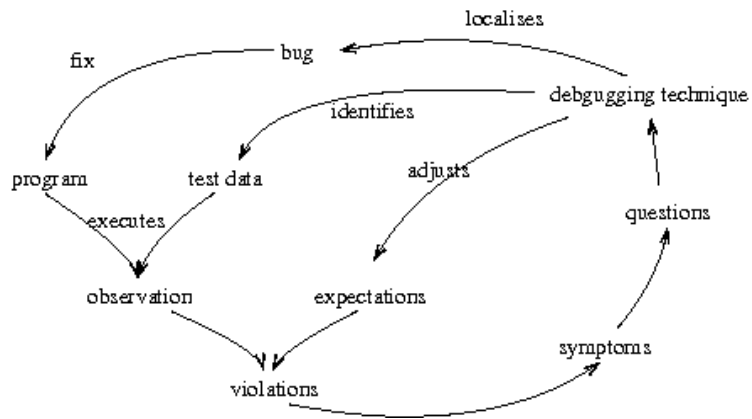


Figure 2.1: Bug lifecycle.

old, see 2.1 for update.

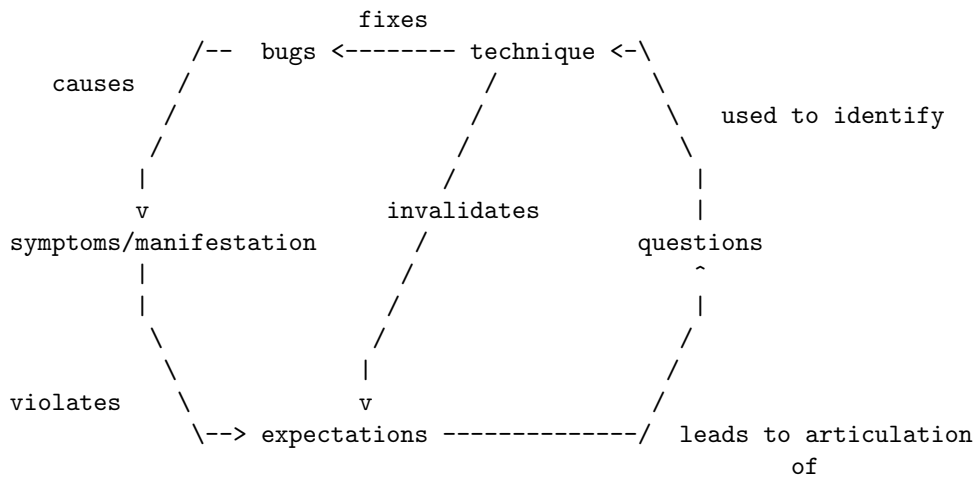


Figure 2.2: Old bug lifecycle.

## 2.2 Research Questions

Thesis/anti-thesis

## 2.3 old stuff

methodologies -¿ plan -¿ dependencies

Set out research question and its foundations. Should be able to leverage significant portions of text from confirmation (currently some of this is pasted in the introduction.)

Spell out methodology. Attack from single angle, tefkat, with view to generalize. What can? To what extent?

Re-iterate introduction point w.r.t. debugging definition. The alignment of mental (expected) execution model vs actual execution.

set expectations around bugs - simple note on how it is hard to present debugging without having appear obvious, this is the nature of debugging.

### 2.3.1 Research Problem

The key to quality in software engineering is testing. Testing is the process of identifying errors or bugs in software artifacts; including source code, configuration and now potentially models and transformations. From this identification, debugging is the process to locate and if possible propose corrections to the errors found in the testing process. In the earliest days of software engineering, testing and debugging were largely manual processes that were both time consuming and problematic. Today, testing and debugging has matured into a variety of forms, including automated testing frameworks for producing coded tests; static code analysis tools, which perform both testing and debugging tasks through processing of raw source code; programming languages with integrated testing support; integrated software development environments (IDEs), which incorporate debugging and testing mechanisms as a part of editing code; and methodologies such as test-driven development (TDD) where tests are considered as important, if not more important, than production code.

Using the lessons of early software engineering, this research intends to bring automated testing and debugging to model-driven development. The current usage of transformations in the process of MDD can be problematic, particularly during early development phases where the transformations are changing rapidly. When a problem occurs in a model-to-model transformation, the problem can go undetected as there is no automated verification of the target, or if a problem is identified (normally through visual inspection), a manual process must be undertaken to trace back through the transformation, trace model and source model to help locate the source of the problem. The first step in introducing automated testing and debugging to MDD will be to explore to what extent it is possible to use constraints to automatically verify model-to-model transformations and, in the case of a constraint not being met, automatically pinpoint the elements from the transformation and source model(s) which caused the constraint to be violated.

The research can be characterised in two parts. Firstly, the identification of problems resulting from a given transformation:

$$M_s \xrightarrow{T} M_t \xrightarrow{V} \{p_1, p_2, \dots, p_n\}$$

**Where:**

$M_s$	is the source model
$T$	is the transformation from source to target
$M_t$	is the target model
$V$	is the verification process
$\{p_1, p_2, \dots, p_n\}$	is a set of problems identified by the verification

For some given source model,  $M_s$ , a transformation,  $T$ , is used to produce a specific target model,  $M_t$ . This transformation is verifiable (tested) through a transformation,  $V$ , which may be a set of constraints or alike to identify problems,  $p_n$ . If  $\{p_1, p_2, \dots, p_n\}$  is an empty set then model verification is successful.

The second part of the research is the location and correction of problems with respect to the source instance, target instance and transformation, and can be characterized as:

$$\forall p \in \{p_1, p_2, \dots, p_n\} : \exists l(\partial M_s, \partial T, \partial M_t) \in L, \exists s(\Delta M_s, \Delta T) \in D$$

**Where:**

$p$	is an identified problem
$\{p_1, p_2, \dots, p_n\}$	is the set of identified problems
$l$	is a tuple which defines the location of the problem
$s$	is a tuple which defines the solution to the problem
$L$	is the set of all possible locations
$D$	is the set of all possible deltas
$\partial M_s$	is some part of the source model instance
$\partial M_t$	is some part of the target model instance
$\partial T$	is some part of the transformation
$\Delta M_s$	is a change to the source model instance
$\Delta T$	is a change to the transformation

For all problems,  $p$ , in the set of problems  $\{p_1, p_2, \dots, p_n\}$ , the goal is to automatically identify the parts of the source instance,  $\partial M_s$ , transformation,  $\partial T$ , and target instance,  $\partial M_t$ , which describes the location,  $l$ , of the problem and a solution,  $s$ , describing the changes required to the source model,  $\Delta M_s$ , and the transformation,  $\Delta T$ . The location of the problem shall be an automated process. The identification of the solution shall be achieved through a combination of manual and automated processes.

### 2.3.2 Research Objectives

The research aims to answer a number of key questions with respect to the debugging process in model-driven development.

1. How can models be validated after transformation to provide sufficient input into the debugging process?
2. How can the output of the validation process be combined with the models, transformations and trace information to perform post-hoc analysis for forensic debugging?

3. Do current transformation approaches provide enough trace information to enable effective forensic debugging? If not, what should they provide?
4. Does forensic debugging provide an adequate solution in model-driven development or do complex situations demand live debugging techniques?
5. How can live debugging be applied to model transformations?

To answer these questions this research intends to produce a set of algorithms and patterns to assist with the debugging process in model-driven development. These algorithms and patterns shall be provided as a practical outcome through the development of a debugging framework for model-driven development. This debugging framework and approaches will be built around the Tefkat transformation engine and utilise the Eclipse Modeling Framework (EMF) upon which Tefkat [?] is built.

### 2.3.3 Research Question

The research objectives and the questions they raise can be summarized by the primary research question.

**“To what extent can the traits of declaritive programming and model transformations be utilised to provide better debugging outcomes through an understanding of the bug-lifecycle and devleopers mental-model, and can the knowl-edge gained from the debugging process be fed back to re-duce bugs and highlight violation in expectations earlier?”**

or the less wordy –

**“To what extent can the traits of declaritive programming and model transformations be utilised to provide better de-bugging outcomes, and can the knowledge gained from the debugging process be fed back to reduce bugs and highlight violation in expectations earlier?”**

This can be broken down into a number of sub questions. [This is pretty random at the moment. I think they all tie back into the bug lifecycle and what I have worked on, it does need some strengthening though]

**“How does a developers mental-model of a system, to-gether with their expectations and observations, contribute to debugging, and can this understanding be exploited for more accurate, focused debugging questions and techniques?”**

**“Can the identification and classification of debugging ques-tions from bug symptoms be leveraged to identify an op-timal debugging technique or techniques for given viola-tions?”**

**“To what extent can declaritive focused debugging tech-niques assist in producing a debugging outcome, and can the general facets of declaritive programming or the spe-cific traits of model transformations be utilized to produce new, more appropriate techniques?”**

**“Can developers expectations be expressed with the program in a manner that compliments rather than duplicates the declaritive transformations?”**

### **Rationale**

Model-driven development has been a productive area of research over a number of years and is now having a significant impact on commercial development practices. Technology leaders are incorporating MDD techniques into their development tools and languages, with UML and code generation compulsory features for any competitive IDE. Business is adopting MDD as a way to capture the business knowledge they have always had, in formats which are more understandable, transferable and contribute to more productive software development. Independent software vendors are turning to MDD and templating techniques as a way to improve rapid application development and support ever changing technological needs. The reason for this push is the variety of benefits which MDD can bring to software development, including consistency, response to change and traceability between design and implementation. However, the next step in MDD needs to be founded on pragmatic approaches to ensure that research and commercial support continues. MDD must become more accessible, more supportable and fit within structured development approaches.

This research aims to make contributions to the MDD approach, by providing well structured techniques and tools for debugging. These techniques and tools will allow the growing number of MDD practitioners to develop and support software using model-driven approach in an easier and more efficient manner.

### **Significance of Research**

This research problem represents a largely unexplored area of MDD which is critical to the continuing evolution and further research of MDD processes and techniques. As already stated many parallels can be drawn between the development of early programming languages and MDD, however MDD has a number of key differentiators. The primary difference, which forms the basis for the anticipated outcomes of this research, is the formalisation of multi-directional traceability between not only the source and target but most importantly the actual transformation or compilation step which is responsible for the relationship. The potential utilisation of this trace information for debugging presents a unique research opportunity that could have a significant impact on the efficiency and effectiveness of debugging MDD and as a result an overall improvement of quality and productivity in software development can be achieved.

Another difference between traditional software language debugging techniques and the proposed research is the use of declarative languages for specifying model transformations. Declarative languages provide many additional complexities to imperative languages with respect to debugging, particularly concerning execution order. Although there have been research into debugging techniques, such as program slicing [?, ?] and algorithm debugging [?, ?, ?], of traditional declarative logic languages like Prolog, there has been limited practical application (none specific to model transformations). This gap in traditional debugging techniques provides further opportunity to make a significant

research contribution.

The practical nature of the research outcomes will also assist in improving the pragmatic application of MDD in software development today. The outcomes of the research aim to make an improvement on the current MDD best practices, by delivering improved productivity in development of transformations, improved maintainability of transformations over the full software development life-cycle and providing a contribution to the overall quality of software developed utilising MDD. Specifically, providing debugging techniques for use with Tefkat will assist with other MDD research that utilises the transformation engine.

The final important contribution which this research aims to make is the analysis of the current level of trace information provided by the transformation process. The identification of redundant or missing trace information will allow future MDD practices and standards to cater for both forensic and live debugging more effectively.

## Chapter 3

# Literature Review

Literature review focussing on the declarative debugging problem. Cover logic/functional debugging. Contrast with traditional debugging. Identify MDD debugging (and lack thereof).

Introduce lit review structure. Introduce obviously relevant material here (primarily debugging). Gradually expand and introduce new concepts/literature at the start of each chapter as its relevance becomes clear. Examples of this would be mental-model material and to a lesser extent testing/static-analysis material.

### 3.1 Logic Debugging

Consider 'Declarative Debugging'?

Current approaches to logic debugging. Naish. Program slicing. Algorithm debugging. Tooling.

### 3.2 Traditional Debugging

Take a look at traditional debugging. Positioning for 'complexity' chapter which explains why many of these techniques do not translate well to a declarative programming model. Step-through debugging. This section should start to draw out live vs forensic debugging.

### 3.3 MDD Debugging

Trace

Look at how traditional approaches have been applied to mdd tools. The difficulty and issues around applying these approaches. Highlight the lack of significant research. This section should really help set-up/justify the research problem.

## 3.4 OLD-WORK-DUMPING-GROUND

Original lit review. Way to wordy and dumbed down. Forget about almost all of this. Introduce testing/verification in later chapters once it becomes more apparent why it is relevant. Drop majority of generic mdd/modelling gumf.

### 3.4.1 Introduction

The purpose of this literature review is to explore the background and current state of model-driven development, including the key areas of transformations and model traceability; illustrate the key concepts of evolution, testing and debugging in software engineering; and show the relevance and applicability of traditional software engineering concepts to MDD.

The research can be divided into clear phases. The first being the testing phase, which involves the identification of problems. The second being the debugging phase, which involves the location, classification and potential rectification of problems. The third phase is the refinement of the techniques, which will involve increasing the level of automation, increasing the level of information which can be derived and minimising the required input data. The final phase will be the pragmatic application of the techniques in varying MDD scenarios.

There are two approaches to testing in software development, dynamic testing and static analysis. Dynamic testing involves the execution of test mechanisms in some type of runtime environment. This is either achieved by test applications executing the target logic with a known case and verifying the output or by instrumenting the binary logic to produce feedback as it executes normally. Static analysis involves the inspection of application artifacts outside of the runtime environment. Static analysis includes both automated tools, which can find common programming mistakes and anti-patterns or use constraints and metrics to identify possible bugs, and manual processes, which rely on physical inspection of design and implementation artifacts to verify program behavior and identify potential shortcomings. Each of these approaches has a place in software development and potentially in model-driven development and will be discussed in depth throughout section 3.4.3. The discussion will describe the initial application of testing to this research, which will be in the form of static analysis, using constraints to verify model states after transformation. From this verification the debugging process can commence.

Debugging involves the gathering of information about a particular problem including the verification details, like what was trying to be verified, the expected output, the actual output and the software artifacts involved in the process. This information is then used to pinpoint the cause of the problem and identify potential solutions. Depending on the type of testing which is used to identify the problem, the debugging process can be both manual or automated. Particularly relevant to this research is automated debugging using static analysis. Section 3.4.4 will provide an overview of the different approaches which can be taken to debugging and the details of how static analysis techniques are going to be applied to models.

The refinement phase requires the analysis of existing transformation practices. This will help identify what information is readily available as input to the debugging process, and of more importance, highlight gaps in this information

that will be important to the pragmatic application of debugging in model-driven development. Section 3.4.2 will provide the basis for the understanding required to leverage existing MDD research to assist in this process.

### 3.4.2 Model-Driven Development

The basis for this research is model-driven development (MDD). This section discusses the foundations of MDD, its current state and its close relationship with early programming languages. To support this background information, particular emphasis will be placed on the aspects of transformations and traceability in MDD and their importance to the debugging process. The purpose of this section is to provide a rigorous understanding of MDD and ensure that the MDD debugging process is rooted in a relevant base as well as being positioned for practical use.

#### Background

Model-driven development, along with other similar terms such as model-driven engineering (MDE) or model-driven software development (MDSO), describes the broad approach to the use of models in software development. The description can encompass many more specific processes. The most well known and adopted approach is the OMG's Model-Driven Architecture (MDA)[?]. Other applications of MDD include model-integrated computing (MIC)[?], aspect-oriented modeling (AOM)[?] and architecture-driven modernisation (ADM)[?].

#### The Process

The basic MDA pattern, involves three key steps [?, ?]:

1. Construction of platform-independent model(s) (PIM) describing the application without concern for target platform characteristics.
2. Translation of the PIM into a platform-specific model (PSM) which has been optimised or enhanced to support target platform characteristics. This translation can either be manual or automated through a transformation language (3.4.2).
3. Translation of the PSM into application artifacts, i.e. code, configuration files and deployment descriptors. Again, this translation can either be manual or automated through a transformation language (3.4.2) or through the use of templating.

A less formal approach to MDD is discussed by Bock's paper *UML without pictures* [?]. The paper explores the non-graphical side of models and how utilizing the non-graphical side can increase the consistency and level of detail in models to the point where implementations can be completely generated.

#### Continuing The Abstraction Pattern

From the earliest days of computers, the most commonly used solution for software problems has been abstraction and MDD is no different. MDD is effectively abstracting away from technology concerns to concentrate on the problem at hand. Within the constraints of the abstraction pattern, parallels can be drawn between third-generation languages, such as COBOL, C and Java, and MDD. In third-generation languages the abstraction or model is represented by

the human readable source code, which is then compiled or transformed for its target environment, machine code in the case of COBOL and C, or byte code in the case of Java. Virtual machine languages like Java draw further parallels to the MDD process, where its byte code represents another abstraction, this time a platform-specific model targeting the Java virtual machine. This platform-specific model could be interpreted as an executable model, or further transformation could take place through a Just-In-Time (JIT) compiler. This comparison between third-generation languages and MDD is important as it demonstrates that important lessons learned for one approach can be applied to the other and, in the case of this research, taking debugging lessons learned from traditional programming languages to achieve optimal debugging process for MDD.

### Underlying Technologies and Specifications

MDD, or more specifically MDA is supported by many technical specifications [?], of which the relevant core includes the Meta-Object Facility (MOF)[?], the Unified-Modeling Language (UML)[?, ?], MOF Query / View / Transformations [?] and XML Metadata Interchange (XMI) [?].

#### *Domain Specific Languages and the Meta-Object Facility*

At the core of MDA, as well as most other model-driven concepts, is the concept of domain-specific languages (DSLs). DSLs are languages targeted at specific problems compared with general-purpose languages, like Java and C, which are designed to be applied across many problem spaces. DSLs are used to specify systems in a platform-independent manner relevant to the particular problem, which is known as the *platform-independent model (PIM)* in MDA terms. DSLs can be constructed in a number of ways, normally either through the use of a generator to convert DSLs into general-purpose language constructs or through direct interpretation of the specific language. Within MDA, OMG use their Meta-Object Facility (MOF), current available specification version 2.0 [?], to describe domain-specific languages. The description of a model is known as metamodeling, and as such, MOF can be seen as a domain specific language for metamodeling. In a MOF based MDD approach, a model is generally described by:

- The MOF, which is a meta-metamodel used to describe the specific domain's metamodel. MOF is self describing, i.e. there is a MOF model which describes the meta-metamodel itself, which ensures there is a finite number of meta-levels. This is known as the M3 layer in MOF.
- A metamodel describing all possible model instances. This is normally a domain-specific metamodel, however there are general purpose modeling languages, the most common of those being the Unified-Modeling Language (UML). This is known as the M2 layer.
- A model instance, containing specific details of a particular case. This is known as the M1 layer.
- A concrete realisation of the model instance. This is known as the M0 layer.

Another approach to specifying DSLs is through the customisation of general-purpose modeling languages. UML provides customisation mechanisms known as UML Profiles that allow models to be marked up for domain-specific use. The UML approach to specifying DSLs has the added benefit of tool support due to the well defined structure of UML.

#### *Unified-Modeling Language*

The Unified-Modeling Language (UML) is a general purpose modeling language. The version current specification is UML 2.0[?, ?]. It is widely used with software development to specify object-oriented designs, dynamic system behaviors as well as functional processes. A key to the adoption of UML has been the standardised graphical notation which makes all UML diagrams, regardless of purpose, similar to draw and understand. UML can be extended through the use profiles, effectively creating a UML based DSL.

#### *MOF Query / View / Transformation*

MOF Query / View / Transformations (QVT) is a specification for dealing with interoperability of transformation languages and tools. QVT is divided into three parts, QVT Core, QVT Relations and QVT Operational where:

- QVT Core is the minimalistic base model for pattern matching.
- QVT Relation is a declarative mechanism for specifying relationships between two or more models. QVT Relational is effectively an abstraction of QVT Core.
- QVT Operational is a mechanism for specifying imperative implementations for transformations. QVT Operational is included for situations where it is too difficult to specify declarative relationships in a clear, concise and efficient manner.

The current version of MOF QVT[?] relates to MOF 2.0 and UML 2.0.

#### *XML Metadata Interchange*

XML Metadata Interchange (XMI) provides a mechanism for representing and persisting MOF compliant models in XML. The current XMI specification is version 2.1[?] and relates to version 2.0 of MOF .

### **Transformations**

The key to MDD being a practical improvement over traditional code based software development approaches is the ability to automatically transform between models. Carrying on the analogy of third-generation languages from section 3.4.2, transformations can be compared to the compilation stage. The key difference in MDD being that transformations will not always be from a less specific to a more specific form. This is important to the ability to visualise many views of a particular model, and more importantly it is the key to some MDD approaches, specifically ADM [?] and Model Round-Trip Engineering[?] where transformations from code to PSM or PSM to PIM are essential. Transformations also provide a mechanism for higher level re-use of architectural, design

and performance patterns. Where models can be specified without concern for lower-level optimisations that can be incorporated into lower-level transformations, again similar to third-generation languages which perform optimisation tasks during compilation.

Model transformations can be broken into two distinct groups, model-to-code transformations and model-to-model transformations [?, ?]. This research is focused on the case of model-to-model transformations. The model-to-model case is an important issue in MDD as any debugging effort is limited by the amount of verification that can be achieved for the target model as traditional runtime testing techniques can not be directly applied.

A further important categorization of approaches can be achieved through the comparison of declarative and imperative approaches[?, ?, ?]. Although the MOF QVT specification 3.4.2 specifies both declarative and imperative approaches, with the QVT relation and operations languages respectively, this research is primarily concerned with the application of the declarative approach. The declarative approach has key defining characteristics[?] such as independence of execution order and removal of model traversal complexity.

The discussion of transformations so far has been at an idealistic level, i.e. for a model a transformation can be applied and the result is another model. In practice this situation is not always the case. There are a number of potential problems for any given model transformation, including:

- An incorrect transformation rule or mapping which results in no target objects being created.
- An incorrect transformation rule or mapping which results in too many target objects being created.
- An incorrect model instance which is not fully catered for by the transformation.

These potential problems highlight the need for appropriate verification and debugging mechanisms to be in place to handle all cases.

### **Traceability**

Using the third-generation language analogy (see 3.4.2), it can be demonstrated that import debugging meta-data can be lost when transforming from a high-level representation to lower-level representation. When a Java application identifies a problem, be it through a test, byte-code verification, or some program invariant or condition an exception or error can be created to provide feedback. When source is compiled into pure byte code instructions, the programmer would have little or no way of tracing the cause of the exception back to a line of code. With this in mind Java byte code is able to be marked at compilation time to include links to specific source code lines or at least method invocations.

Following similar principles the key to debugging of any MDD transformation is the ability to trace model elements in both directions from generated target elements back to their source elements and from source elements to their resultant target. This concept was presented in early MDA papers [?] and consolidated in the MDA Guide [?] as a *Record of Transaction*. As highlighted by Aizenbud-Reshef et al.[?] it is also important for trace information to provide

meaningful context. The MDA Guide[?] takes this further by stating the importance of providing trace information back to the specific mappings which were used in the transformation as well as the source and target elements. An important note to the progression of traceability in model transformations is that the requirements have been largely driven through the need for incremental update, that is transform only what has not been transformed before [?].

Similar to traditional languages, traceability forms the cornerstone of debugging techniques for MDD. Traceability has been incorporated from the earliest MDA research and as such provides a solid concept on which to build debugging techniques from.

### Implementation

As a concrete example of transformations, Tefkat is presented as an implementation of a declarative language for model-to-model transformations in the paper *Practical Declarative Model Transformations with Tefkat* [?]. The Tefkat engine is based on a MOF specified abstract syntax.

Tefkat maps closely to the QVT relations language, and provides a record of trace that is sufficiently detailed to allow Tefkat to be used in further experimentation of verification and debugging techniques. In Tefkat the record of transformation is provided through a dedicated tracking model. This tracking model contains elements for each of the target elements, with links to the source element(s) and rule(s) which were responsible for its creation.

### 3.4.3 Software Verification

One of the important facets of this research is the identification of problems in models. To provide a solid base for model verification this section discusses the general application software verification or testing and more specifically the approach to verification of models.

#### Background

Software verification is a process used to increase the quality and reliability of systems by measuring the correctness of software with respect to its requirements. Verification should not be mistaken with validation which is the process of measuring the correctness of the requirements with respect to the actual needs of the user.

The result of software verification should involve the identification of *problems* in a system. Different software verifications techniques will result in various levels of detail, but for each problem any technique should be able to describe a constraint that has been violated and how that violation manifested itself, i.e. what was the externally visible symptoms of the constraint violation. Generally, the output of the software verification stage is the input to the software debugging stage, where the cause of a *problem* is located and corrected. An important point to make about software verification is that due to the complexities of software requirements and implementations, any verification technique will only be able to confirm the **presence** of a *problem* and will not be able to confirm the **absence** of *problems*. However, with the adoption of more stringent verification mechanisms and the refinement of software engineering processes,

the gap between the problems which are identified and those which aren't can be significantly reduced.

Traditionally software verification involves a large amount of manual interaction. An example of this would be a user sitting at a terminal, running a program and visually inspecting the result for any noticeable anomalies. This approach has a number of limitations, particularly the effort required, the reproducibility of results and the general rigor which can be achieved through a manual process. To address these problems there is a large push from the software engineering community for automated testing and runtime checking through the use of invariants, pre- and post-conditions to ensure that software verification is an ongoing and repeatable process through out the software life-cycle.

Software verification can be categorised into a hierarchy of techniques. The highest level differentiators of software verification techniques are static versus dynamic approaches. Static approaches are generally known as *analysis* techniques and dynamic approaches are what is generally known as *software testing*.

#### **Static Analysis**

Static analysis is usually applied to source or design artifacts rather than the application itself. These approaches can be further divided into manual and automated processes. Manual processes take the form of code or documentation reviews. Manual processes form an important part of any software verification process, they are used to identify a variety of complex problems which normally require human analysis to solve. Manual analysis techniques break from traditional software solutions as they are complemented rather than replaced by automated techniques. Automated techniques are normally targeted at smaller repetitive tasks where computer analysis can be applied to identify problems and due to the nature of problems found by these tools they can also complete part of the debugging process by suggesting solutions.

Automated tools fall into two categories. Pattern matching tools and metric gathering tools. Pattern matching tools look for code patterns, which can be improved in same way, or anti-patterns, which are basically patterns which should be avoided. Metric tools gather statistics about code and identify points where those statistics indicate that problems could occur, i.e. overly complex code or fragile dependency matrices. Both pattern matching and metric gathering approaches are appropriate for verification of target models. However, they do have limited applicability to the ongoing debugging process as they are mainly concerned with *potential* problems and as such are unlikely to provide enough contextual information to pinpoint the exact elements concerned.

#### **Dynamic Testing**

Dynamic testing techniques are applied either during or upon completion of execution of an application in a runtime environment. The most common forms of dynamic testing include:

- Known test case execution - selection, execution and verification of a single unit of work with a known output.
- Execution and inspection - execution of an application, and inspection, either visually or through the use of tools of the output for expected results.

- Runtime Checks - the use of invariants, pre-conditions and post-conditions during program execution.

Another not so common form of dynamic testing is the use of instrumentation to generate additional trace information. Instrumentation is a process where the executable application is modified to include markers which are interspersed between the normal application code. These markers can provide valuable information such as the path taken through the code and the number of times a certain piece of code was executed.

### **Model Verification**

A number of techniques for model verification are currently being researched. However, the majority of research is concentrated on techniques for graph transformation and have limited applicability to other types of transformation approaches. Varró and Pataricza [?, ?] have presented formal approaches to model verification using state and transition based model checking for graph transformations. A similar approach has been adopted by Zhao et al[?] using model transformations to petri-nets for analysis.

For the use of model verification as a practical input to debugging in MDD there are a number of requirements which must be met. These key requirements are the ability of model verifications to accurately identify the elements in a model which cause a particular test or constraint to fail and the ability to derive some meaning or classification from that failure which will allow the debugging process to find appropriate solutions.

### **3.4.4 Software Debugging**

The key goal of this research is the enablement of debugging in model-driven development (MDD). This section discusses the details of software debugging, from both a traditional software engineering view and a more specialised model-driven view. This discussion is broken down into locating bugs, resolving bugs and tool support for debugging. The purpose of this section is to provide an understanding of debugging in software engineering and to position this research with the latest current MDD debugging practice and research.

#### **Background**

##### **What Is A Bug?**

A bug, in computer terminology, is effectively a problem or defect in the design or implementation of a piece of software. Bugs can be found in two ways, either through a software verification technique (see 3.4.3) or through the use of software for which it was designed. The first method is preferred as it saves time, money and embarrassment for the software development team and, most importantly, can provide better input into the debugging process.

##### **What is Debugging?**

Where verification is the identification of a bug, debugging is the process by which the source of that bug is located and corrected. Debugging is an integrated process to any software development. Almost all modern languages are designed

with debugging mechanisms and tools in mind. The ease by which problems can be corrected in any technology will directly contribute to its success or failure.

### **Tracing Bugs, The First Step**

The first step in the debugging process is always locating the source of a bug. In traditional code-oriented software development this process is achieved by executing the program in a way known to trigger the failure, and then using trace information, which is a sequence of output messages indicating the programs execution path, or interactive debugging, which is a process that allows the execution of a program on a manual step-by-step process rather than executing it all at once.

An issue faced when attempting to debug problems with models and transformations using these traditional approaches is the use of declarative logic to specify mappings compared with the imperative nature of traditional languages. To address this issue there has been a large amount of research into debugging of declarative languages in general, including assertions [?], program slicing [?, ?] and algorithm debugging [?, ?, ?]. This research can be re-applied to the domain of declarative transformational languages.

### **Resolving Bugs, The End Game**

The ultimate goal of the debugging process is the removal of bugs. By matching common bug patterns and their relative solutions there are methods which could assist in resolving bugs. The current aims of this research are concentrated on providing feedback to MDD modelers on the types and locations of bugs and as such these automated resolution techniques are currently not in scope but do provide the possibility for further research.

## **3.4.5 Conclusion**

Research into model-driven development and its subsequent application to real-world development shows the tremendous potential of treating models as first-class artifacts of the development process. From the research into the key enabling features of MDD such as metamodeling and transformations it is important that core engineering principles can be applied to the associated processes to enable improved quality, productivity and maintainability of all MDD solutions.

The research presented in this paper aims at starting to satisfy these engineering goals by presenting a basis for conducting model and transformation verifications. Using these verification and debugging processes, tools and techniques shall be developed to ensure that MDD can be effectively used in ongoing development of complex systems.

This literature review has shown that debugging in MDD has little closely related research, yet it does have a number of closely related research fields which can be drawn upon. Important questions which have been raised include:

- The close relationships between MDD and third-generation languages has been highlighted. How far can this relationship be taken by applying traditional code-oriented verification and debugging techniques? In particular, what types of software verification techniques can be applied to models

and their transformation? and then, what level of detail and semantics can be drawn from different techniques? To address these questions, multiple approaches will have to be experimented with and related to the requirements for debugging.

- Is it possible and appropriate to specify constraints for model verifications as a transformation? To answer this question, experimentation with the Tefkat engine could be undertaken. Using Tefkat, firstly could constraints be easily represented as transformations and secondly could they produce meaningful, machine-processable input to the debugging process.
- Can traditional debugging procedures for declarative languages be adapted to model transformations languages? For example, programming slicing and algorithm debugging have been researched and proposed for varying logic and functional languages. Due to their declarative nature it is likely that similar lessons from the related research can be applied to model transformations.

By addressing these points, this research intends to combine a number of existing areas of research and apply them to solve the new problem of how to effectively verify models and transformations in MDD. Then, in the case of a problem being identified how to effectively debug the problem within a model-driven paradigm.

### **Further Research**

The initial goals of this research are mainly concerned with debugging and more specifically the location and identification of the participants which contributed to a particular problem. As a result, there is a significant opportunity for further research into the resolution phase of debugging. The resolution phase of debugging would most likely build upon the research proposed in this literature review.

In addition to further debugging research, there are a number of aspects not

currently covered by model verification research. Most notably is verification of model-to-code transformations and the subsequent debugging from code back to a model.

### 3.5 Mental Model References

some old references and notes for mental model work

```
@conference{borgman1985usm,  
  title={{The user's mental model of an information retrieval system}},  
  author={Borgman, C.L.},  
  booktitle={Proceedings of the 8th annual international ACM SIGIR conference on Research  
  pages={268--273},  
  year={1985},  
  organization={ACM New York, NY, USA}  
}
```

Notes:

Again primitive/simple subjects. Interesting (and common sense i guess) outcomes that a well formed mental does not matter, for simple tasks, only for complex tasks and when things go wrong, i.e. debugging.

```
@article{storey1999cde,  
  title={{Cognitive design elements to support the construction of a mental model during s  
  author={Storey, M.D. and Fracchia, FD and M{"u}ller, HA},  
  journal={The Journal of Systems \& Software},  
  volume={44},  
  number={3},  
  pages={171--185},  
  year={1999},  
  publisher={Elsevier}  
}
```

Notes:

Need to look at this more, possible insights into the things that matter vs the excessive details, for a good mental model.

```
@conference{vonmayrhauser1997pub,  
  title={{Program understanding behavior during debugging of large scale software}},  
  author={Von Mayrhauser, A. and Vans, A.M.},  
  booktitle={Papers presented at the seventh workshop on Empirical studies of programmers  
  pages={157--179},  
  year={1997},  
  organization={ACM New York, NY, USA}
```

}

Notes:

Actually dealing with experienced engineers. I think it demonstrates that any previous knowledge has a large influence on how it is thought about. Top down vs bottom up discussion, highlights that we probably need to link this sort of thinking back to the debugging triangle.

```
@article{vonmayrhauser1995pcd,  
  title={{Program comprehension during software maintenance and evolution}},  
  author={Von Mayrhauser, A. and Vans, AM},  
  journal={Computer},  
  volume={28},  
  number={8},  
  pages={44--55},  
  year={1995}  
}
```

Notes:

Excellent summary of mental models. Explores some open questions, including partial understanding. Links closely to how I think of the debugging triangle. I see it as a highly iterative process, where each level is a partial understanding, narrowing scope and increasing detail. Drefus model.

```
@conference{ko2003dae,  
  title={{Development and evaluation of a model of programming errors}},  
  author={Ko, A.J. and Myers, B.A.},  
  booktitle={IEEE Symposia on Human-Centric Computing Languages and Environments},  
  pages={7--14},  
  year={2003}  
}
```

Notes:

Links debugging process, less mental model stuff, but some good mental flow analysis linked closely to debugging. Again, to highly focused on novice programmers. Why do people not get that things do not need to be easy to learn, hard won knowledge is often best, and correspondingly the best things are often the hardest to learn but provide the highest level of momentum for each piece of knowledge.

## Chapter 4

# Declarative Model Transformations and Tefkat

Chapter 3 has discussed a number of methods for model transformation. This thesis, aims to presents practical, applied, debugging techniques, as such techniques will be discussed in terms of a single, declarative, transformation engine.

### 4.1 Tefkat

Tefkat is presented as an implementation of a declarative, logic-based, language for model-to-model transformations[?].

Tefkat shall be used as the implementation engine used for examples and discussion throughout the thesis. The rationale for using Tefkat as the implementation language, is its suitability for the criteria, as a declarative, logic-based transformation language, and the debugging problems associated with this type of language. Tefkat is well specified, has an accessible (open-source) implementation written in java and is familiar to the research team, with local Tefkat users writing practical model-transformations.

Another important part of the selection criteria is the ability to, in further research, generalise techniques for other declarative transformation approaches. This is supported by Tefkat's precise semantics and MOF specified abstract syntax that mean the semantics are defined independently of the concrete syntax. This will assist in further generalisation of this research to other languages.

A small discussion on the internals of Tefkat is required to understand the problems and potential benefits of debugging of declarative, logic-based, model transformations of traditional programs.

#### 4.1.1 Meta-Models

A core concept of model transformations are the meta-models. So as is the purpose of transformation languages, a Tefkat program takes a set of input models and produces a set of output models (note that these are a disjoint set in Tefkat, but not all model transformation languages). Tefkat enforces that these models are formally described by meta-models, and takes self-description further as the Tefkat program itself is described by a meta-model as well as

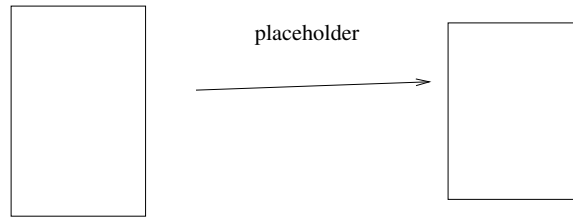


Figure 4.1: Trace Meta-model

the tracking links between objects in the target models and the triangulation between source, transform and target.

### 4.1.2 Evaluation Order

Although declarative programs have no explicit order, an implicit *partial* ordering exists in the program. This partial ordering exists due to dependencies between rules. Tefkat uses a process of stratification to encode rule dependency. Any rules dependent on the *completion* of another rule will appear in a higher strata. Note that applies only to rules dependent on the completion, rules may exist in the same strata if they consume objects created by another rule, but do not require complete evaluation before continuing. This partial ordering means that there is a set of valid Tefkat programs (syntactically) that have no useful execution, that is they can not be stratified due to some mutual dependence.

### 4.1.3 Trace

There are two types of trace information produced by Tefkat. Trackings between target objects. These trackings are user defined, they are accessible within the transformation and are used to build intermediate objects and dependencies between data. The second type of trace exists to track the actual execution, linking target object with the rule and source that contributed to their creation.

As already stated, the trace information has a formal structure specified by the meta-model in figure 4.1.

### 4.1.4 Data structures

It is also important to understand the internal execution model of Tefkat. There are a number of approaches implement logic engines, Tefkat uses an approach of .....

This is represented in the Tefkat as a forest of trees. Each tree represents the evaluation of a term, the tree expands until the first *true* leaf is evaluated. A tree may have a large number of failure leaves before a successful evaluation is obtained.

FIX: diagram of trees.

### 4.1.5 Optimisation

The declarative nature, of the language, gives rise to optimisations that eliminate evaluations, similar to how a just-in-time compiler may eliminate dead

code. An example of this is the tree structures in the evaluation engine that can be optimised through a de-forestation process. By determining the valid cross-products of the first layer of terms (up-front) the tree can be partially collapsed leading to a smaller overall tree size (with greatly reduced failures). This optimisation comes at a cost, similar to tail-call optimisation, that execution information is lost and this may skew statistical analysis and must be accounted for.

## 4.2 Running Example

At this point we introduce a running example. This example shall be used to discuss debugging concepts and introduce a set of debugging techniques for declarative model transformations. We shall also use this example as an opportunity to introduce Tefkat's syntax.

This initial example is intentionally straightforward. The example exhibits a number of characteristics allowing us to introduce debugging concepts simply and clearly, without concern for a more complex domain obscuring our intent.

- The example is small, allowing *complete* examples to be shown concisely.
- The domain is both simple and familiar, and should be easily understood by all readers.
- The transformations can be expressed with a (simpler) sub-set of the Tefkat's language constructs. In particular we will not be introducing negation, recursion, reflection or a requirement for explicit `FROM` clauses to control the creation of target elements.
- The transformations contain no implied ordering or mutual dependence, meaning that the transformations can be read and discussed top-to-bottom, in a manner similar to an imperative program.

Once we have a clear understanding of the debugging concepts, chapter 9 shall introduce a larger, more realistic example to demonstrate the utility of the debugging techniques. This larger example will cover a more complete set of Tefkat's features, and will exhibit the difficulty in debugging declarative languages where evaluation order is not (and for some transformations, can not) be linear.

The example consists of two meta-models, and a series of transformations between the two. The transformations will first be presented in their correct form, with variations on these introduced in subsequent chapters to demonstrate *buggy* transformations.

### 4.2.1 Meta-Models

The meta-models represent different forms of the same information. This means that we can write total, symmetric, transformations between the two. At the coarsest granularity, the meta-models describe models for representing a set of people, each person conveys a representation of the following information:

- Each person has a name.

Person
+name: EString
+id: EString
+sex: EString
+spouse: EString

Figure 4.2: Unisex meta-model, representing people in the running example.

Person	Person	Person	Person
+name = John	+name = Paul	+name = George	+name = Ringo
+id = 1	+id = 2	+id = 3	+id = 4
+sex = m	+sex = m	+sex = m	+sex = m
+spouse = 6	+spouse = 7	+spouse = 8	+spouse = 9

Person	Person	Person	Person
+name = Yoko	+name = Linda	+name = Olivia	+name = Barbara
+id = 6	+id = 7	+id = 8	+id = 9
+sex = f	+sex = f	+sex = f	+sex = f
+spouse = 1	+spouse = 2	+spouse = 3	+spouse = 4

Person
+name = Stuart
+id = 5
+sex = m

Figure 4.3: Example input for unisex meta-model.

- Each person has a unique identifier, such as a social security number, or national identity number.
- Whether the person is male or female.
- The person's spouse, if applicable.

The first meta-model, shown in figure 4.2, presents this in a flat structure. Each piece of information is represented by a simple, textual, attribute: name, id, sex and spouse. There are no references between the elements in the model. Any links are represented only by the data, for example the spouse is identified by their unique id rather than a reference to the actual spouse. This model will be referred to as the unisex model, and is represented by the URI <http://unisex>.

Figure 4.3 shows a valid input model for the unisex meta-model. The model contains nine elements. Four couples, with males John, Paul, George and Ringo; their respective spouses of Yoko, Linda, Olivia and Babara; and an individual male Stuart.

This input will subsequently be used as the input data for all transformation using the unisex model as a source.

The second meta-model, shown in figure 4.4, presents this in a more sophisticated, hierarchical, linked structure. In this model name and id are represented as textual attributes similar to the unisex model, however a person's sex is represented by a *type*, the **Male** and **Female** classes, rather than an attribute. Spouses are represented by direct references, rather than by the data. This

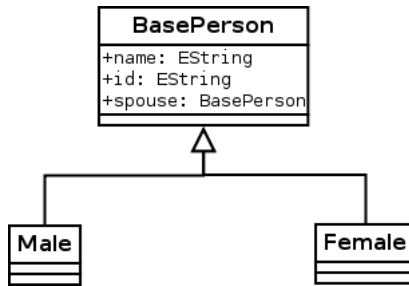


Figure 4.4: Gender meta-model, representing people in the running example.

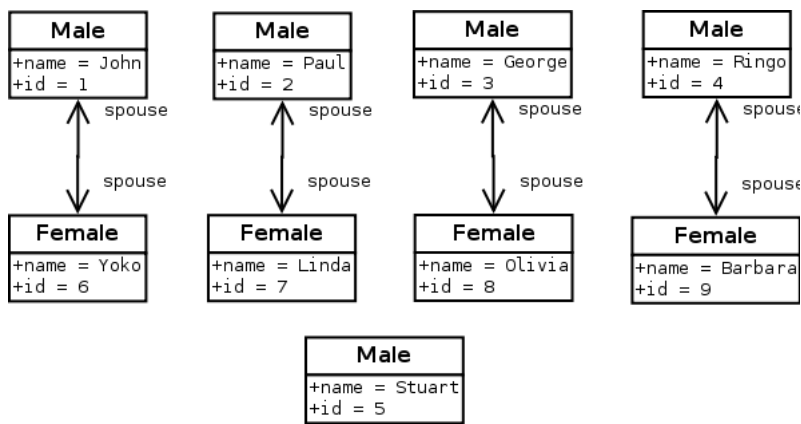


Figure 4.5: Example input for gender meta-model.

model will be referred to as the gender model, and is represented by the URI `http://gender`.

Figure 4.5 shows a valid input model for the gender meta-model. This model contains equivalent data to the unisex example, with males John, Paul, George and Ringo linked to females Yoko, Linda, Olivia and Barbara respectively as well as the individual male Stuart.

This input will subsequently be used as the input data for all transformation using the gender model as a source.

#### 4.2.2 Person to Male-Female Transformation

The first transformation takes the unisex model and produces an equivalent gender model, for example taking the model specified in figure 4.3 and producing the model specified in figure 4.5. In Tefkat, each transformation begins with a prelude declaring transformation name, each of the source and target models, as well as the namespaces to pull in each of the meta-models required for the transformation.

```

1 TRANSFORMATION unisex2gender : unisex -> gender
2
3 NAMESPACE http://unisex
4 NAMESPACE http://gender
  
```

In this case we are declaring a transformation named `unisex2gender` with an input model, `unisex`, and an output model, `gender`. There are two meta-models imported, the input model identified by `http://unisex` and the output model identified by `http://gender`. The model names declared in this prelude may be used to fully qualify elements from a particular model, but in the case of our simple example, this qualification will not be necessary.

Transformations are built from rules for transforming source elements into target or intermediate elements. Rules are defined as a set of constraints on the source and target models. The Tefkat engine evaluates the constraint terms, creating target elements as required to ensure all constraints hold for the given source and target.

Our example consists of three co-operating rules, one to produce all `Male` elements, one to produce all `Female` elements, and one to link spouses.

The first rule takes all `Person` elements with a `sex` attribute set to "m", transforming them to `Male` elements in the target model.

```

1  RULE Male
2    FORALL Person p
3      WHERE p.sex = "m"
4      MAKE Male m
5      SET m.name = p.name ,
6         m.id = p.id
7  LINKING Me2Person
8    WITH me = p ,
9         person = m
10 ;

```

This rule introduces some important concepts. The rule can be divided into two parts, input selection (source constraints) and output generation (target constraints).

Inputs are selected based upon the `FORALL` and `WHERE` clauses.

From an imperative viewpoint, the `FORALL` clause selects elements of a particular type, giving the *impression* of iteration. In this case we are selecting elements of the type `Person` and associating those elements with the expression `p`.

In the context of Tefkat, and logic programming in general, the `FORALL` establishes a constraint, in the example rule that the expression `p` has the type `Person`. All elements that hold true for this constraint are bound to the expression `p`. The term *bind*, stemming from vocabulary surrounding traditional logic programming languages such as Prolog, is used to indicate the association of elements with an expression for a particular step in the evaluation. As further constraints are evaluated, the set of elements that are bound to an expression may be reduce further.

Using the input model specified in figure 4.3 as an example, we have 9 input elements: {John, Paul, George, Ringo, Yoko, Linda, Olivia, Barbara, Stuart }. All 9 elements satisfy the constraint of being of type `Person`, and are bound to the expression `p`:

$$p \Leftrightarrow \{ \text{John, Paul, George, Ringo, Yoko, Linda, Olivia, Barbara, Stuart} \}$$

`WHERE` clauses are constructed as a series of terms that constrain each input, only inputs that can be fully evaluated and hold true for all terms will be selected.

At this point it is also worth noting that that the `FORALL` clause is just a specialisation of an equivalent `WHERE` term that filters based upon type.

This rule has just a single term in the `WHERE` clause, `p.sex = "m"`, as both sides of this term are bound values, `p.sex` from the input model, and the constant `"m"`, this term acts as a predicate, constraining the input to only those `Person` elements who have a `sex` attribute of `"m"`. Using the example input data, we start with the complete set of elements bound to `p` (as a result of evaluating `FORALL` clause): `{John, Paul, George, Ringo, Yoko, Linda, Olivia, Barbara, Stuart }`, respectively with their attribute `sex` set to `{m, m, m, m, f, f, f, f, m}`. After applying the additional constraint `p.sex = "m"` the set bound to `p` is reduced:

`p ⇔ {John, Paul, George, Ringo, Stuart }`

Note as well that the evaluation of a term can have the side effect of binding additional expressions. To demonstrate this, it is possible to break the term from the example into a conjunction:

```
1 WHERE p.sex = sex
2   AND sex = "m"
```

Written like this, the first sub-term `p.sex = sex` evaluates in a manner similar to assignment in imperative languages. The right hand expression of the term starts off unbound, thus the term can be successfully evaluated by binding the `sex` attribute to the value of `p.sex`. From our example:

TODO: How to express this????

`sex ⇔ p.sex ⇔ {John.sex = m, Paul.sex = m, George.sex = m, Ringo.sex = m, Yoko = f, Linda = f, Olivia = f, Barbara = f, Stuart.sex = m }`

The second sub-term `sex = "m"`, acting on two bound expressions would then act as a predicate.

`sex ⇔ {John.sex = m, Paul.sex = m, George.sex = m, Ringo.sex = m, Stuart.sex = m }`

Because of the constraint between `p.sex` and `sex`, `p` can only hold true for:

`p ⇔ {John, Paul, George, Ringo, Stuart }`

The output generation part of the example rule can be further divided for analysis. The `MAKE` and `SET` clauses operate on the output model and the `LINKING / WITH` clauses create intermediate structures.

The `MAKE` clause creates elements in the target model and the `SET` clause assigns values to attributes of those new elements. From the example rule, we are creating `Male m` from the source `Person p` element, setting the `name` and `id` attributes based upon values from the source `Person`.

The `LINKING / WITH` clause creates intermediate elements for use by other rules. In this example we create intermediate elements of the type `Me2Person`. Tefkat allows intermediate structures to be defined inside the transformation.

```
1 CLASS Me2Person {
2   Person me;
3   BasePerson person;
4 };
```

This class has two attributes, the source `Person`, and a target `BasePerson`, noting that a `BasePerson` is one of either a `Male` or `Female`. Our example `LINKING / WITH` creates instances of this class to link the target objects created

from specific source objects, these intermediate links will be used by our third rule to create the correct references to spouses.

The second rule is a close analogue of the first, this time taking all **Person** elements with a **sex** attribute of "f", transforming them to **Female** elements in the target model.

```

1  RULE Female
2    FORALL Person p
3      WHERE p.sex = "f"
4      MAKE Female f
5      SET f.name = p.name ,
6         f.id = p.id
7  LINKING Me2Person
8    WITH me = p ,
9         person = f
10 ;

```

This rule does not introduce any new concepts from Tefkat, so we can present it as is and move to the third rule.

As already discussed, the third rule is responsible for establishing the references between target objects used to represent the **spouse** relationship.

```

1  RULE Spouses
2    WHERE Me2Person LINKS me = source , person = source_person
3      AND Me2Person LINKS me = spouse , person = spouse_person
4      AND source.spouse = spouse.id
5      SET source_person.spouse = spouse_person
6 ;

```

This rule has a similar structure to the first two, however instead of using source elements as input, it uses a new concept, **LINKS**, to match on the intermediate elements created by the other rules. This rule is matching on all *pairs* of **Me2Person** elements, using the **LINKS** statements to bind expressions to each of the attributes. Binding **me** to **source** and **person** to **source\_person** for the first element. Binding **me** to **spouse** and **person** to **spouse\_person** for the second element. The pairs are then filtered, by the term **source.spouse = spouse.id**, so that only *pairs* where the second element is a spouse of the first are selected. The **SET** clause of this rule then creates the reference, creating a reference to **spouse\_person** element for the **source\_person** elements spouse attribute.

Appendix A presents this transformation in full.

### 4.2.3 Male-Female to Person Transformation

The next transformation is the inverse of the first, taking **Male** and **Female** elements from the gender model and transforming them to **Person** elements in the unisex model. An example taking the model specified in figure 4.5 and producing the model specified in figure 4.3.

Once again, the transformation commences with a prelude describing the transformation and the models involved.

```

1  TRANSFORMATION gender2unisex : gender -> unisex
2
3  NAMESPACE http://gender
4  NAMESPACE http://unisex

```

This transformation consists of two symmetric rules, one for converting **Male** elements into **Person** elements and the other for converting **Female** elements into **Person** elements.

```

1  RULE Male
2    FORALL Male m
3      WHERE IF m.spouse = spouse
4              THEN spouse_id = spouse.id
5              ELSE spouse_id = "n/a"
6            ENDIF
7      MAKE Person p
8      SET p.name   = m.name ,
9          p.sex    = "m" ,
10         p.id     = m.id ,
11         p.spouse = spouse_id
12 ;
13
14  RULE Female
15  FORALL Female f
16  WHERE IF f.spouse = spouse
17          THEN spouse_id = spouse.id
18          ELSE spouse_id = "n/a"
19        ENDIF
20  MAKE Person p
21  SET p.name   = f.name ,
22      p.sex    = "f" ,
23      p.id     = f.id ,
24      p.spouse = spouse_id
25 ;

```

The above listing represents the first version of the **Male** and **Female** rules. The rules introduce the use of **IF/THEN/ELSE** for conditional evaluation. Using the **Male** rule as an example, the **IF** statement can be read as: If, and only if, the term `m.spouse = spouse` can be evaluated, then attempt to evaluate the **THEN** clause terms, in this case `spouse_id = spouse.id`, otherwise evaluate the **ELSE** clause terms, in this case `spouse_id = "n/a"`.

Returning to how terms are evaluated, it is important to understand how the **IF** term `m.spouse = spouse` can fail. In the first example, we discussed the case where both the left and right hand expressions of a term were bound, resulting in the term acting like a predicate, and the case where only one side of the term was bound, and the term acted in a manner similar to assignment, binding the unbound to term to the bound one. When neither the left or right side are bound, as would be the case where `m.spouse` is not set, then the term can not be evaluated. In a basic **WHERE** clause, if a term can not be evaluated, it simply filters the input, in the **IF** case it instead evaluates the **ELSE** terms. An incorrect implementation of the **Male** rule may assume that `spouse` is always set:

```

1  RULE Male
2    FORALL Male m
3      WHERE m.spouse = spouse      // *INCORRECT*
4            AND spouse_id = spouse.id
5      MAKE Person p
6      SET p.name   = m.name ,

```

```

7         p.sex      = "m",
8         p.id       = m.id,
9         p.spouse   = spouse_id
10    ;

```

This naively simplified rule results in incorrect evaluation. In this case, the term `m.spouse = spouse` can not be evaluated when `spouse` is not set. This counter-example demonstrates the appropriate use of a conditional clause for these example rules.

Appendix B presents this transformation in full.

#### 4.2.4 Factored Male-Female to Person Transformation

Having established the correctness of the first version of the **Male** and **Female** rules, we will introduce new Tefkat concepts, **PATTERNS** and **TEMPLATES**, to refactor the initial implementation and remove duplication. This model is equivalent to the one specified in section 4.2.3, where we are taking the model specified in figure 4.5 and producing the model specified in figure 4.3

**PATTERN** definitions allow us to extract common source constraints for re-use across multiple rules and contexts. The rules in this example show a common pattern that fits with **PATTERN** use:

```

1  PATTERN spouseId(p, spouse_id)
2     WHERE IF p.spouse = spouse
3           THEN spouse_id = spouse.id
4           ELSE spouse_id = "n/a"
5           ENDIF
6  ;

```

Similar to how **PATTERN** allows re-use of source constraints, **TEMPLATE** declarations allow re-use of target constraints. Using a **TEMPLATE** declaration with our example:

```

1  TEMPLATE person(mf, sex, spouse_id)
2     MAKE Person p FROM f(mf)
3     SET p.name    = mf.name,
4         p.sex     = sex,
5         p.id      = mf.id,
6         p.spouse  = spouse_id
7  ;

```

The rules can now be re-written, to use the **PATTERN** and **TEMPLATE** declarations:

```

1  RULE Male
2     FORALL Male m
3         WHERE spouseId(m, spouse_id)
4         MAKE person(m, "m", spouse_id)
5  ;
6
7  RULE Female
8     FORALL Female f
9         WHERE spouseId(f, spouse_id)
10        MAKE person(f, "f", spouse_id)
11 ;

```

Appendix C presents this re-factored transformation in full.

## 4.2.5 Evaluation Order

Note that the discussion of the examples has informally assumed a top to bottom, left to right, evaluation order for the sake of familiarity. However, as already noted, the evaluation may not occur in this order.

TODO: rework this into a more complete example... For example it is possible to evaluate the second sub-term before the first. In this case, the left-hand expression `sex` is unbound and the right is bound as a constant `"m"`. This can be evaluated by the left-hand expression being bound to the value of the right. In this alternative evaluation strategy the first term then acts as a predicate as both sides of the term are bound.

## 4.2.6 FROM

TODO: Work this back into an example - or more likely - shift to the complex example.

Note the implication that `Male m` was created from `Person p`. This rule asserts this implicitly as there is only a single source element selected. More complex rules may require an explicit `FROM` clause. This rule could be re-written with an explicit `FROM`:

```
1  RULE Male
2    FORALL Person p
3      WHERE p.sex = "m"
4      MAKE Male m FROM f(p)
5      SET m.name = p.name ,
6          m.id = p.id
7  LINKING Me2Person
8    WITH me = p ,
9          person = m
10 ;
```

This states that the `Male m` element is created from a function of `Person p`. `FROM` clauses are an important, but complex part of Tefkat. They will be discussed in more detail in further examples, but it is sufficient to understand that they exist and what they state for these simple rules.

## Chapter 5

# Bugs, Observations and Expectations

Program ‘bugs’ encompass a wide variety of program failure, or perceived failure. Bugs have always been a part of software development, and those that have written even the simplest program have likely experienced the effects of a bug. However, bugs often escape precise definition, it is straight forward to identify examples of bugs, but it is more difficult to enumerate the differing aspects of failure that are marked as bugs. Bugs often suffer from strong bias based on past experience and as a result new or unusual bugs that are often difficult to diagnose appear obvious in hindsight. This trait of bugs means that (better) observation, is a key to an improved debugging process.

This chapter explores what we mean by a ‘bug’, deriving a clear definition that will be used to present a mental model for debugging. We will expand the model by looking at program observation and the expectations that developers place on their programs.

### 5.1 What is a bug?

Development of a program is often thought of as a linear process, as shown in figure 5.1. Program specification, formal or more often as a collection of informal inputs, is used by developers to build a mental model, or design, of the program. The mental model is applied, to produce a concrete implementation. The program is then executed to produce some output. This output is observed and measured for correctness.

At a fundamental level, a bug is a difference between the specification and the observed output. The specification forms a set of expectations that can be measured against the actual outcome or behaviour of the program. In the naive model of program development, bugs span from the initial setting of expectation

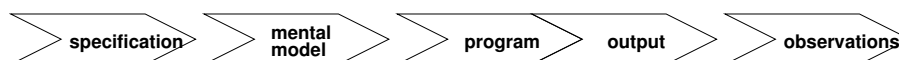


Figure 5.1: Linear view of program.

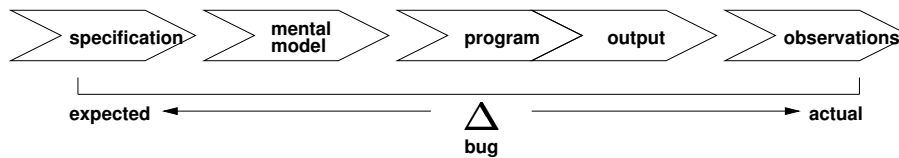


Figure 5.2: Naive view of a bug.

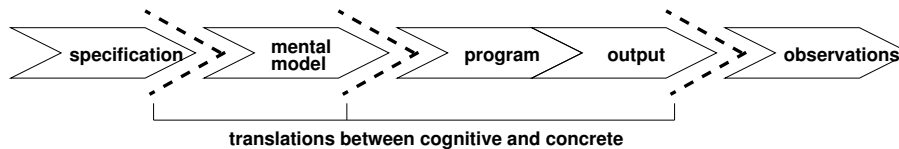


Figure 5.3: Linear view of program emphasis on translations between cognitive and concrete aspects.

to the final outcome as shown in figure 5.2.

This model starts to draw attention to the transition between cognitive and concrete steps in the process. These *transitions* are points where a difference can occur between the expected and actual, a bug, for a given stage of the development process. Figure 5.3 highlights these transitions.

As we construct a more sophisticated model for program development, a number of facets emerge in the formation of a correct program. We will discuss each facet in more detail, but first it is important to draw out a pattern that underpins the process of developing a program.

Figure 5.4 shows three actors in producing a correct program: a concrete input that forms the expected view of the program; a mental-model of the expected program held by the developer; and a concrete output that is the actual program. Building on the canonical bug definition of expected against actual, correct programs can be seen as the intersection of the expected and actual views,  $A \cap C$ , shown in figure 5.5.

Although figure 5.5 shows correct output, it also highlights an interesting dichotomy that a correct program can be achieved (however unlikely) *without* a correct mental-model. This situation arises when there are two mismatches (read: bugs) that effectively negate each other. So, whilst the correct solution is foremost, it may *not* be bug free, and as such is likely to be fragile and further change will cause these currently *hidden* issues to manifest according to the naive end-to-end definition of a bug. So if correct programs are represented as  $A \cap C$ , ideal programs can be represented by  $A \cap B \cap C$  as shown in figure ???. This view shows a more sophisticated definition of a bug: a difference between concrete and cognitive aspects of program development. The new definition ties back to figure 5.3 where we first highlighted the schism between cognitive and concrete aspects of program development. It is this more sophisticated definition that we wish to use when discussing the removal of bugs, where debugging is moving programs towards the *ideal* programs where expected, actual and the developers model align.

This model can also be used to assist in the classification of bugs. Figure 5.7 highlights the difference between the expected and the cognitive model. This bug, a cognitive error, indicates there is a fundamental difference between the ex-

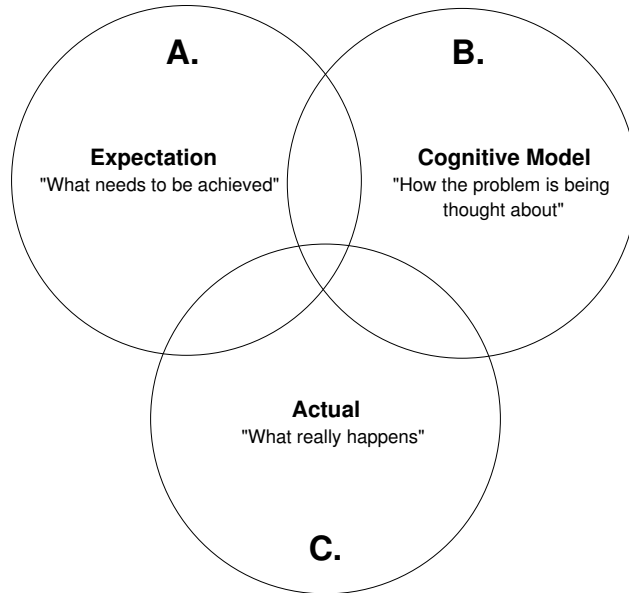


Figure 5.4: Shared model for development elements.

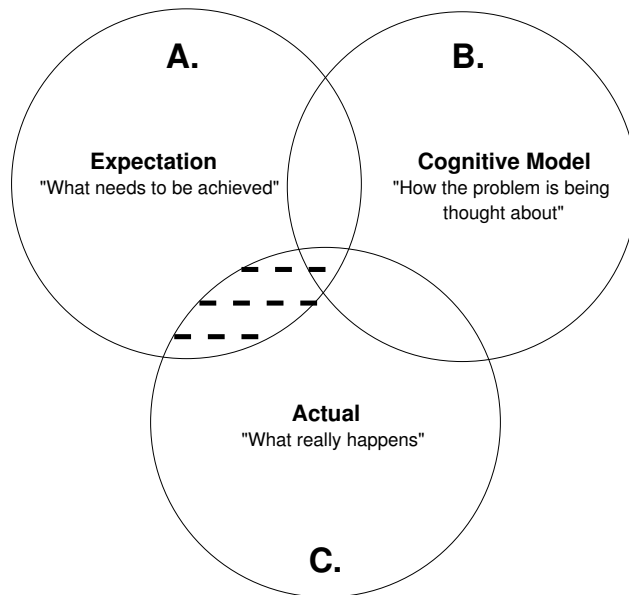


Figure 5.5:  $A \cap C$ , the set of correct programs.

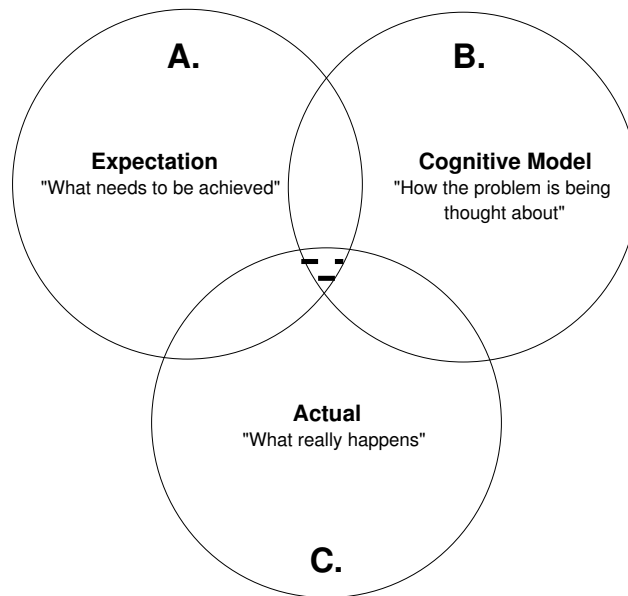


Figure 5.6:  $A \cap B \cap C$ , the set of ideal programs.

pected program and the mental-model of the program formed by the developer. Figure 5.8 highlights the difference between the cognitive and actual model. This bug, an implementation error, indicates a failure to correctly translate a mental-model to a concrete implementation.

The expected, actual, cognitive division can be applied to a number of factors in the development in correct programs. The core set of factors in the development of a program we would like to highlight is the input data, the development platform, the problem or specification being addressed. Complementing these development factors, we can trace backwards through the divisions to assist with verification of program, most importantly, observation demonstrate the expected vs actual vs cognitive breakdown.

## 5.2 Cognitive Error

The entry into software development, be it new software or maintenance, is problem definition and articulation. Developers are required to synthesize many inputs to this process: functional - specifications, requirements, knowledge of existing program function and data; technical - knowledge of the programming environment and tools, system constraints, knowledge of existing program form and implementation. These inputs vary in accuracy, from detailed specifications and test cases to vague ideas of how a user is to interact with a feature, and credibility, from formal, reviewed, documentation through to word of mouth socialised from other people in and around a team of developers.

The set of inputs that contribute to defining a program can be ambiguous, conflicting or simply impractical. Yet, a developer (or more often a team of developers) forms mental models from this understanding and build systems based on those models. Developers, sometimes over-zealously, place strong assertions

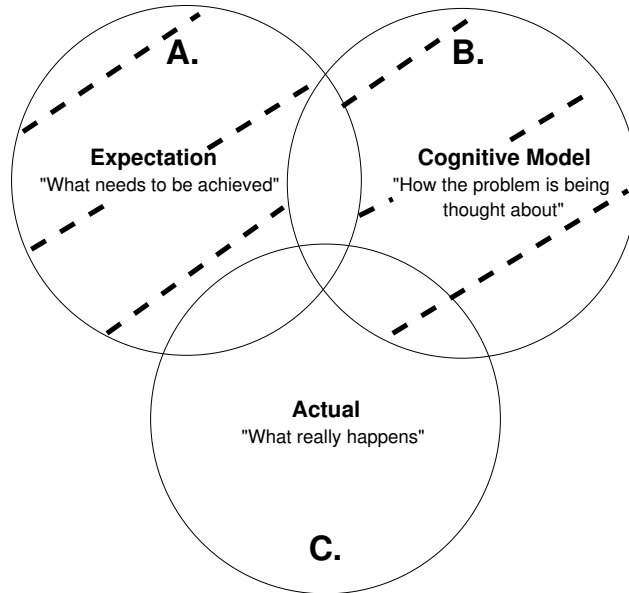


Figure 5.7:  $A \Delta B$ , Cognitive bugs.

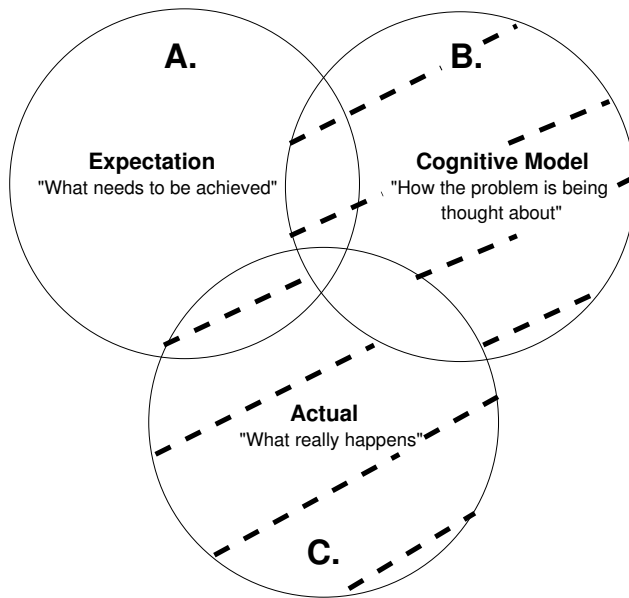


Figure 5.8:  $B \Delta C$ , Implementation bugs.

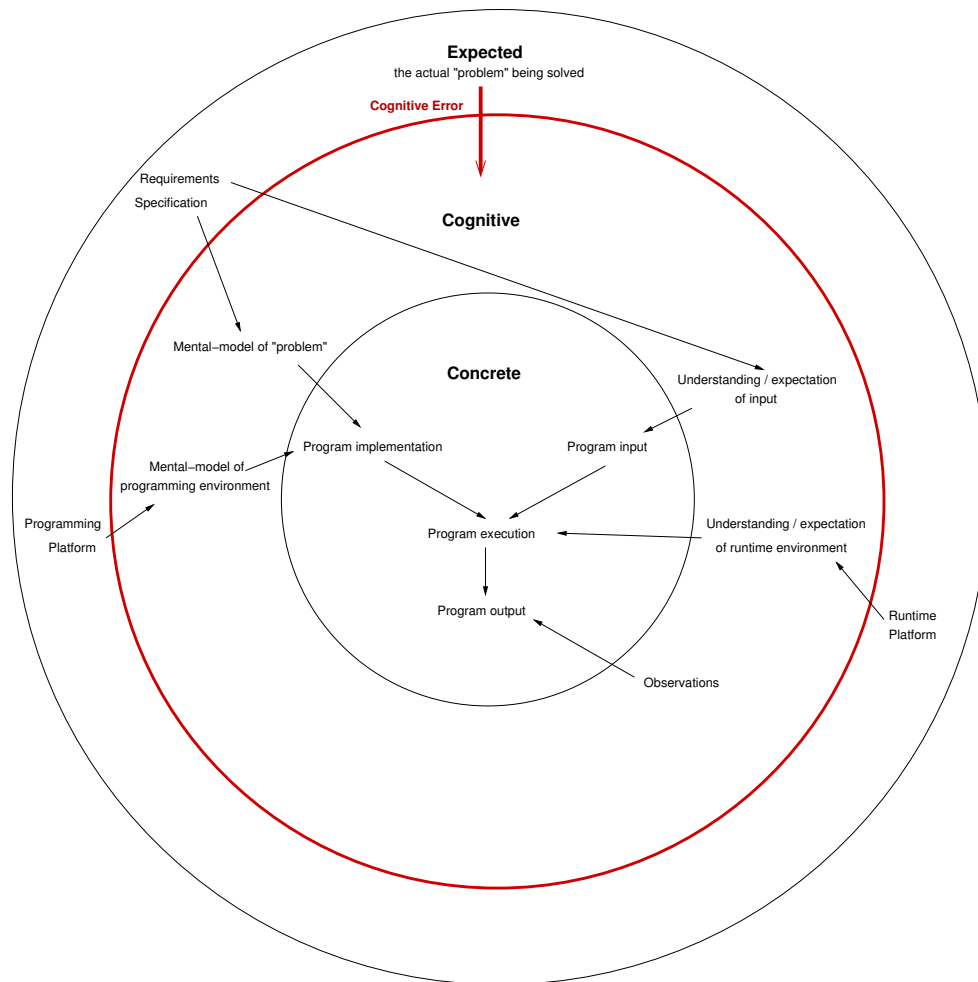


Figure 5.9: Cognitive bugs.

on the correctness of their mental-models, but the translation of knowledge and expectations to a mental-model is often a source of bugs. In this case the bug is a cognitive error, a difference between the expected program and the developers mental-model as shown in figure 5.9.

An example of a, cognitive, problem definition bugs can be shown by the requirement “sort a list of strings”. Lets first look at the input data, a file containing:

```
pear
carrot
grape
apple
Banana
```

And the corresponding linguistic ordering:

```
apple
```

Banana  
carrot  
grape  
pear

The developer forms an understand of the problem: my language knows how to sort strings, each letter has an ordinal value that can be compared and arranged in ascending order. The developers mental model now reads as:

pear	=>	[112,101,97,114]
carrot	=>	[99,97,114,114,111,116]
grape	=>	[103,114,97,112,101]
apple	=>	[97,112,112,108,101]
Banana	=>	[66,97,110,97,110,97]

So by ordering the ordinals, we now have a list of sorted strings:

Banana	=>	[66,97,110,97,110,97]
apple	=>	[97,112,112,108,101]
carrot	=>	[99,97,114,114,111,116]
grape	=>	[103,114,97,112,101]
pear	=>	[112,101,97,114]

The developers thinks they have their understanding of the problem is distilled to a workable solution and moves on to implementation. However, a critical and common mistake has been made, by over-simplifying the problem the actual desired output is incorrect. The developer goes and implements the program as they planned, fails testing - even though the program is coded exactly as intended - due to a cognitive error.

TODO work in the classical requirement mis-understanding. And the canonical funny – <http://www.projectcartoon.com/cartoon/1>

This highlights the importance of problem identification and formalisation in software development, it is often where the first bugs slip into software and are often the hardest to identify due to the fact the program is doing what the developer thinks it should be doing, but it is in fact the developers understanding that is incorrect.

Another example of this is bad assumptions on the input data. Given the requirements for a program that tracks family history, there are a number of false assumptions that may be made about names:

- That names are unique - in a group of closely related people this would be a particularly bad failure scenario.
- That everyone has a canonical name - new born babies being a classic example.
- Names are always in English, and representable in an ascii character set.

Each of these form possible failures, read bugs, of the developers mental model of the input data. Applying this knowledge will likely result in software faults, in this case perhaps by not being able to distinguish different people in the system.

Having discussed problems in domain specific mental-models such as the problem definition or input data, we must also consider faults in the mental model surrounding the development environments and tools. Programming languages, particularly for novice programmers, present many opportunities for a difference between the language semantics and what the developer may think the semantics are. Fundamental mis-understanding of programming languages and paradigms are common and varied, often due to complexities and trade-offs made by languages. A good example of this in a popular modern language is javascript. Crockford – ref js - the good parts, appendix - explains the == operator and its complex type coercion semantics.

Insert quote here....

Developers coming from other relatively modern languages such as java and python may not fully understand these semantics and mis-apply there knowledge from another language where the == operator is much simpler. This could lead to implementation bugs such as ... TODO explain the bug scenario ...

### 5.2.1 Cognitive Errors in Model Transformations

Logic based model transformations suffer similar, if not identical, issues with cognitive error to traditional programs.

, and more generally declarative languages,

TODO: this sub-section is ripped from older ramblings, still needs to be sorted out.

FIX: TODO: need to dive into the cognitive area a bit. The important aspect is simply understanding and deriving a clear picture of the relationship to the model transformation domain. There are some clear positives in this area that, declarative MT is built around the idea that transformations can be (are better off for) being defined in a way that closely relates to the specification. There is also the alternative which is that things that may be more difficult to express in a declarative MT language may lead to more cognitive bugs. (language effecting how you think? LtU WS refs?)

So although cognitive bugs are not unique to model transformations, there are some aspects of model transformations that contribute to cognitive bugs and need to be addressed. The remainder of this thesis will ignore the generic problems and implication of cognitive bugs, but will continue to investigate and discuss those aspects that are important to addressing debugging of model transformations.

The close relationship to the specification that aids in the reduction of cognitive bugs for declarative model transformations, can make application more difficult. A more abstract programming model, can potentially, lead to unexpected behaviour. And an important point here is that the unexpected behavior may provide a massive deviation from the good path for a very small problem. This is different to a more imperative approach where bugs are more likely to be 'closer' to their failure. (FIX: not sure this is a good reflection on what I am trying to say, but there is something important that I am trying to say with all of this, need to tease it out a bit more.)

## 5.2.2 Mental Models

TODO: this sub-section is ripped from older ramblings, still needs to be sorted out.

Cognitive errors

Debuggers mental model, beliefs and expectations, how they translate their logical understanding to the (somewhat) abstract execution model. Extract formal model. Exploiting the mental model for better debugging, lint fuzzy expectations expressed as rules/annotations.

TODO: Need to discuss partial understanding. Can we somehow define, read: name, levels of understanding or reasoning about the transformation. Core: Class-;Table Relationships. Classes containing attributes -; Tables containing columns. — need to map this into rules.

TODO: compiler mental-model lit.

TODO: bug tracking -; how does the information that it provides link back to model?

### Layered Models

TODO: this sub-section is ripped from older ramblings, still needs to be sorted out.

1. basic understanding
2. partial ordering
3. recursion
4. mda
5. tefkat

### Experience and Mental Models

Derived largely from experience...

Binding of concepts to names.

## 5.3 Implementation Error

Given a developers solid mental model for a problem domain, they must combine their knowledge and mental model of the programming environment to produce a concrete implementation. A failure to apply correct cognitive models into an actual program is the classical bug, as shown in figure 5.10. This can be due to structural problems, incorrect logic or more simply a typographical error.

A common typographical error is ....

Logic errors creep into programs as they become more complex .... for example

An incorrect mental model of the programming can lead to ....., such as...

Examples: 1 - issue with language mental model, lots of language puzzlers could be used as examples here; 2 - issues with implementation, common programming mistakes, off-by-one, loops, initialisation;

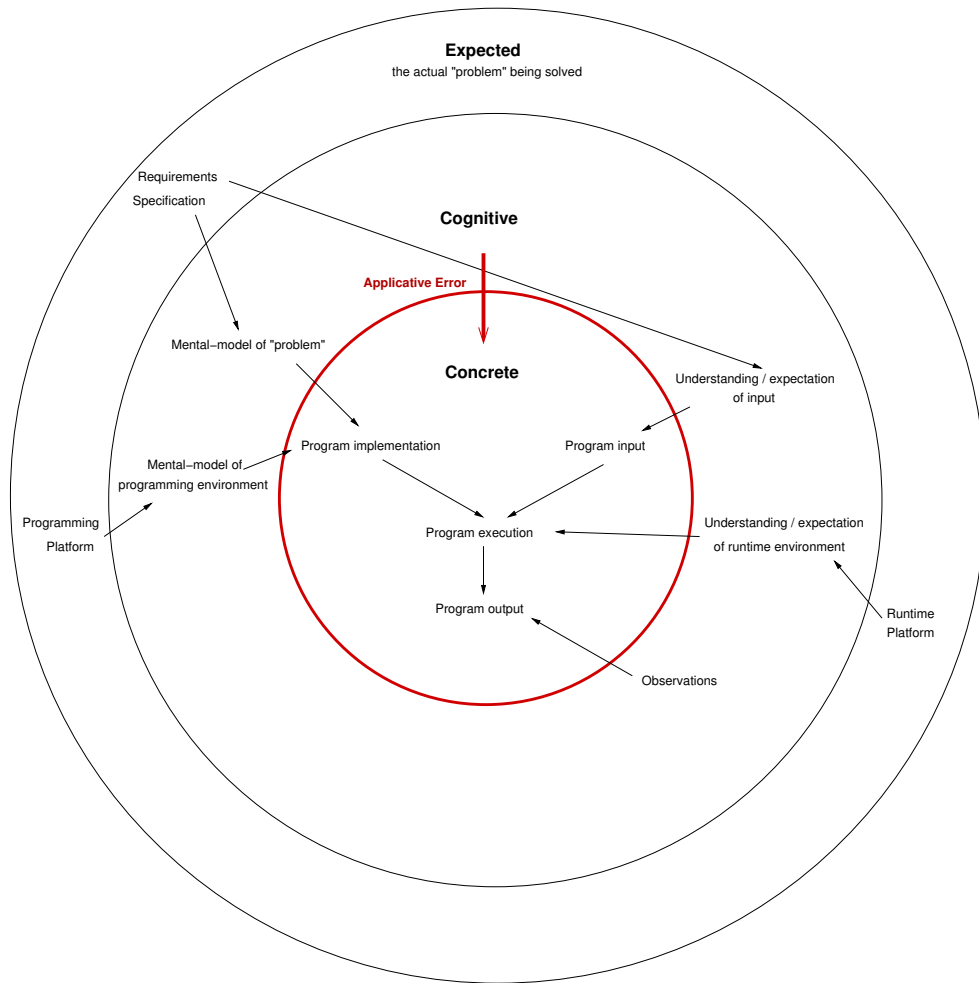


Figure 5.10: Implementation bugs.

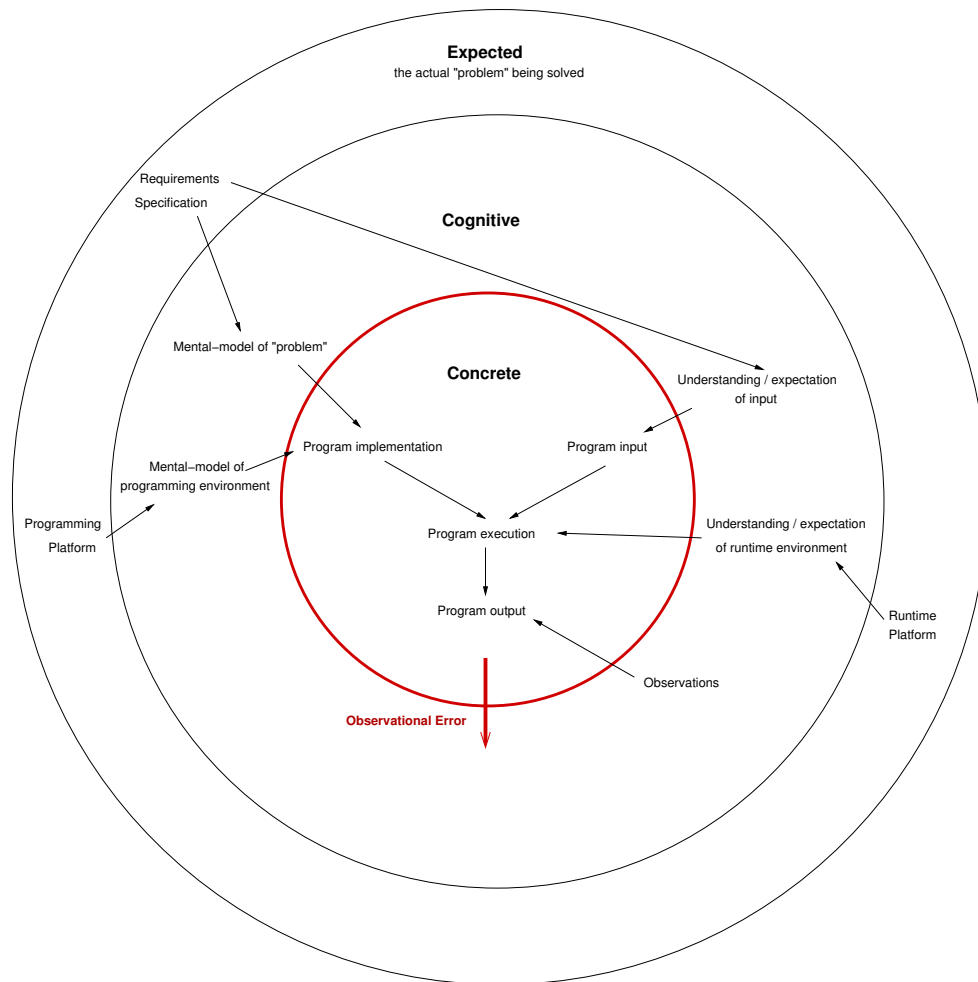


Figure 5.11: Observation bugs.

## 5.4 Observation Error

Now given a program that produces output. A developer must observe and interpret the output to verify the expectations that they have built up. Observations have a number of key influences, where it is important to note that an observation is not equivalent to the actual concrete output. Observations can have bugs, a fundamental difference between expectation (the concrete form) and the actual (the developers view of the output), and are not always a complete view of the result due to issues of volume and scale. This is shown in figure 5.11. The core influences of observations include expectations that have been built up, bias based on previous experience - such as past bugs, knowledge of the last change to a program.

For example, given the c program in figure 5.12, a programmer makes a change to add a summary line at the end of each report. This change can be seen in figure 5.13.

```

#include <stdio.h>

void
printreport(void)
{
    printf("report =====\n");
    printf("lots of output that makes\n");
    printf("observing the entire output\n");
    printf("data set difficult\n");
}

int
main(void)
{
    int i;
    for (i = 0; i < 100; ++i)
        printreport();
    printf("completed report %i\n", i);
    return 0;
}

```

Figure 5.12: A simple c program that prints blocks of output.

Due to knowledge of the change being made, “it was just a simple one liner to print out for each report”. The developer is focused on the last line to ensure the change is as expected, figure 5.14 shows the developers observation. From this observation, the developer notes that the last line is as they expect - a summary indicating the last report has just been run. Now look at the actual output shown in figure 5.15, and the disconnect becomes more obvious, there is a bug, the summary line has only been printed for the final report. The developer missed this due to an observational error. Without this, the developer could have returned to the program, and in hind-sight the bug would have been fairly obvious - incorrect formatting and missing braces meant that the summary line was outside of the loop as shown in figure 5.16.

FIX: note that I need to cover off the “not a bug” path. I need to argue that the traditional “not a bug” entry in bug trackers are actually bugs. It is covers important arcs in the bug life-cycle and is just as much about syncing the cognitive and concrete views of the program as “real bugs”. the effects of not starting the debugging process.

Given the definition of bug, and the different classes of bug, it is important to understand how bugs manifest and are communicated to the programmer. This does not imply the origin of the bugs themselves, but the point at which they are revealed or measured in the development of a program. This is to say that observation relates to symptoms, rather than bugs.

## Input

TODO: define/discuss input and output w.r.t. observations.

```

#include <stdio.h>

void
printreport(void)
{
    printf("report =====\n");
    printf("lots of output that makes\n");
    printf("observing the entire output\n");
    printf("data set difficult\n");
}

int
main(void)
{
    int i;
    for (i = 0; i < 100; ++i)
        printreport();
    printf("completed report %i\n", i);
    return 0;
}

```

Figure 5.13: A simple c program that prints blocks of output with summary line.

```

...

report =====
lots of output that makes
observing the entire output
data set difficult
completed report 100

```

Figure 5.14: Developers observations of the output.

```

report =====
lots of output that makes
observing the entire output
data set difficult
report =====
lots of output that makes
observing the entire output
data set difficult

...

report =====
lots of output that makes
observing the entire output
data set difficult
report =====
lots of output that makes
observing the entire output
data set difficult
completed report 100

```

Figure 5.15: The actual output of the program.

```

#include <stdio.h>

void
printreport(void)
{
    printf("report =====\n");
    printf("lots of output that makes\n");
    printf("observing the entire output\n");
    printf("data set difficult\n");
}

int
main(void)
{
    int i;
    for (i = 0; i < 100; ++i)
        printreport();
    printf("completed report %i\n", i); // BUG !!
    return 0;
}

```

Figure 5.16: A simple c program with bug.

## Output

Output is all externally observable aspects. This includes program output, side-effects, and runtime aspects such as space and time of program execution.

## Issues in Observation

It is important to note that observations are not absolute, more simply “observations are not output”. Although observations provide an important link between the program and debugging, an observed *bug* can equally be a mismatch caused by an incorrect program as it is by incorrect observation.

There is also difficulty in observation for the general case. How easy it is to *spot* a bug? Volume of data? Clarity of data? Chapter 7 further investigates techniques that aid in observation.

TODO: explore further.

1. volume?
2. filtering?

## Observation and Expectation

TODO: link up the two sections.

## 5.5 Expectations

Each of these views contribute to a chain of expectations for the program (noting that these expectations may be correct or incorrect), as shown in figure 5.17. Tracing through these links, each expectation provides a point to audit and test the translation between layers. Where any violation creates an entry point into the debugging process. – TODO mush, trying to say something important but it is not coming out at all.

Examples: 1 - incorrect observations being built up, something to do with input, all tables have a primary key or similar.

---

Observation raises the question, what do developers measure correctness against? What are their expectations, where are they derived from and how do they validate them.

Developer expectations are difficult to fully qualify. They range from explicit and measurable properties, such as specifications, tests and constraints through to implicit, and subjective properties of output such as orders of magnitude and distributions of values. Some expectations are even directly triggered from observations where developers examine the output and gather expectations of size and structure based upon previous executions of the same or similar programs.

Assumed understanding

Beliefs about the program

1. assumed understanding
2. spec
3. tests

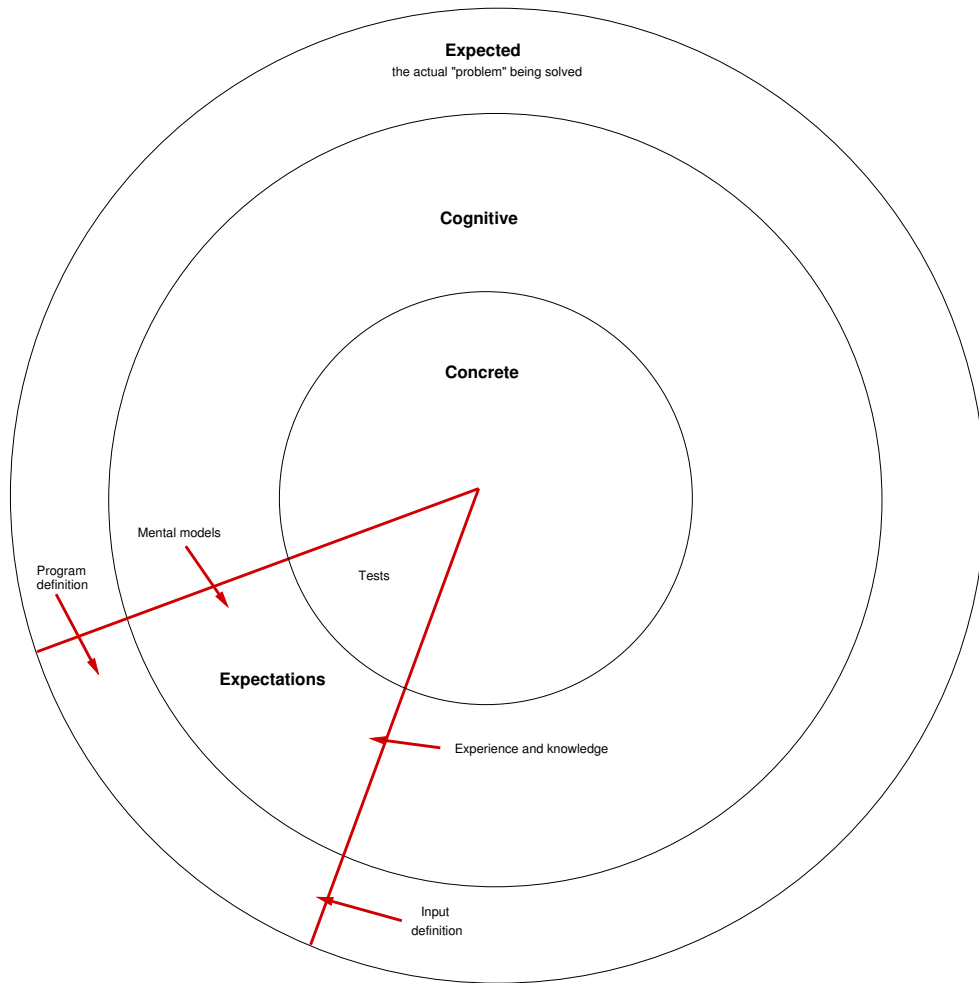


Figure 5.17: Expectations.

4. knowledge of program
5. verbal / team orientated

### 5.5.1 Explicit vs Implicit Expectations

### 5.5.2 Output vs Program Expectations

### 5.5.3 Positive Programming Philosophy

1. see success cases well
2. fail to see failure cases

### 5.5.4 Expectations and Focus

TODO: Maintenance vs development, and the varying levels of focus. Also link in to mental model, wrt. maintenance being about improving the mental model as you move forward.

TODO: any literature around the benefits of referential transparency may fit in here. Closely related to the ability (or lack of ability) to focus on particular parts of the program given knowledge of the change or failing sub-section of a program.

### 5.5.5 Expectations and the Definition of a Bug

Similar to the fact that “observations are not output”, “programs are not expectations”, which complements our definition of a bug. Programs are not a precise statement of our expectations, although the aim of logic based transformations languages is for this to be as close as possible, and the mismatch between what programmers think there program is going to produce and what it does produce is the classical *bug*.

If programs are not expectations, then it goes to demonstrate the importance of a developers mental model to connect the abstract notions of expectations with their concrete program. Expectations are developed and refined over time and link closely with how a developers mental model matures with knowledge of both the specific program and the program environment in general. The next section shall explore the mental models involved with debugging.

## 5.6 Origins of a bug

Figure ?? shows a slice of the bug life-cycle. This slice covers bug introduction through to a violation of expectations that triggers debugging (or not in the case of undetected bugs). To design successful debugging techniques it is important to understand how and why bugs originate. By identifying these origins, we can demonstrate the relationship between bugs and a difference in a programmer’s mental model to a concrete implementation.

The debugging techniques presented in chapter 7 leverage this by trying to bridge the gap between the programmers mental model and the actual reality of the program.

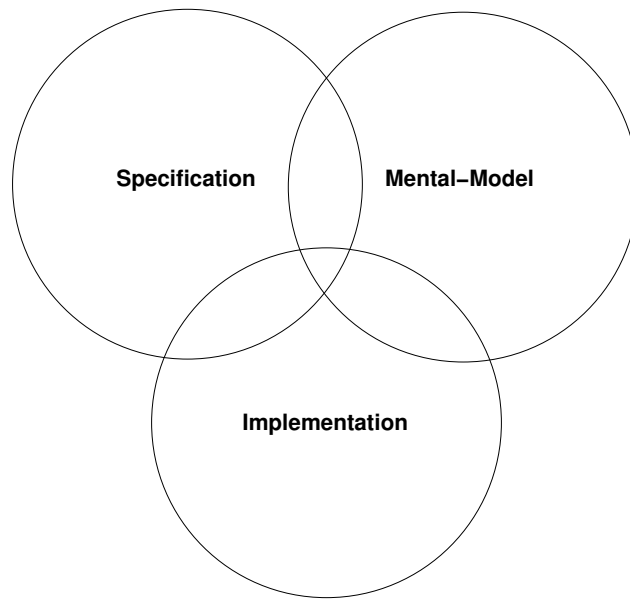


Figure 5.18: Overlapping models.

## 5.7 Left-over diagrams

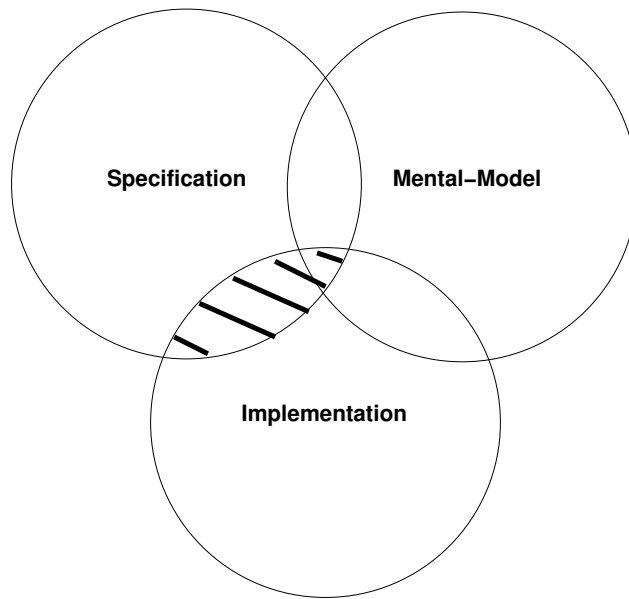


Figure 5.19: Overlapping models.

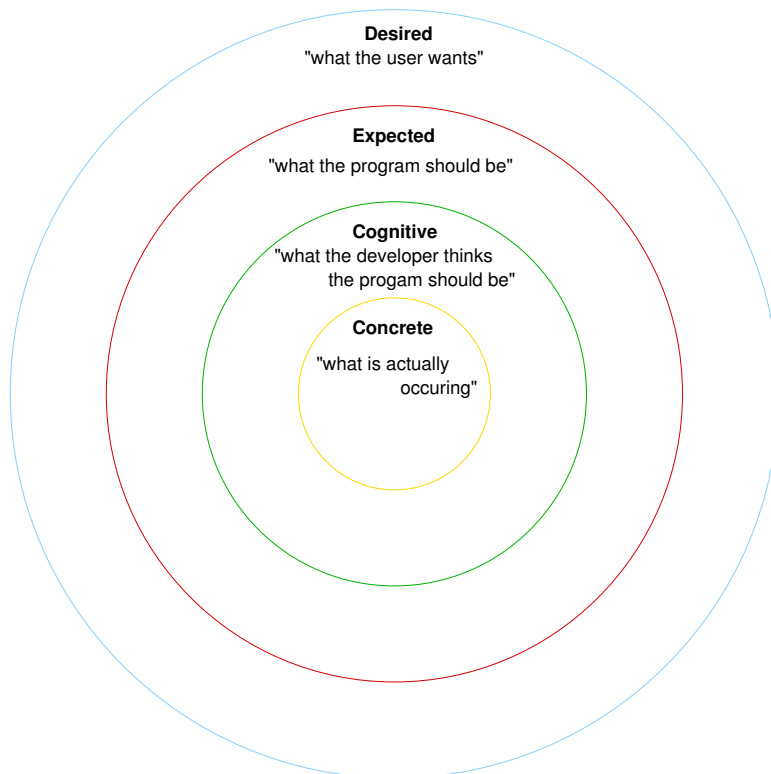


Figure 5.20: High level views of a program.

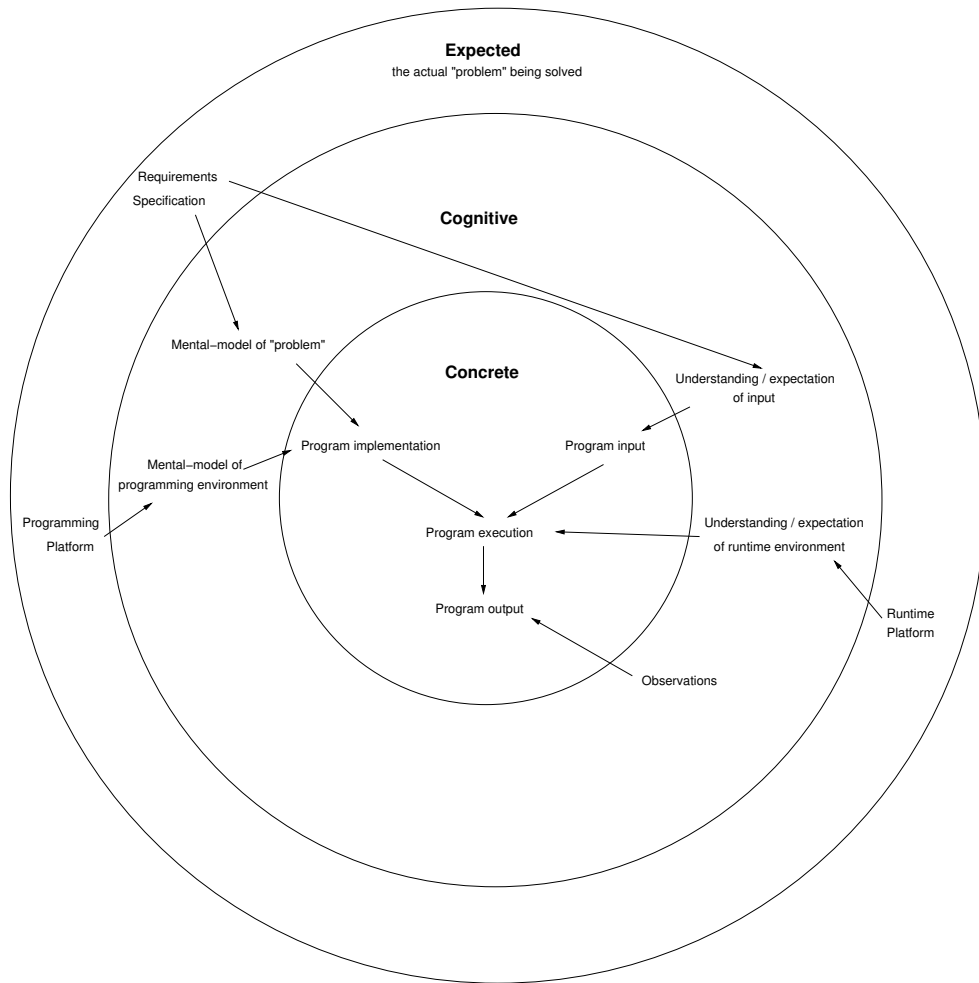


Figure 5.21: Reality vs Cognitive vs Concrete views.

## Chapter 6

# Symptoms and Questions

Having established expectations for a transformation, bugs manifest as violations or inconsistencies in those expectations. These symptoms may be identified through testing procedures, metric analysis or often just observation of output that does not meet the pre-conceived notions of a developer. No matter how they are identified, the symptom share a set of common traits by which they can be classified.

This chapter aims to establish a link through expectations, symptoms and finally the debugging questions which fall out in the course of observing symptoms. This link is then used in subsequent chapters to establish a basis and selection criteria for debugging techniques.

### 6.1 Traits of a Failure

Bug symptoms would be familiar to programmers in any domain, in model transformations symptoms range from smells of output or program fragments that do not look right, to outright failure to produce anything useful. There are number of traits to be considered in classifying symptoms.

FIX: not sure if it is worth looking at/mentioning a link between types of languages / language design and the distribution of these symptoms

FIX: this needs to link back into the expectations/observations section when complete.

#### 6.1.1 Location

There are a number of observable artifacts in model transformations. The most obvious of these are the trace and output models, however it is possible that symptoms may appear be first noticed in the source model or transforms. An example of this is a coverage report which shows a rule that is never executed but the developer considers critical to the successful outcome of the transformation.

Chapter 4 provides a more detailed description of the model-transformation environment. Each artifact involved in a transformation can demonstrate symptoms.

FIX: This is not quite what I had in mind, kind off went off course. Need to rework into a better description of the details of where/how symptoms mani-

fest and importantly the granularity at which bugs show up. The granularity discussion is important, I want flesh out the discussion of localising to rule, vs localising to term. When you can and can't (or don't want to) narrow due to evaluation orders. Also it needs to link in with observation chapter.

### **Target Models**

The target models are the most obvious, and common, location bug symptoms can occur. By definition, any bug will manifest in the target model, however there are many cases where the target model may be correct but a transform bug exists (due to the content of the source models) or where the target model does not violate any constraint or test and as a result bug symptoms are observed in other artifacts first.

### **Trace**

Trace output can highlight buggy patterns and structures that may not be detected or obvious in target models. Trace also forms an important debugging function, allowing navigation between other artifacts to identify cross-location symptoms.

### **Transform**

The transform, or more likely, runtime analysis of the transform highlights bug symptoms such as under or over executed rules.

### **Source Models**

Symptoms in source models are limited. However they are often a trigger that highlights incorrect expectations and lead to further investigation to ensure that transformations do handle source models with the discovered structure or content.

## **6.1.2 Failure Mode**

Another important aspect of symptoms is how the failure actually manifests. Failure modes are:

### **Incorrect output**

The output is valid, and meets hard constraints, but does not meet have the correct meaning or violates some expectation for the given input.

### **Invalid output**

The output is invalid. It is not well formed or violates hard constraints such as cardinality or containment.

### **Correct, but not timely output**

The correct output is produced, but does so in a way that invalidates the output, such as failing to meet latency expectations.

### **Termination**

The transform terminates early, without producing any output. This can be due to the inability to determine an order of evaluation, but commonly the novice will experience termination due to technology specific errors. This thesis does not investigate technology specific symptoms.

### 6.1.3 Type

As discussed in chapter 5, there are number of expectations that a developer may have for different aspects of the transformation. These expectations translate into types of symptoms that can be observed.

#### **Magnitude**

Magnitude, or cardinality, of elements found in the output models. Generally expectations will be none, one, a few or many.

#### **Structure**

Incorrect output structures. Violates expectation of target structure and relationships between core elements.

#### **Detail**

Incorrect or incomplete details. Violates expectations of target attributes and values.

#### **Balance**

Relative sizing of related elements. This is the ratios between core elements on the source and target sides. Expectations of balance vary whether a transformation is explosive or implosive, but are very good indicators of bugs. FIX: link back to expectations for concrete examples such as the classic I expect almost as many tables as classes.

#### **Coverage**

Transformation execution metrics demonstrates violations of a developers expectations of rules, such as the main-line vs corner-case rules. FIX: again link back to expectations for better explanation examples.

## 6.2 Debugging Questions

As developers gain experience, they learn to ask questions in an attempt to localise and trace bugs to their source. Utilising our experience with transformation languages, Tefkat in particular, we can extract these debugging questions and draw a correlation to bug symptoms.

It makes sense to break down these debugging questions into a number of categories, characterised by the debugging goal. The breakdown shows a relationship between the type of symptom and the debugging questions used to investigate. The following section details the debugging questions in four categories: detail, structural, trace, execution and smells.

FIX: describe question phrasing, terminology and variable usage.

FIX: bake in cross cutting concerns, links to symptoms.

### 6.2.1 Detail Questions

Detail questions are concerned with attributes and their values.

- 1 Why doesn't object  $x$  contain any references?
- 2 Why does a particular reference point to object  $x$ ?

- 3 Why isn't reference  $r$  set?
- 4 Why are references  $r_1$  to  $r_n$  out of order?
- 5 Why does attribute  $a$  have value  $v$ ?
- 6 Why isn't attribute  $a$  set?
- 7 Why was the single valued reference,  $r$ , assigned more than once?
- 8 What violated meta-model constraint  $c$ ?

### 6.2.2 Structural Questions

Structural questions investigate the existence (or non-existence) of objects in the target as well as the relationships between objects in the target that make up the *structure* of the output models. Structural questions are important to investigating violations in expectations of balance. The relationship between source and target structure forms the essence of a transformation and carry the informal expectations that transformation developers place on the balance of core elements. This expectation can be demonstrated by the common expectation that the elements in a model may have an approximate one-to-one relationship or maybe a many-to-one relationship. FIX: Need more context here. Examples will help. i.e. table for most classes.

- 1 Why are there no objects of type  $t$  in the target?
- 2 Why are there so many objects of type  $t$  in the target?
- 3 Why are there so many objects of type  $t$  in the target?
- 4 Why is there only one object of type  $t$  in the target?
- 5 Why didn't source type,  $t$ , result in any target objects being created?
- 6 Why isn't object  $x$  contained?
- 7 Why is there an instance,  $x$ , of an abstract class  $c$ ?
- 8 Why is there an instance,  $x$ , that has been created with two different classes  $c_1$  and  $c_2$ ?

### 6.2.3 Trace Questions

Trace questions relate to the links between source, target and transformation. This category of questions tends to be exploratory and are most often used to localise symptoms and provide the best points to provide program or data slicing. Trace as a core concept of model-driven development provides an important debugging platform that allows questions of origin and interaction points to be communicated and answered.

- 1 Given a target object, what source objects contributed to its creation?
- 2 Given a source object, what target objects did it contribute to?

- 3 Given an object type, which transformation rules reference the type?
- 4 Given a target object, what are the relevant slices of the transformation that could effect its creation and/or attributes?
- 5 Which source objects contributed to the creation of target objects?

## 6.2.4 Execution Questions

Execution questions covers stratification, performance and termination. Execution questions sit on the edge of the scope for this thesis. Execution questions have been further divided into two categories, those that are relevant to declarative model transformations and those regarded as technology specific questions. The technology specific questions remain out of scope but have been included for completeness.

### Model transformation questions

- 1 What prevents an execution order being determined for this transformation, i.e. stratification in Tefkat?
- 2 Why did the transformation take so long?

### Technology specific questions

- 1 Why did this transformation terminate without output (compared with empty output which is a structural question?)

## 6.2.5 Smells

Bug smell questions are those used to identify whether *buggy* or questionable patterns exist in the target or trace models. Bug smell questions are often exploratory. Unlike the other question categories, bug smell questions can, and often are, asked of transformations that show no obvious symptoms. This means that bug questions act as an *observation* entry point in the bug lifecycle.

FIX: extract the rest of the questions from lint impl.

- 1 Which source objects did not contribute to any target objects?
- 2 For all source types, which source objects of the selected type did not contribute to the creation of any target objects?

## 6.3 Old Debugging Questions Section — IGNORE BELOW - REFERENCE ONLY

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 6.1 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.

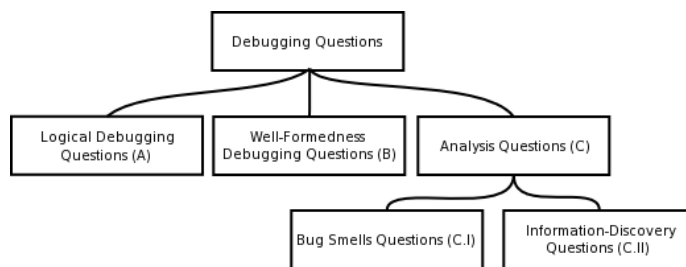


Figure 6.1: Question categories.

### 6.3.1 Classes of Bugs

The debugging questions raised in section ?? 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 be analyzed to determine effective approaches to bug localisation that may be applied to model transformations.

## 6.4 Notes

Most of the bug lifecycle edges make sense, and will be scoped and addressed. The missing piece that is not as explicit as I would like is the link from question and technique. Basically my thesis is:

1. I can produce a classification/ontology of questions that is meaningful to the types of expectations that are violated
2. I can show how this ontology can be used to identify or at least narrow optimal debugging techniques for a given class of question and back through to its expectation.

This would provide a clear, powerful link into the next section.

Classes? magnitude, distribution, structural

This chapter aims to start breaking down the types of debugging in model transformation languages by identifying a set of debugging questions.

I still have a few concerns how I express where the questions come from. Not real sure about the current hand-wavy 'derived from experience' approach.

I expect that I will have to delve into different types of transformations (explosive vs implosive) and how that impacts the types of debugging questions that are likely to be asked.

I would like to be able to provide a hypothesis for the tendency towards 'lots and lots' or 'none' type questions in this programming model.

I expect this chapter will lead to a natural progression through subsequent chapters: traditional approaches  $\hat{}$  hard work, dont map well  $\hat{}$  forensic debugging  $\hat{}$  well suited, some issues in obtaining all required data (within reasonable performance/space trade-offs  $\hat{}$  supplement with live debugging  $\hat{}$  starting to derive a mental model for debugging of declaritive languages.

Forensic debug paper plays in strongly here. See  $\hat{}$

# Chapter 7

## Debugging Techniques

FIX: This chapter will be disproportionately large. Although it probably makes sense, as it is the meatiest contribution i guess.

FIX: better discussion of granularity. or perhaps in symptoms chapter.  
localisation.

### 7.1 Traditional Debugging Approaches

This chapter takes a more in-depth look at applying traditional debugging approaches to mdd. It is positioned earlier in the 'complexity' chapter, where it outlines the difficulties involved in mixing traditional techniques with a declarative. There should be demonstration of what can potentially be done and why it does not provide the information to really diagnose bugs. Fairly concrete look at tefkat debugging, leaning heavily on MLs PDM paper. By the end of this chapter it should be evident that a fairly significant change in approach is required. Provides a natural progression to the forensic debugging chapter.

### 7.2 Forensic Debugging

Outline the investigation into forensic debugging work. Fairly meaty look at the debug environment, what artifacts are available, what could be available? Look at the techniques from the forensic debugging paper. Discussion on some of the information not available and some of the limitations w.r.t. space/time. Lead out that this additional information can be captured in a live debugging environment

There are a number of possible approaches to forensic debugging, however model-driven development provides a unique opportunity for forensic debugging due to the general availability of detailed, bi-directional, trace information for model transformations. This trace information can be leveraged to provide a detailed post-hoc analysis of model transformation which is not possible in more traditional software development approaches. Utilizing the problem classifications from chapter ??, a number of generic debugging questions can be answered.

### **7.2.1 Identifying Potential Causes of Problem XYZ**

– detail the progression through each of the simple problem classifications which can be answered with the current level of trace information (1 section per) –

### **7.2.2 Pinpointing The Problem**

– introduce techniques for narrowing analysis (deltas, ...) –

### **7.2.3 Initial Discussion**

– conclusion: answers simple questions, but it would nice to have some more or different trace information available for more detailed analysis and further narrowing of analysis –

### **7.2.4 Forensic Debugging with Enhanced Trace**

Now that the potential for forensic debugging in model-driven development has been established, enhancements can be made to the available trace information output from a model transformation to execute more precise debugging algorithms.

### **7.2.5 Trace Enhancements**

– Discuss changes to trace information and how it was implemented –

### **7.2.6 Utilisation in Forensic Debugging**

– Improvements to forensic debugging algorithms to utilise new trace information –

### **7.2.7 Lint**

### **7.2.8 OLD-GEAR-DUMPING-GROUND**

### **7.2.9 Localisation of Model Transformation Bugs**

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

#### **7.2.10 Analysis**

Analysis involves gathering evidence from the artifacts available in the normal model transformation environment (see figure ??). 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, programmatically 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 contributors.**

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), ( $TRM[source-references]$ ), 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.

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

#### **Source objects that did not contribute to the creation of a target object.**

Similarly, 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.

$$y = TRM[source-references] - SM$$

As discussed in section ?? 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.

```

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;

```

Figure 7.1: Simple Tefkat rule containing a bug.

### 7.2.11 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 [?, ?] and predicate switching [?] 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 7.1, shows a Tefkat transformation rule.

This rule is working with a simplified UML meta-model, figure 7.2. 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 A.1, “why are there no *UMLClasses* in my output?”. To answer this question re-enactment is used.

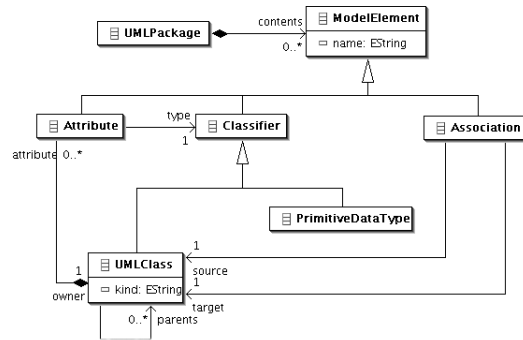


Figure 7.2: Simple UML meta-model.

### 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 ??, 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 7.3. 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.

```

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 mofInstances
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
  
```

Figure 7.3: Head-first predicate switching.

The tail-first switching algorithm is shown in figure 7.4. 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.

```

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 mofInstances
  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

```

Figure 7.4: Tail-first predicate switching.

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 ?? and section 7.2.9 this may not be the root cause of the bug, however it localises the problem sufficiently to 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<sup>2</sup>. 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 7.3 and 7.4 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 7.3 and 7.4 address a simple case where all the effecting logic is encapsulated within a single rule 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 7.5 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 7.3 or step 4.1 in figure 7.4. The first additional check is for an OR condition. If either of these statements are encountered, each of its branches

<sup>2</sup>The transformation can not complete execution as a rule is dependent on variable that is never bound.

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

Figure 7.5: Handling multiple branches.

must be traversed separately. It is possible that the conditional statement will result in 0, 1 or 2 additional results.

Other advanced constructs; IF/THEN/ELSE statements, PATTERNS 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.

## 7.3 Live Debugging

Live debugging approaches. Advantages/problems of having runtime information. Timeliness and feedback loops vs post-execution reproducibility. Execution trace, live data moves back to forensic world.

Some fairly good discussion to be had around statistics and visualisation. Expect a fairly heavy tefkat flavour, look at some of the proof of concept things. Traffic lights and other information radiators.

### 7.3.1 Term Coverage

### 7.3.2 Data Profiling

I need to work out where rule assertion stuff fits it. Originally it spun out of live debugging, but ties closely into the mental model gear.

## Chapter 8

# Observation and Verification Techniques

This chapter aims to split out the issues inherent with manual observation. This will drive out the 'lint' work and the 'rule / expectation' annotations.

## Chapter 9

# Advanced Example of Debugging Techniques

### 9.1 UML to Relational

In order to demonstrate debugging techniques, we will utilise two running examples. The examples shall also serve to introduce Tefkat's syntax. The first of these is a UML to relational database model transformation. The UML to relational transformation is often the canonical example in model-transformation literature, and should provide a familiar domain.

The problem is to take a model, conforming to the simplified UML meta-model specified in figure 9.1 and transform it into a corresponding model, conforming to the relational meta-model specified in figure 9.2.

Figure 9.3 illustrates the key relations, increasing in detail. The overarching task is the translation of a simple UML model to a relational model. The crux of this in the translation of *persistent* UMLClasses into Tables, this provides the structure of the output model where all top level elements are Tables. The output model must then be filled out with Columns. This is can be thought of as two relations, the common case, of translating class attributes to table columns, and the corner case, of translating inherited attributes to table columns.

In Tefkat, each transformation begins with a declaration of the source and target models and namespaces to pull in each of the meta-models required for the transformation.

```
1 TRANSFORMATION uml2rel : uml -> rel
2
3 NAMESPACE http://simpleuml
4 NAMESPACE http://relational
```

In this case we are declaring a transformation named `uml2rel` with an input model, `uml`, and an output model, `rel`. There are two meta-models imported, the input model identified by `http://simpleuml` and the output model identified by `http://relational`. The input names may be used later to fully qualify elements from a particular input model, but in the case of a single input model this qualification is not necessary.

Transformations are built from rules for transforming source elements into target or intermediate elements. The core or main-line rule for the UML to

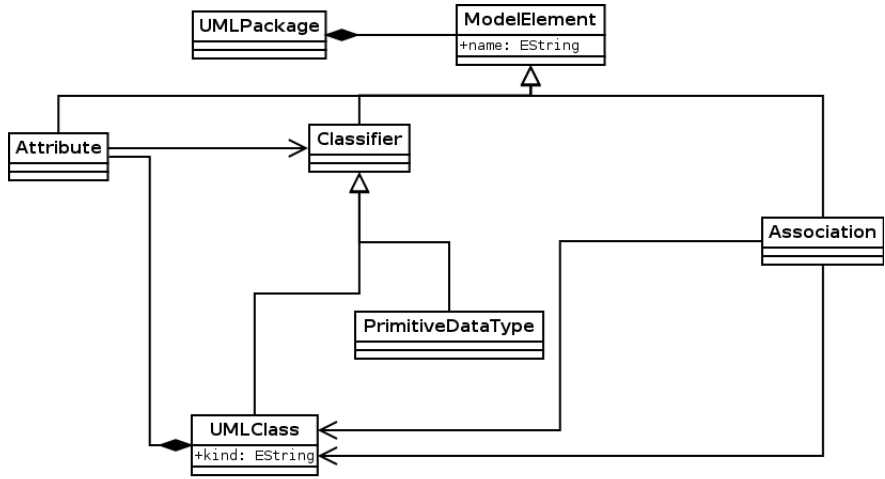


Figure 9.1: Meta-model representing our simplified UML example.

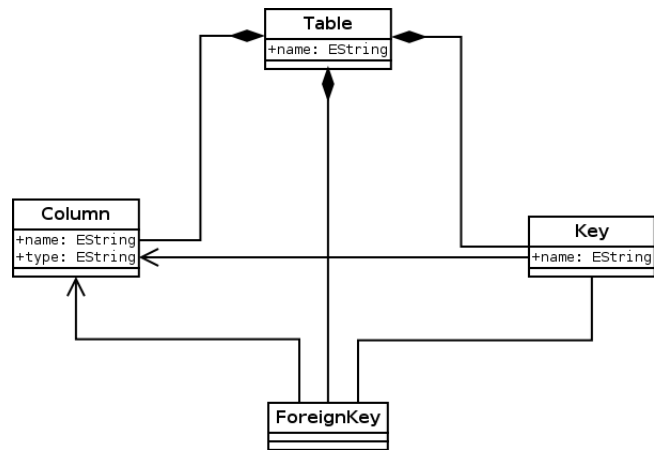


Figure 9.2: Meta-model representing our relational model example.

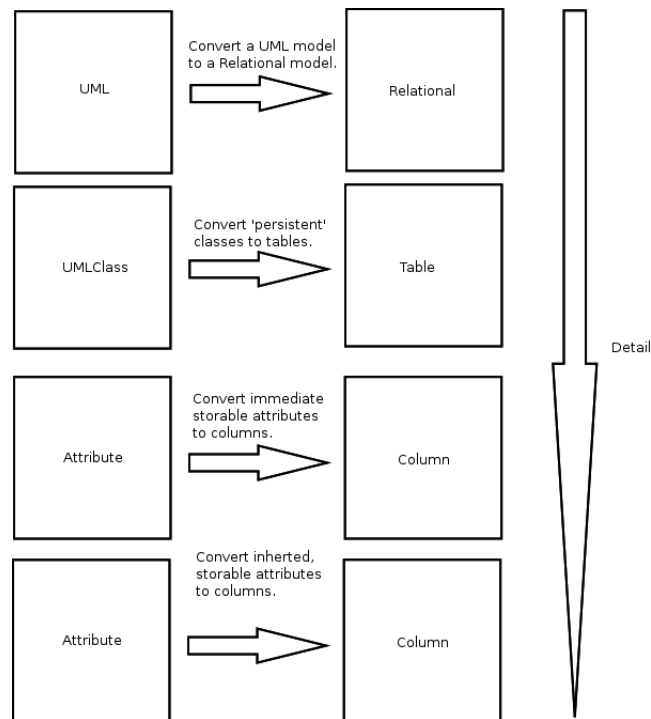


Figure 9.3: Source to target relations of increasing detail.

relational example is responsible for transforming *most* UML classes to tables in the relational model.

```

1  RULE Class2Table(c, t, k)
2    FORALL UMLClass c
3      WHERE c.kind = "persistent"
4      MAKE Table t, Key k
5      SET t.name = c.name,
6          t.key = k,
7          k.name = c.name
8  LINKING ClsToTbl
9    WITH class = c,
10         table = t
11 ;

```

This rule introduces some important concepts. The rule can be divided into two parts, input selection, output generation. Inputs are selected based upon the `FORALL` and `WHERE` clauses. The `FORALL` selects all elements of a particular type, in this case elements of the type `UMLClass` and binds the elements to the name `c`. The `WHERE` clause is a series of terms, where only input elements that can be evaluated for all terms will be selected. In this case the term `c.kind = "persistent"` acts as a predicate, much like it would in SQL, however terms in the where clause are more than simple predicates. Take for example the same `WHERE` could be re-written as:

```

1  WHERE c.kind = kind
2    AND kind = "persistent"

```

In this case there are two terms that need to evaluate for each input. This where would reduce to the same selection of inputs, but has evaluated an additional variable `kind`, that may be referenced later in the rule.

The second part of the rule, output generation, can be further divided for analysis. The `MAKE` and `SET` clauses operate on the output model. Where `MAKE` creates elements in the output model and `SET` assigns values to attributes of those new elements. The `LINKING`, `WITH` clauses creates intermediate elements, in this case `ClsToTbl`. Tefkat allows intermediate structures to be defined inside the transformation.

```

1 CLASS ClsToTbl {
2   UMLClass class;
3   Table table;
4 };

```

This class has two attributes, `class` and `table`, that link an input `UMLClass` and an output `Table`. These intermediate classes can be used as input to other rules, providing the ability to decompose the transformation into a number of smaller rules, and establish dependencies between rules that give rise to an implied ordering.

This main-line rule provides the structure of the output model in the tables, but leaves the detail of column generation to other rules.

```

1 RULE Attribute2Column(c, t, a, n, col)
2   FORALL Attribute a
3     WHERE ClsToTbl LINKS class = c, table = t
4         AND hasAttr(c, a, n)
5         AND storable(a)
6     MAKE Column col FROM c2a(c, a, n)
7     SET col.name = n,
8         t.column = col
9   ;

```

The `Attribute2Column` rule has a similar structure to `Column2Table`. The rule uses all `Attributes` as input, but also relies on a new construct `LINKS` to use the intermediate `ClsToTbl` instances created by the first rule, `Class2Table`. The `LINKS` term is expressing that for all instances of `ClsToTbl`, bind the class attribute to `c`, and the table attribute to `t`. The rule is now stating for all `Attribute` and for all `ClsToTbl`, giving us a cross product, the complete set of pairs of `Attribute` and `ClsToTbl`.

The next step is to filter these pairs to only those where the `Attribute` is owned by the corresponding `UMLClass` and then further refine to only those attributes which can be considered storable. To do this, we will introduce a new feature of Tefkat - patterns. The `Attribute2Column` rule has two pattern uses, `hasAttr(c, a, n)` and `storable(a)`.

```

1 PATTERN hasAttr(c, a, n)
2   FORALL UMLClass c, Attribute a
3     WHERE (
4       inhAttr(c, a) AND
5       a.name = n
6     )
7   OR (
8     inhAttr(c, a2)           AND

```

```

9         a2.type = c2           AND
10        UMLClass c2           AND
11        c2.kind != "persistent" AND
12        hasAttr(c2, a, n2)     AND
13        n = append(a2.name, append('_', n2))
14    )
15 ;

1 PATTERN inhAttr(c, a)
2     FORALL UMLClass c, Attribute a, UMLClass c2
3         WHERE a.owner = c2
4             OR (c.parents = c2 AND inhAttr(c2, a))
5 ;

1 PATTERN storable(A)
2     FORALL Attribute A
3         WHERE C = A.type
4             AND (PrimitiveDataType C
5                 OR (UMLClass C AND A.type.kind = 'persistent'))
6 ;

```

## 9.2 Example 2

TODO... now that the syntax should be more familiar, I want to introduce a more sophisticated example. I would expect to step through this a lot quicker, as there need not be a lot of time spent on syntax (except templates possibly), just emphasising the cognitive process. Structural rules - Detail rules, Main-line - special case rules.

## Chapter 10

# Proof of Concept Wrapup

Applying concepts in practice. Wrapup implementation contribution. Should recap forensic (rule manipulations, rule coverage, trace history/snaphots) , live (stats, radiators), lint, programatic expectations as well as interacting with developers w.r.t. debugging questions and gathering context

### Observation

**lint** have scaffolding and a fairly large set of inspection metrics, still need to define the rules on what-how metric are used. big one, plays into localisation and verification.

**historical** record, and do statistical analysis on deltas

### Symptoms and Questions

**question ui** basic ui for driving user through question, rudimentary at best, needs work

### Localisation

**term rewriting** mostly done, rule rewrite, rule selection needs a bit of work, ties in with symptom question gear

**traffic light branches** pretty colours and all that, works down to term level, some questions around highlight overlaps , mostly done, requires teekat patch

**lint** lint crosses over here, see observation.

### Expectation - Verification - Violation

**annontation** rules, scaffolding, basic external grammer for parsing rules, magnitude type rules still need to break down different types of expectations, wire all the rules up

**lint** lint crosses over here, see observation.

Notes about annotation / rules.

0. constants - $\iota$  comparisons 1. expectations about source 2. expectations about target 3. src - $\iota$  target @ transformation 4. src - $\iota$  target @ rule 5. src - $\iota$  target @ trace - $\iota$  from 6. relative to a rule 7. src - $\iota$  linkings 8. linkings - $\iota$  trg 9. patterns

– magnitude – comparators

groups of rules - $\iota$  rules that work together, similar but disjoint set of cases sequencing - $\iota$  rules following sub-types or containment structural vs detail mainline: ?? consumption vs production ?? cornercase: initial/root: mainline = ? maincount = 0 for r in rules: count = —Tables by r— if count  $\iota$  maincount: mainline = r, maincount = count

# Chapter 11

## Discussion

### 11.1 Interactions with live debugging

. Can we utilise expectations to trigger better step-through debugging. . How does thist work w.r.t. percentages. .

### 11.2 Generalisation

Move away from specific tekat discussion, and start discussing the applicability to wider audience.

#### 11.2.1 Model Transformations and Other Declarative Programming Models

Yeh...

#### 11.2.2 Imperative Programming Models

Mainly discussion on potential of applying mental models in an imperative language and some simple examples of how the concepts may map to an imperative transformation (say something written in java).

### 11.3 Future Work

[From forensic debugging paper](#)

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 ?? 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 ?? . 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.

#### [From declarative debugging paper](#)

With the developer able to verify their expectations, the next step is to capture this information for use in testing and automated bug localization. One proposed approach to be investigated in the future, is the ability to capture success rates and object creation over-time resulting in a “typical execution history” of the rule and any variations of actual success rates to expected success rates can be quickly noticed.

Another approach to be considered, is the the ability for developers to annotate their intentions for terms. Similar to determinism which can be expressed by Mercury’s type system [?], the annotations would mark execution expectations for each rule. The classes of expectations would correspond to our term highlighting techniques, where a term would be expected to consistently succeed, fail or succeed only some of the time. For example, identifying navigational terms that should always succeed, or using fuzzy logic techniques to identify terms that succeed *most* or *some* of the time. These annotations could be used by the transformation engine to verify expectations at run-time.

# Chapter 12

## Conclusion

- Directly address research questions and contribution to debugging.
- Provide nothing new - summarize/reference existing.
- Re-iterate chapter content. e.g. in chapter X we showed Y.

[from forensic debugging paper](#)

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 ??, we presented a framework, as a set of questions, to define the goals of model-transformation 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.

[From declarative debugging paper](#)

We have shown that practical high-level debugging techniques can be used with declarative transformation languages, shielding the developer from a detailed understanding of the underlying transformation engine's execution strategies.

We demonstrated new debugging techniques that exploit data and execution trace to visualize the logical transformation execution in a “debugging question” dependent manner. Allowing the debugging context to be driven through well

understood domain-specific debugging questions, our approaches were able to deal with the volumes of data that can be produced by a model transformation and present relevant execution summaries in a manner easily understood by developers. Using the visualised execution summary, developers are able to verify, from a high-level, that the intent of the transformation and any underlying assumptions were met.

## Chapter 13

# Dumping ground...

### 13.1 confirmation



Faculty of Information Technology  
Queensland University of Technology

# Confirmation and Articulation Report

December, 2007

## Debugging in Model-Driven Engineering

Mark T. Hibberd  
mt.hibberd@student.qut.edu.au  
04375319

Supervisors:  
Professor Kerry Raymond  
k.raymond@qut.edu.au

Dr Michael Lawley  
m.lawley@qut.edu.au

# Contents

<b>1</b>	<b>Research Outline</b>	<b>3</b>
1.1	Research Problem . . . . .	3
1.2	Research Questions . . . . .	4
1.2.1	Research Question 1 . . . . .	4
1.2.2	Research Question 2 . . . . .	4
1.3	Significance of Research . . . . .	5
1.4	Limitations of Research . . . . .	5
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Model-Driven Engineering . . . . .	7
2.1.1	Continuing The Abstraction Pattern . . . . .	7
2.1.2	Formalising MDE, The Concepts . . . . .	8
2.1.3	MDE in Practice . . . . .	8
2.1.4	Modeling Languages and Meta-Modeling . . . . .	10
2.1.5	Model-Transformations . . . . .	11
2.1.6	Traceability . . . . .	14
2.1.7	Tefkat Perspective: The Model-Transformation Environment . . . . .	15
2.2	Software Debugging . . . . .	17
2.2.1	What Is A Bug? . . . . .	17
2.2.2	Software Verification or Bug Identification . . . . .	17
2.2.3	Bug Localisation . . . . .	18
2.2.4	Bug Correction . . . . .	20
2.3	Discussion . . . . .	20
<b>3</b>	<b>Research Methodology and Design</b>	<b>21</b>
3.1	Research Approach . . . . .	21
3.2	Implementation . . . . .	21
3.2.1	Iteration 1: Problem Definition . . . . .	21
3.2.2	Iteration 2: Defining the Debugging Questions . . . . .	21
3.2.3	Iteration 3: Forensic Debugging . . . . .	22
3.2.4	Iteration 4: Trace Enhancements . . . . .	22
3.2.5	Iteration 5: Forensic Debugging with Extended Trace . . . . .	23
3.3	Outcomes . . . . .	23
3.4	Prototype . . . . .	25
3.4.1	Underlying Technology . . . . .	25
3.4.2	Functionality . . . . .	25
3.4.3	Interfaces . . . . .	25
3.5	Scholarly Activities . . . . .	26
3.6	Research Time-line . . . . .	26
3.7	Risk Management . . . . .	26
3.8	Budget . . . . .	28
3.8.1	Hardware/Software . . . . .	28
3.8.2	Travel . . . . .	28
3.9	Ethical Clearance . . . . .	28

<b>4</b>	<b>Work to Date</b>	<b>29</b>
4.1	Iteration 1: Problem Definition . . . . .	29
4.2	Iteration 2: Defining the Debugging Questions . . . . .	29
4.3	Iteration 3: Forensic Debugging . . . . .	29
4.4	Iteration 4 and Beyond . . . . .	30
<b>5</b>	<b>Articulation</b>	<b>31</b>
5.1	Extended Research Problem . . . . .	31
5.2	Extended Research Questions . . . . .	31
5.2.1	Research Question 3 . . . . .	31
5.2.2	Research Question 4 . . . . .	32
5.3	Revised Research Plan . . . . .	32
5.3.1	Iteration 6: Addressing Debugging Questions in a Live Environment . . . . .	32
5.3.2	Iteration 7: Alternate Live Debugging Paradigms . . . . .	33
5.3.3	Iteration 8: Model-Transformation Testing and Constraints . . . . .	33
5.3.4	Revised Time-line . . . . .	34
	<b>Appendices</b>	<b>42</b>
<b>A</b>	<b>Published Work</b>	<b>42</b>

## List of Figures

1	Generic MDA pattern. . . . .	8
2	The model-transformation environment. . . . .	16
3	Model trace environment. . . . .	16
4	Before and after localisation of potential problems. . . . .	18
5	Time-line for completion of Masters candidature. . . . .	27
6	Revised time-line for completion of PhD candidature. . . . .	35

## Abstract

In software, models are abstractions used to define a system in a way that aims to be easier to understand, communicate and reason about. Model-driven engineering (MDE) utilises models as first-class artifacts in the software development process. Developers are able to define applications using modeling languages, both graphical and textual, that are more expressive and better at capturing domain knowledge without having to deal with technical concerns.

MDE has a wide range of potential benefits, which, depending on the mode of MDE used, can lead to higher quality, productivity, maintainability or some combination of these key measures of software development. In an *ideal* world the benefits of MDE over traditional code-centric approaches would be clear and easily measurable. However, the world of software development is far from perfect; people make mistakes; bugs happen.

Mature software development languages realise and embrace this fact. Testing and debugging support are integral features for any modern language. Without this support, the barrier to adoption and effective use of a new language would be far too high. MDE is faced with this same challenge. Its adoption and effectiveness is currently limited by the unresolved issues involved with handling the imperfect nature of software development - that is, bugs. How to tell if there is a bug? How to find the bug? And, how to fix the bug? If these questions can be answered effectively, MDE will be significantly closer to realising its full potential.

This research investigates these questions in more detail, drawing upon traditional debugging techniques in order to introduce new debugging approaches specific to, and optimised for, MDE. The aim of this research is to **define** the debugging issues that MDE faces, **analyse** current MDE practices in order to identify improvements to debugging support and **develop** pragmatic, generalized solutions for debugging in MDE.

# 1 Research Outline

## 1.1 Research Problem

Model-driven engineering (MDE) holds the promise of an improved level of quality, maintainability and productivity in software development. Given this, it is no surprise that MDE has been a highly researched topic and is now having a significant impact on commercial development practices [1]. Technology leaders are incorporating MDE techniques into their development tools and languages, with UML and code generation compulsory features for any competitive integrated software development environment (IDE). Business is adopting MDE as a way to capture the business knowledge they have always had, in formats which are more understandable, transferable and contribute to more productive and correct development of software. Independent software vendors are turning to MDE and templating techniques as a way to improve rapid application development and support ever-changing technological needs.

This research and adoption has been heavily weighted towards the productivity aspect of MDE, such as automation and code generation. Whilst productivity is extremely important, MDE's benefits are now being limited by its quality characteristics, such as correctness and maintainability [2]. This has been demonstrated by a shift in research priorities of recent times towards making MDE a more complete and pragmatic development technique [3]. Examples of this include, model comparison [4], model merging [5], model-transformation validation [6, 7, 8] and model checking [9, 10, 11]. Current quality-related research concentrates on preventing or at least detecting problems in the MDE process in attempt to continue forward momentum. The research does not extend to case where something has already gone wrong and a recovery process is required. What can be done when bug occurs in MDE?

MDE encompasses a wide range of technologies and processes. Many of which are shared with more traditional forms of software development. In these cases, the answer to the proposed question is straight forward. Experience generally shows the types of bugs which occur and the debugging techniques which can be applied to solve these problems. However, there are several areas of MDE which do not have this experience or immediacy of solutions. One example, and an integral part of MDE, is model-transformations, in particular declarative model-transformations, which present a set of debugging challenges where the set of possible bugs is yet to be fully explored and importantly, there is little rigour or support in its debugging approaches.

Declarative model-transformations share some of their debugging problems, such as lack of a well defined execution order, with other declarative programming approaches like logic languages (e.g. Prolog). However, declarative model-transformations have some interesting differences to other declarative mechanisms. The most important of these is the concept of traceability as a first-class participant in model-transformations. Traceability could potentially provide the extra level of information required to produce more powerful, model-transformation specific, debugging approaches.

The characteristics of this research problem, debugging of declarative model-transformations with strong trace information, help to define the requirements for the MDE tools and approaches required to investigate this problem further. Using these characteristics as a guide, this research shall use the Eclipse Mod-

elings Framework (EMF) [12] and the Tefkat [13] model-transformation engine. Tefkat uses a declarative approach to model-transformations. It has a formal trace model, which links the target, source and transformation. These tools provide a strong match for the research problem, and as they are already used for MDE, they allow for effective evaluation of this research on existing MDE artifacts.

By combining research from the MDE community with traditional and declarative debugging techniques this research aims to introduce post-hoc, or forensic, debugging approaches to model-transformations. Forensic debugging techniques treat the run-time environment as a black-box, relying on external artifacts, compared with live debugging approaches which access the internals of the run-time environment. This research aims to make contributions to the overall MDE approach, by providing well-structured processes and tools for debugging. These processes and tools will allow the growing number of MDE practitioners to develop and support software using a model-driven approach in an easier and more efficient manner. These goals are captured by the research questions in the following section.

## 1.2 Research Questions

Two overarching research questions have been identified:

### 1.2.1 Research Question 1

*What are the types, qualities and relationships of bugs that occur in model-transformations?*

This question can be further broken down, into:

- *What are the informal debugging questions asked by transformation developers?*
- *How do these debugging questions relate to model-transformation bugs?*
- *What are the defining attributes and classes of model transformation bugs?*

### 1.2.2 Research Question 2

*To what extent can the trace information created during model transformations be leveraged for forensic debugging to address the debugging questions, and building on this evaluation can the available trace information be improved to better support forensic debugging?*

This question can be further broken down, into:

- *What forensic debugging techniques can be applied to the model transformation data?*
- *Can the model-transformation data be enhanced to better support the forensic debugging techniques?*
- *To what extent can these techniques be automated?*
- *To what extent can these techniques improve developer productivity?*

### 1.3 Significance of Research

This research problem represents a largely unexplored area of MDE which is critical to the continuing evolution and further research of MDE processes and techniques. Research has shown that debugging of model-transformations is difficult [13, 2]. There are no well defined or understood techniques that allow for model transformation debugging to be as effective as traditional approaches such as source-level step-through debugging and program-tracing [14] for imperative languages.

Although model-transformations draw on many other languages and programming paradigms there are a number of key differentiators over traditional software debugging.

Firstly, the formal trace information produced by model-transformations provides a multi-directional link between the source and target as well as the actual transformation or compilation step which is responsible for the relationship. The use of this trace information for debugging presents a unique research opportunity that could have a significant impact on the efficiency and effectiveness of debugging in MDE. As a result, an overall improvement of quality and productivity in software development can be achieved.

Another difference between traditional software language debugging techniques and the proposed research is the use of declarative languages for specifying model-transformations. Declarative languages provide many additional complexities to imperative languages with respect to debugging, particularly concerning execution order. Although there have been research into debugging techniques, such as program slicing [15, 16] and algorithm debugging [17, 18, 19], of traditional declarative logic languages like Prolog, there has been limited practical application (none specific to model-transformations). This gap in traditional debugging techniques provides further opportunity to make a significant research contribution.

The practical nature of the research outcomes will also assist in improving the pragmatic application of MDE in software development today. The outcomes of the research aim to make an improvement on the current MDE best practices, by delivering improved productivity in development of transformations, improved maintainability of transformations over the full software development life-cycle and providing a contribution to the overall quality of software developed utilising MDE. Specifically, providing debugging techniques for use with Tefkat will assist with other MDE research that utilises the transformation engine.

The final important contribution which this research aims to make is the analysis of the current level of trace information provided by the transformation process. The identification of redundant or missing trace information will allow future MDE practices and standards to cater for both forensic and live debugging more effectively.

### 1.4 Limitations of Research

The current scope of this research is limited to the application of forensic debugging techniques for the bug-localisation phase of debugging. The scope, in the context of achieving Masters candidature, does not include live or interactive debugging techniques. The scope also limits the research into the bug-identification or bug-correction phase of debugging choosing to concentrate

on the most difficult phase of debugging, localisation.

Traditionally debugging techniques are often centered around reducing the scope of the problem. This can be achieved by data slicing [20], a reduction of scope through a smaller input set, and program slicing [21], a reduction of scope through the execution of a smaller part of the program. In the case of model-transformations, specifically using declarative languages, a small input set can produce a large output set, and potentially require a significant amount of resources, i.e. memory and time, which reduces the effectiveness of data slicing techniques. Where as program slicing techniques are more effective with declarative language, as they are more powerful, i.e. each term carries more meaning or does more work, than their imperative alternatives and as a result any simplification (slice) of a program can provide a significant reduction in complexity and effort required to debug. As a result, this research is focused only on the program slicing case.

Whilst this research aims to be as generic as possible across all variations of model-driven engineering, there will be limitations in the application of the practical outcomes. The debugging processes shall only be applicable to declarative transformation approaches such as those provided by Tefkat and specified by the MOF QVT [22] standard as the QVT relations and core languages.

Another limitation will be to the actual implementation of the debugging framework. The framework is intended to be specific to Tefkat and EMF. This does not preclude the concepts from being used with other specific tools, but does mean further development work would be required for their support.

The limitations of this research have been carefully considered to maintain a high level of relevance to modern software and model-driven engineering. The choice of utilising a declarative transformation language (Tefkat) for the proof of concept is supported by the adoption of the OMG's QVT standard [22], specifically the relations language whose features and advantages loosely correlate to those provided by Tefkat. This direction is further supported by the adoption of Object Constraint Language (OCL) [23] in modeling, which approaches the application of constraints to models through declarative statements.

## 2 Background

Model-driven engineering (MDE) has been a highly researched area over the past several years. This research has resulted in continued improvement and increased adoption of MDE in commercial environments. However, as yet, there has been little or no research into the problems modelers face when MDE does not work as planned, that is, a bug occurs. To better understand this problem, this section surveys literature from both the MDE and software debugging fields.

The first section, Section 2.1, discusses the foundations of MDE, including the technology and techniques that contribute to it. To build on the initial background and core concepts, particular emphasis will be placed on the aspects of model-transformations and traceability in MDE. The purpose of this section shall be to provide a rigorous understanding of MDE and ensure that any MDE debugging techniques presented are rooted in a relevant base as well as being positioned for practical use.

The second section, Section 2.2, introduces debugging concepts, including the debugging phases and the different approaches to debugging, as well as outlining the goals of debugging software. These concepts shall be used to discuss the parallels and potential reuse of research in debugging of declarative programming languages. The purpose of this section is to elaborate on the goals of software debugging, and introduce concepts that can be adapted to support debugging in model-driven engineering.

### 2.1 Model-Driven Engineering

Model-driven engineering, along with other similar terms such as model-driven software engineering (MDSE), model-driven development (MDD) and model-driven software engineering (MDSD) describes the broad approach to the use of models as first class artifacts in the software development lifecycle. MDE encompasses a combination of technologies, processes and approaches. Its uses are varied, with each specific use or mode of MDE having different advantages and limitations. This section refines this definition, introducing the individual concepts and technologies which combine to enable MDE.

#### 2.1.1 Continuing The Abstraction Pattern

From the earliest days of software, abstraction has been central to development improvements; MDE continues this theme. MDE abstracts away technology concerns, allowing developers to concentrate on the problem at hand. Parallels can be drawn between third-generation languages, such as C and Java [24], and MDE. In third-generation languages the abstraction or model is represented by the human readable source code, which is then compiled or transformed for its target environment, machine code in the case of C, or byte code in the case of Java. This comparison between third-generation languages and MDE is important to demonstrate that important lessons learned for one approach can be applied to the other and, in the case of this research, taking debugging lessons learned from traditional programming languages to achieve optimal debugging process for MDE.

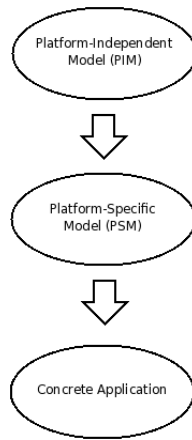


Figure 1: Generic MDA pattern.

### 2.1.2 Formalising MDE, The Concepts

The Object Management Group (OMG) have attempted to formalise what model-driven techniques encompass by defining the Model-Driven Architecture (MDA) [25, 26]. The MDA Guide [27] released by the OMG attempted to define the transformation concepts the OMG saw as critical to the success of MDE. The basic MDA pattern (Figure 1), involves three key steps [25, 28]:

1. Construction of platform-independent model(s) (PIM) describing the application without concern for target platform characteristics.
2. Translation of the PIM into a platform-specific model (PSM) which has been optimised or enhanced to support target platform characteristics. This translation can either be manual or automated through a transformation language (Section 2.1.5).
3. Translation of the PSM into application artifacts, i.e. code, configuration files and deployment descriptors. Again, this translation can either be manual or automated through a transformation language (Section 2.1.5) or through the use of templating.

A less formal approach to MDE is discussed by Bock's paper *UML without pictures* [29]. The paper explores the non-graphical side of models and how utilizing the non-graphical side can increase the consistency and level of detail in models to the point where implementations can be completely generated.

Discussion of the concepts at this high level is important, however the reality of MDE is more diverse, with several specific modes of MDE in use, some of which have been a part of software development much longer than the more formal definitions of MDE.

### 2.1.3 MDE in Practice

Since the publication of the MDA process, MDE research and experience has progressed significantly.

There are now several clear *modes* of MDE emerging in different communities:

**Parameterised Code Generation:** Code generation techniques have been utilized since the earliest days of computer programming [30]. Models allow for effective definition of the specific details (i.e. the parameters) without concern for the generation technique. Utilising templating or other generation techniques, the models can be transformed into common patterns. This technique is extremely effective in large systems where entities all require similar functionality [31]. An example of this would be a system that has a large number of business entities that all follow a CRUD (Create-Read-Update-Delete) pattern. The pattern is defined once, i.e. the transformation. Then each details of each entity are defined, i.e. the model.

**Incremental Transformation:** The use of incremental transformations in MDE closely parallel what can be termed the pure MDA pattern (Figure 1). The system is modeled in a generic manner, then transformed into more specific model(s). This transformation process is repeated, until the final system components are defined. The incremental transformation process is powerful as each transformation can be quite specific concentrating on a single step in the process, without the complexities of a monolithic, do everything at once transformation.

**Super Compiler:** Whilst most other MDE techniques concentrate on the productivity and flexibility aspects of model-driven process, the super compiler paradigm targets quality as its key goal. By combining extremely detailed transformations with model checking techniques [9], systems can be defined entirely as a set of models before being transformed to the target language. The key aspect of this approach is the advanced static analysis and constraint checking that can be built into the transformations and model checkers to pick up a large number of critical problems before the code can even be generated. This process of quality checking as the code is generated parallels the role of a code compiler, particularly in statically typed languages. Where a compiler picks up syntactic and typing errors, the code generation process can be used to detect logic errors, un-handled inputs and missing exception handling. This mode of MDE is often used in the embedded device and hardware fields, specifically in model-integrated computing [32, 33], where quality is all important. Weil et al. have demonstrated this approach to MDE in a commercial setting [34].

**Software Modernization / Reverse Engineering:** There is a significant amount of legacy code still being supported; this code is often poorly documented and under supported. When software evolution is required, changing or rewriting this code is often associated with high risks and costs. MDE techniques are being utilized to extract models of code that can be used to partially generate or at the very minimum verify behaviour of new updated code bases in modern development languages. The OMG have highlighted this style of MDE as Architecture-Driven Modernization [35]. Fleurey et al. [36] and MacDonald et al. [37] have demonstrated an MDE modernization process in a commercial environment.

**Model Composition:** To enable highly flexible systems, it is often desirable or required to have multiple models to describe a system. An alternative to performing incremental transformation on multiple inputs, is to compose these models into a single system. The reasons for doing this are varied, however there are two areas of research which deal with this model composition approach in more detail. Aspect-Oriented Modeling (AOM) [38] and Software Product Lines (SPL) [39].

#### 2.1.4 Modeling Languages and Meta-Modeling

Modeling languages are core to model-driven approaches. They provide a mechanism for defining model instances. Modeling languages have varied syntaxes, from textual, such as Tefkat’s model-transformation language [13], to graphical, such as UML’s graphical syntax [40, 41]. This differs from traditional programming languages, which, despite stylistic and structural differences, are fairly consistent in that their concrete syntax is almost always textual, comprising some basic combination of keywords and expressions. Another key difference of MDE over traditional approaches is the embracing of domain-specific languages [42]. DSLs are languages designed to be good at addressing specific problems or a single *domain* [43], compared with general-purpose languages which aim to solve a wide range of problems across multiple domains. As DSLs are targeted at a small problem set, they are often more concise and effective than their general-purpose counter-parts. However, this advantage is often negated by the fact that each new problem requires a new language, and with it, a new set of skills and tools.

To address this issue, MDE uses meta-modeling [44] to describe and define modeling languages. The meta Within MDA, OMG use their Meta-Object Facility (MOF), current available specification version 2.0 [45], to define meta-models. MOF is specific to the domain of meta-modeling, making it a DSL itself and importantly MOF is reflective, meaning that it can define its own meta-model.

MOF defines a layered stack of models, each layer containing a different level of meta-data, M0 for the concrete syntax through to M3 for MOF itself. The MOF model stack is made up of:

- The MOF, which is a meta-model used to describe the specific domain of meta-models, effectively making MOF the meta-meta-model [44]. MOF is self describing, i.e. there is a MOF model which describes the meta-meta-model itself, which ensures there is a finite number of meta-levels. This is known as the M3 layer in MOF.
- A meta-model describing all possible model instances, i.e. defining the modeling language. This is normally a domain-specific metamodel, however there are general purpose modeling languages, the most common of those being the Unified-Modeling Language (UML) [40, 41]. This is known as the M2 layer.
- A model instance, containing specific details of a particular case, i.e. using the abstract syntax of the modeling language. This is known as the M1 layer.

- A concrete realisation of the model instance, i.e. using the concrete syntax of the modeling language. This is known as the M0 layer.

The concrete syntax of modeling languages is also an import feature in enabling effective use of DSLs. Commonly the concrete syntax of modeling languages are graphical, allowing domain expert rather than just engineers to use the language. It is important to note that the concrete syntax is fluid from one modeling language to the next, and is not always graphical. An example of this is the Tefkat [13] transformation language syntax, which is discussed further in Section 2.1.5.

So far modeling languages have been discussed solely in the context of DSLs, however it is possible to define general-purpose modeling languages. The most common and well known of this is the Unified-Modeling Language (UML) [40, 41]. The Unified-Modeling Language (UML) is a general purpose modeling language. The version current specification is UML 2.0. It is widely used with software development to specify object-oriented designs, dynamic system behaviors as well as functional processes. A key to the adoption of UML has been the standardised graphical notation which makes all UML diagrams, regardless of purpose, similar to draw and understand. UML can be extended through the use profiles, effectively creating a UML based DSL.

One of the consequences of utilizing DSLs is inconsistent or lacking tooling support. To help combat this MOF also defines a common mechanism for representing and persisting MOF compliant models in XML, the XML Metadata Interchange (XMI) specification. The current XMI specification is version 2.1 [46] and relates to version 2.0 of MOF.

Simply put, modeling languages allow you to define models. There is no restrictions to the types of models which can be defined. Using MDA terminology, the PIM, PSM and even the final artifacts, which can be thought of as models described by their own syntax or meta-model, be it XML, Java code or alike, can be thought of as models to be defined by a modeling language. So now that models can be defined, the critical step in the MDE process is to transform these models, effectively converting one modeling language to another.

### 2.1.5 Model-Transformations

Model-transformation is the process of taking a set of one or more models and transforming it into another set of models of a different type. 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. Model-transformations can occur on the same or different levels of abstraction, i.e. converting one platform-independent form into another or converting a platform-independent form into a platform-specific form.

Carrying on the analogy of third-generation languages from Section 2.1.1, transformations can be compared to the compilation stage. The key difference in MDE being that transformations will not always be from a less specific to a more specific form. This is important to the ability to visualise many views of a particular model and more importantly it is the key to some MDE approaches, specifically ADM [35] and Model Round-Trip Engineering [47] where transformations from code to PSM or PSM to PIM are essential. Transformations also

provide a mechanism for higher level re-use of architectural, design and performance patterns. Where models can be specified without concern for lower-level optimisations that can be incorporated into lower-level transformations, again similar to third-generation languages which perform optimisation tasks during compilation.

Model-transformations are defined using model-transformation languages (MTLs). There are a number of MTLs which utilise different techniques to solve the transformation problem. The defining characteristics [48, 49, 50] of these techniques include:

- the level of automation; applied programatically or manually.
- 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; described by a meta-model or free text.
- multiplicity 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.

Only a limited set of the feature matrix is of interest to this research. Specifically:

- automated transformations.
- transformations specified as a domain-specific language.
- transformations using a declarative approach.
- model-to-model transformations, where each model is described by a formal meta-model.
- transformations involving any number of models, i.e. many to many transformations.
- transformations that trace between source and target meta models.

Automation of model-transformations is the key to effective MDE, as such, it is assumed that only those transformations which achieve a level of automation are considered by this research. Non-automated techniques are not practical in MDE and do not offer any debugging problems due to the human centric nature of manual transformations.

It is given that general-purpose languages have their own debugging processes and tools, as such this research is only concerned with the context of domain-specific languages. The domain being model-transformations. As yet, there are no debugging process specific to and optimised for MTLs. Although some debugging paradigms have been mapped to some model-transformation tools, they are often ill-suited and do not take advantage of artifacts unique to the MDE space.

The next division of model-transformation categories is that of declarative and imperative approaches [51, 52, 53]. Again, this research aims to concentrate on just one aspect of the model-transformation approach, in this case, declarative transformations. Imperative approaches to model-transformations do not present a debugging challenge as it is a well understood problem. There are a number of imperative debugging techniques that are transferable from language to language such as program tracing [14] and step-through debugging. This is in contrast to declarative approaches, where even after significant research (see Section 2.2) there are few accepted debugging practices for their unique challenges, in particular independence of execution order. Another key driver for concentrating on declarative languages is their suitability for the model-transformation problem, as shown by Gerber et al [52]. By specifying what, not how, declarative approaches provide the expressive power required to define relationships between models without the concerns, such as model traversal, of imperative approaches.

Model-transformations can be broken into two groups, model-to-code transformations and model-to-model transformations [48, 49]. Model-to-code transformations are really just a specialization of model-to-model transformations, however they're clearly divided in practice, as each case lends itself to particular transformation techniques. This research is focused on the case of model-to-model transformations. The model-to-code transformation process is generally achieved by templating techniques, this approach means that debugging model-to-code transformations is normally a case of debugging the generated code which can be achieved through the target languages normal debugging processes and tools. The model-to-model case is an important issue in MDE. There has been little research specific to this area and it presents some unique challenges and opportunities in the debugging space.

The number of models involved in a transformation should not be a great concern. This research places no restriction on the number of source or target models in a model-transformation.

Traceability is a core concept in MDE, see Section 2.1.6. The trace information produced by model-transformations is a key point of difference between declarative MTLs and other declarative languages. This research intends to leverage the trace information for more effective debugging with a higher level of automation then would be possible without the trace information.

Following on from MDA and MOF, the OMG also defines a standard for MTLs, MOF Query / View / Transformations (QVT) [22]. MOF QVT is a specification for dealing with interoperability of transformation languages and tools. QVT is divided into three parts, QVT Core, QVT Relations and QVT Operational where:

- QVT Core is the minimalistic base model for pattern matching and object creation.
- QVT Relations is a declarative mechanism for specifying relationships between two or more models. QVT Relational is effectively an abstraction of QVT Core.
- QVT Operational is a mechanism for specifying imperative implementations for transformations. QVT Operational is included for situations

where it is too difficult to specify declarative relationships in a clear, concise and efficient manner.

The current version of MOF QVT [22] relates to MOF 2.0 [45] and UML 2.0 [40, 41]. Even though MOF QVT specifies a concrete syntax, it currently lacks backing of a real implementation. However, the concepts, particularly those from QVT Core and Relations map to MTL tool implementations which have their own syntax.

As a concrete example of this, Tefkat is presented as an implementation of a declarative language for model-to-model transformations in the paper *Practical Declarative Model Transformations with Tefkat* [13]. The Tefkat engine is based on a MOF specified abstract syntax and is build upon the Eclipse Modeling Framework (EMF) [12].

Tefkat maps closely to the QVT relations language, and provides a record of trace that is sufficiently detailed to allow Tefkat to be used in further experimentation of verification and debugging techniques. In Tefkat the record of transformation is provided through a dedicated tracking model. This tracking model contains elements for each of the target elements, with links to the source element(s) and rule(s) which were responsible for its creation.

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.

### 2.1.6 Traceability

Section 2.1.5 highlights traceability as a key feature for model-transformation techniques; so what does traceability mean in the context of MDE? Continuing the third-generation language analogy (see 2.1.1), it can be demonstrated that important debugging meta-data can be lost when transforming from a high-level representation to lower-level representation. When a Java application identifies a problem, be it through a test, byte-code verification or some program invariant or condition an exception or error can be created to provide feedback. When source is compiled into pure byte code instructions, the programmer would have little or no way of tracing the cause of the exception back to a line of code. With this in mind Java byte code is able to be marked at compilation time to include links to specific source code lines or at least method invocations.

Following similar principles the key to debugging of any MDE transformation is the ability to trace model elements in both directions from generated target elements back to their source elements and from source elements to their resultant target. This concept was presented in early MDA papers [25] and consolidated in the MDA Guide [27] as a *Record of Transaction* and by the DSTC et al QVT submission [54] as a core concept in model-transformations. As highlighted by Aizenbud-Reshef et al. [55] it is also important for trace information

to provide meaningful context. The MDA Guide [27] takes this further by stating the importance of providing trace information back to the specific mappings which were used in the transformation as well as the source and target elements. An important note to the progression of traceability in model-transformations is that the requirements have been largely driven through the need for incremental update, that is transform only what has not been transformed before [22].

Although trace information is regarded as an important aspect of model-transformations and trace concepts are implemented in a majority of model-transformation tools, there is no formal definition of what details should be available and what those details would be used for. Each tool implementation provides its own perspective. Details of the Tefkat trace information are provided in Section 2.1.7.

Similar to traditional languages, traceability forms the cornerstone of debugging techniques for MDE. Traceability has been incorporated from the earliest MDA research and as such provides a solid concept on which to build debugging techniques from. Gerber et al. [52] even noted the possibility of utilising trace information for tasks such as debugging, change propagation and round-trip engineering. As discussed further in Section 2.2.3, this trace information is yet to be fully exploited for debugging. Traceability is important as a stand alone component of model-transformation, but the entire transformation environment must be examined to completely understand its relationships.

### 2.1.7 Tefkat Perspective: 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 2) 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 3).

The information contained in the trace extent can be leveraged for more effective post-hoc debugging techniques than is possible with traditional languages. Figure 3 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.

Given this environment and the aspects which contribute to MDE it is easy to see its potential benefits. However, the discussion to this point has been

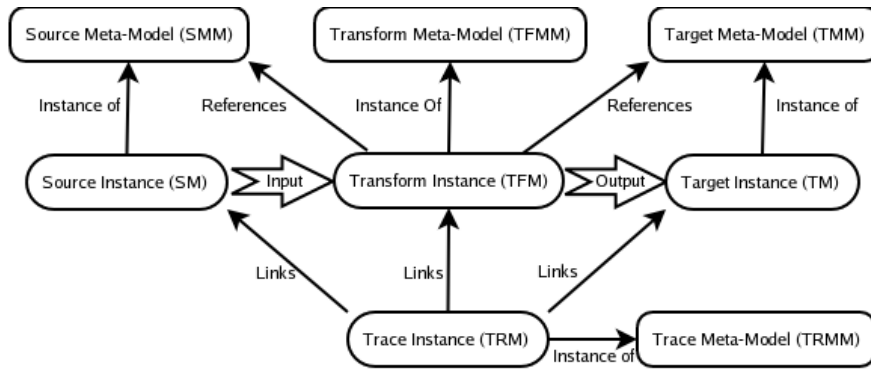


Figure 2: The model-transformation environment.

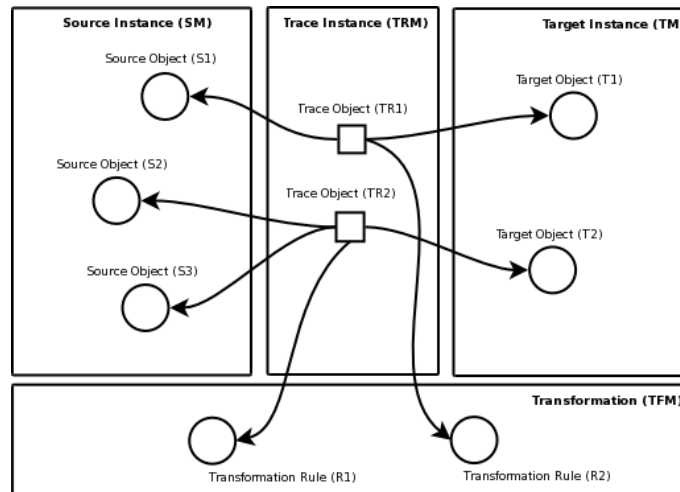


Figure 3: Model trace environment.

primarily concerned with an ideal world. The next question becomes, what happens when something goes wrong?

## 2.2 Software Debugging

Debugging is readily classified into three parts, identifying the existence of a problem, fault localisation and the actual correction of the problem [56]. 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 [57, 58, 56] 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.

### 2.2.1 What Is A Bug?

A bug, in software parlance, is a problem or defect in the design or implementation of a piece of software. Bugs can be identified in two ways, either through a software verification technique (see 2.2.2) or through failure, often severe failure, in use or *production*. The first method is preferred as it saves time, money and embarrassment for the software development team and, most importantly, can provide better input into the debugging process.

### 2.2.2 Software Verification or Bug Identification

Software verification is the process to ensure that the behaviour or contents of a system match what is expected [59]. Verification is utilised to increase the quality and reliability of systems by measuring the correctness of software with respect to its requirements [60]. Verification should not be mistaken with validation which is the process of measuring the correctness of the requirements with respect to the actual needs of the user.

The result of software verification should involve the identification of *problems* in a system [61]. Different software verifications techniques will result in various levels of detail, but for each problem any technique should be able to describe a constraint that has been violated and how that violation manifested itself, i.e. what are the externally visible symptoms of the constraint violation. Generally, the output of the software verification stage, bug identification, is the input to the software debugging stage, bug localisation, where the cause of a *problem* is located and corrected. An important point to make about software verification is that due to the complexities of software requirements and implementations, any verification technique will only be able to confirm the **presence** of a *problem* and will not be able to confirm the **absence** of *problems*. However, with the adoption of more stringent verification mechanisms and the refinement of software engineering processes, the gap between the problems which are identified and those which aren't can be significantly reduced.

Software verification techniques in MDE context are not as well defined as those for traditional code-oriented approaches. However, there are a number of techniques for model verification under research [62] and in use [34]. The majority of these techniques are concerned with the *correctness* of model instances, that is, verifying that a model instance conforms to its corresponding

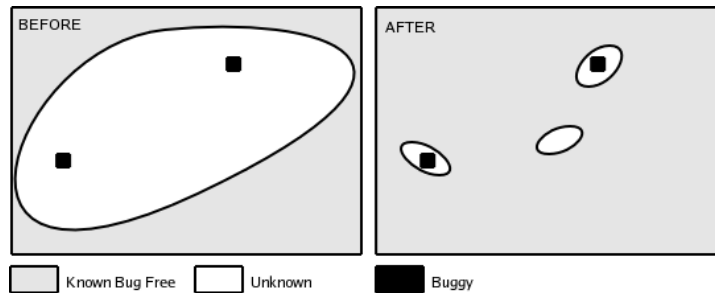


Figure 4: Before and after localisation of potential problems.

meta-models relationships and constraints. Meta-model constraints are normally defined through OCL [23].

There are fewer verification techniques for the *correctness* of model transformations. Of the model-transformation verification techniques that have been presented [7, 8, 63, 64, 65], the general approach is to re-state the source to target relationships in a more formal manner. One reason for this is that the relationships defined in a declarative transformation provide the most concise definition. As a result, the verification mechanisms are often more complicated and just as error prone as the original transformation. Additionally a significant portion of model-transformation verification research is concentrated on techniques for graph transformations and have limited applicability to other types of transformation approaches. Varró and Pataricza [63, 64] have presented formal approaches to model verification using state and transition based model checking for graph transformations. A similar approach has been adopted by Zhao et al [65] using model-transformations to petri-nets for analysis.

Assuming that a bug has been identified by a verification mechanism, automated or manual, it must then be localised.

### 2.2.3 Bug Localisation

Where verification is the identification of a bug, debugging is the process by which the source of that bug is located and corrected. Debugging is an integral part of any software development process. Almost all modern languages are designed with debugging mechanisms and tools in mind. The ease by which problems can be corrected in any technology will directly contribute to its success or failure.

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

Debugging is a well understood problem for traditional, imperative program-

ming languages. There are two accepted processes, program tracing [14] and interactive debugging [66]. Tracing involves observing the execution path of a program. This can be done in a number of ways, the most primitive being insertion of *print* statements in the application or more advanced techniques involve the use of dynamic debugging tools such as DTrace [67]. Interactive debugging involves the developer interacting with the program execution directly, through break-points and watches. Both of these techniques rely on a clearly defined execution order and a close mapping between the source and executable code. Neither of these are true of declarative languages. Lawley and Steel [13] have demonstrated that although interactive techniques can be applied to model-transformations, their declarative nature makes this a sub-optimal solution. To address this concern they highlight related work in debugging of declarative logic languages.

Logic languages, such as Prolog [68] and Mercury [69], share their declarative nature, and hence their debugging problems, with declarative model-transformations languages. There has been a wide range of research into debugging of logic programs, including algorithm debugging [17, 18, 19, 70], which is also known as deductive or declarative debugging [71, 72], assertions [73, 70] and program slicing [21, 15, 16]. However, as yet, these techniques have not been applied in a model-transformation context.

In algorithm debugging, the declarative program's computation tree is traversed in an attempt to find a function call or relationship whose output is incorrect, but all its related (child) outputs are correct [70]. Shapiro's original algorithm debugging process is heavily dependent on an oracle [70], i.e. a developer with understanding of the *correct* output for each function or relationship. Although some techniques, such as those presented by Fritzon et al. [17] and Naish [71], can significantly reduce this burden on the developer, the requirement for the developer to interact with the debugging process at each step still exists. The oracle is walked through a series of questions to determine the *correctness* of each output. The answers to these questions are used to drive the debugging process, eliminating correct branches under the assumption that a correct output of one function, indicates that all its sub-functions are also correct. Eventually a function will be identified where its output is incorrect, but the output of all of its children is correct, effectively localising the bug.

Assertions are constraints or attributes in a program that should always hold true. Puebla et al. [73] demonstrate that assertions can be used in declarative languages to eliminate known correct branches, in a similar process to that used by algorithm debugging. The key difference of assertions over algorithm debugging is that the oracle can be avoided in situations where the assertion can fully evaluated in the program. Puebla et al. [73] differentiate between assertions which can be evaluated for the program, where the assertion is tagged as *true*, and assertions which require oracle interaction, where the assertion is tagged as *trust*.

Program slicing is the decomposition of a program into the smallest possible component whilst maintaining a particular behaviour [21]. In debugging, this behaviour is the bug or side-effect that is being analysed. The program is reduced, or sliced around the identified problem. Recursively this technique is applied, reducing the program down until the behaviour changes, then back-tracking until the behaviour returns. The process is then repeated, trying to reduce the program at an alternate point. At each slice, the bug is being localised

to a smaller part of the program, thus achieving the debugging goal.

These three techniques, provide a different view of the same goal. Each one is attempting to narrow or localise the search for a bug to the smallest possible part of a program. In algorithm debugging and assertions, this is achieved by identifying and eliminating the positive or correct outputs of a program. This contrasts with program slicing which concentrates on the negative or incorrect outputs of the program. Importantly all three share a common trait, in that they are only concerned with the declarative knowledge of a program, that is only its outputs and not the steps to determine the outputs [71]. Localisation

Algorithm debugging, assertions and program slicing are relevant to debugging model-transformations. Algorithm debugging is the most suited approach, with relationships defined in transformation often requiring the developer to completely understand the correct output. However, each approach is not mutually exclusive, and taken together they provide a number of interesting avenues of investigation. Assertions would be particularly useful to identify parts of the transformation that should never fail (i.e. always produce a result) and program slicing would be useful in reducing or eliminating side effects that are not related to the bug being localised.

#### **2.2.4 Bug Correction**

The ultimate goal of the debugging process is always the removal of bugs. By matching common bug patterns and their relative solutions there are methods which could assist in resolving bugs. Traditionally the majority of effort is spent on bug localisation [57, 58, 56] and the case in model-transformations is no different. As such, the current aims of this research are concentrated on providing feedback to MDE modelers on the types and locations of bugs, leaving automated resolution techniques for future research.

### **2.3 Discussion**

Research into model-driven engineering and its subsequent application to real-world development shows the tremendous potential of treating models as first-class artifacts of the development process. This potential is now being translated into real world impact [1]. Experience has shown both productivity [31] and quality [34] improvements. However, it has also highlighted the need for better process and tool support [2].

To address these concerns, there are a number of researchers looking into verification techniques for model-transformations [7, 8, 63, 64, 65]. However, there has been little research into bug localisation techniques for model transformation. To initiate this investigation, the relationship between debugging of declarative logic languages and declarative model-transformation languages has been shown. Related work into debugging of declarative logic languages, particularly that of Shapiro [74, 70] and Naish [71, 72] will have an impact on any outcomes delivered by the research presented in this report.

## 3 Research Methodology and Design

### 3.1 Research Approach

The primary aim of this research is to develop a set of algorithms and patterns to assist with the debugging process in model-driven engineering. This will be provided in the form of a concrete implementation of a debugging framework, in the context of EMF and Tefkat, utilising the algorithms and patterns in practice. With the practical nature of this outcome in mind, an action research methodology shall be applied over several iterations. In order to evaluate outcomes for each iteration, other research techniques, in particular a running case study, have been incorporated into the overarching action research implementation.

### 3.2 Implementation

The research will take place over five action research iterations:

1. Definition of the research problem.
2. Identification of model-transformation debugging questions and classification of model-transformation bugs.
3. Forensic debugging within the context of currently available trace information.
4. Enhancement to trace information produced by model-transformations.
5. Forensic debugging utilizing the improved trace information.

#### 3.2.1 Iteration 1: Problem Definition

The first task is the elaboration of the research area. A literature review shall be undertaken to analyse MDE tools and processes together with general verification and debugging techniques. This literature review shall ascertain the current state of the research problem, identifying related research that can be built upon. This understanding will be used to refine the problem into a clearly defined set of research questions. These research questions can then be fed into subsequent iterations to assist in targeting specific outcomes. The literature review shall be maintained throughout the research, and shall be incorporated into the final thesis.

#### 3.2.2 Iteration 2: Defining the Debugging Questions

Using the outcomes from the first iteration, iteration 2 aims to identify a set of debugging questions to be addressed. The debugging questions shall encompass the manual steps a developer works through to localise and correct a model-transformation that does not execute as expected. The basis for these debugging questions, shall be derived from several sources, including:

- The literature review from iteration 1.
- The research teams experience in implementing model-transformations.

- Informal survey of model-transformation issues raised in public forum, in particular, the Tefkat news group [75] provides a catalog of model-transformation problems from various sources.

Using these debugging questions, and knowledge of their solutions where possible, a taxonomy of model-transformation bugs shall be constructed. The taxonomy shall include the classes of bugs that occur, the relationships between the classes of bugs and the identifying characteristics of each bug. This bug taxonomy will be used to classify solutions to the debugging questions in later iterations.

### **3.2.3 Iteration 3: Forensic Debugging**

Iteration 2 has identified what can go wrong, the bug taxonomy, and how a developer goes about localizing and correcting the problem, the debugging questions. Iteration 3, builds on this, exploring the information available to answer the debugging questions. A series of debugging algorithms shall be developed and investigated to evaluate their applicability to debugging of model transformations. Of particular interest is the benefits and limitations of debugging forensically and the level of automation which can be achieved. Upon completion of this iteration, the research should identify:

- The types of bugs that can be addressed forensically.
- A set of forensic approaches/algorithms for answering debugging questions.
- Potential improvements to the debugging information available, to better support forensic debugging.

On top of these theoretical outcomes, a working prototype, using Tefkat and EMF, should demonstrate the approaches in practice. This practical outcome, with the help of a representative case study, will contribute to the evaluation of each technique.

It is intended that the theoretical outcomes of iteration 3 will be used as the basis for a research paper discussing forensic debugging techniques for model-transformations.

### **3.2.4 Iteration 4: Trace Enhancements**

Iteration 3 identifies the shortcomings of and potential improvements to the transformation outputs for debugging, specifically the trace model. Iteration 4 shall use this as a starting point to investigate and implement improvements to the trace information produced by Tefkat. The experience of implementing these improvements for Tefkat will then be generalized to identify the core requirements for trace information in model-transformations with respect to debugging. The research undertaken in iteration 4 is to be used as the basis for a second research paper concentrating on the evaluation of current trace information and the proposed enhancements.

### 3.2.5 Iteration 5: Forensic Debugging with Extended Trace

Iteration 4 implements improvements to the available trace information. Building on the work from iteration 3, this information can now be leveraged to better address the debugging questions. Of particular interest will be the questions that could not be addressed efficiently or at all with the original trace information.

In addition to the work to enhance the debugging algorithms, this iteration will be used to evaluate the outcomes to date. These evaluations include:

- The usefulness of the trace enhancements to forensic debugging.
- The practicality of forensic debugging for model-transformations.
- The classes of bugs particularly suited to this approach.
- The reasons why the other classes of bugs can't be addressed, or can't be addressed practically.

To conduct these evaluations, two mechanisms shall be used. Firstly, the case study originally introduced in iteration 3 shall be extended to incorporate the final forensic debugging work. Secondly, the prototype shall be released to the Tefkat users community. Feedback from this community will be solicited through networks of known users and the Tefkat news group [75]. Users who choose to participate will be asked to apply the debugging tools to both correct and incorrect model-transformations. From this, it is expected that users will be able to evaluate the effectiveness of the techniques and provide feedback on aspects of model-transformation debugging that have not been addressed.

## 3.3 Outcomes

The objectives of this research can be divided into three categories, which combine to address the research questions defined in Section 1.2. The categories are, theoretical outcomes, practical outcomes and validation and verification outcomes.

The anticipated theoretical outcomes for this research are:

- The definition of generic algorithms and patterns which can be applied to forensic debugging in model-driven engineering.
- The identification of limitations in the current state of trace information generated by model-transformation together with a set of proposed enhancements to address these limitations.
- Evaluation of forensic debugging approaches for model-transformations.

These theoretical outcomes have two key deliverables:

1. Two research papers: one detailing the application of forensic debugging to model-transformations; one establishing the current state of trace in model transformations and its applicability to debugging as well as the deduced requirements for trace information in debugging and the suggested enhancements. These papers will present research progress and findings to the model-driven engineering and debugging research communities. also

2. A thesis which directly addresses the research questions (see subsection 1.2) including a complete literature review and thorough documentation and evaluation of the outcomes.

The anticipated practical outcome for this research is a prototype implementation of the forensic debugging algorithms and patterns. The key deliverables for the practical outcomes are:

1. Implementation of a forensic debugging framework utilising Tefkat and EMF 3.4.
2. Enhancements for Tefkat, based upon the findings to enhance trace output for use in forensic debugging.

The anticipated verification and validation outcomes for this research are:

- Validation that the debugging questions presented are applicable to model-transformations.
- Verification that the debugging algorithms address the debugging questions, and do so in a practical manner

These verification and validation outcomes have four key deliverables:

1. A case study of debugging the *UML to Relational* transformation.
2. Verification results of final implementation against transformations with known problems.
3. Verification results of final implementation against transformations with no known problems.
4. Documented analysis of feedback from Tefkat and MDE communities on the validity of the debugging questions and corresponding solutions.

The *UML to Relational* [76] transformation forms the canonical example for MDE research. It demonstrates a broad spectrum of model-transformation features within the context of a well-understood problem space.

As debugging is largely a quality-oriented task, any accepted contributions have an expectation of reliability and accurate, verifiable results. To achieve both quality aims, reliability and verifiability, the entire research approach is to be under-pinned with a number of quality processes in addition to the verification and validation outcomes:

- Upfront construction of test cases, a test-driven research approach. Test cases should include both known correct and known incorrect transformation, i.e. negative and positive test cases.
- Early and constant exposure to Tefkat community.
- Early and constant exposure to experienced MDE practitioners.
- All investigation is to be undertaken in the context of a case study, the canonical UML to Relational example, to provide a specific problem context and keep solutions on track.
- Sanity checks against a variety of complex models and transformations.

## 3.4 Prototype

As specified in the outcomes, a key component of this research is a prototype debugging framework. This prototype shall be used to experiment and test debugging approaches throughout the project as well as providing a basis for validation of the project outcomes.

### 3.4.1 Underlying Technology

A core design constraint of the prototype is to concentrate and limit its function to the relevant, research oriented aspects of debugging model-transformations. This means that the prototype should not be concerned with pre-existing concepts, such as modeling, meta-modeling, transformations and language concerns. To meet address this, the prototype shall be reliant on existing technologies and MDE tools:

- Java [77] - The tool shall be developed using the Java programming language, it provides an effective development language and environment with many MDE tools and libraries.
- Eclipse [78] - Eclipse provides an extensible Java development environment with support for a number of MDE tools, Eclipse will provide the user interface components for the prototype.
- Eclipse Modelling Framework (EMF) [12] - The Eclipse modelling framework provides the tools required for meta-modeling and interacting with models.
- Tefkat [79] - Tefkat provides a model transformation engine and language built upon EMF.

### 3.4.2 Functionality

It is expected that Iteration 2, see Section 3.2.2, will identify sets of questions that need to be answered when debugging a model transformation. These questions may be asked in terms of the input models, the transformation, the output models or the trace information. The prototype shall provide a framework that allows each of the questions addressed by this research to be asked in an EMF/Tefkat model-transformation environment. This means that a user shall be able to select some MDE artifact (or element of an artifact) and be presented with a set of possible questions that can be asked about the artifact and its relationships. Once the user selects a question, the debugging framework will employ the techniques investigated in this research to provide meaningful output, allowing the user to localise bugs in the transformation or models.

### 3.4.3 Interfaces

The prototype shall be exposed as a component in Eclipse. As the prototypes main concern is demonstrating the underlying debugging techniques, it is expected that only a minimalistic user interface shall be provided. The interface shall allow debugging questions to be asked of various MDE artifacts and the answers will be returned as straight forward, primarily text based, output. Visualisation of the output is viewed as a critical part of the user experience and

acceptance of any debugging tool, however advanced visualisation is out of scope for this research to allow for focus on the debugging techniques themselves.

### 3.5 Scholarly Activities

During the course of this candidature I have taken part in a number of scholarly activities. These include:

- Completion of ITN100 - Research Methodologies with a High Distinction.
- Co-authored conference paper - “*Forensic Debugging of Model Transformations*” [80] - which was accepted for the 10th International Conference on Model-Driven Engineering Languages and Systems

M. Hibberd, M. Lawley, and K. Raymond. Forensic Debugging of Model Transformations. *Model Driven Engineering Languages and Systems*, pages 580–604, 2007.

- Presented “Forensic Debugging of Model Transformations” paper at MoDELs 2007 Conference, Nashville, October 2007
- Track Tefkat users group for model-transformation bugs and feedback.
- Attended multiple Faculty of IT seminars presented by fellow students.

### 3.6 Research Time-line

Figure 5 shows the proposed time-line for completion of this masters candidature. Each iteration has its own tasks and expected outcomes, however, it is important to note that the scheduling of the iterations overlap. This is due to two factors:

1. The time versus effort ratio is not constant for each task. Some tasks may progress over long periods of time, but may only require a small number of hours each week. Therefore multiple tasks are undertaken at the same time.
2. It is not expected that each iteration can be completed in total isolation. For example, Iteration 4, enhancing the trace information, and Iteration 5, taking advantage of the trace information, will benefit from a back and forth iteration. Iteration 4 will investigate the initial trace enhancements and feed this to Iteration 5. Iteration 5 will attempt to use the trace enhancements, identifying further issues to be investigated in Iteration 4.

### 3.7 Risk Management

As with all software development projects, particularly when combined with new research, this project faces certain risks. To help reduce and manage the risk involved, the key issues related to completing the research within time and with appropriate quality, rigor and contribution to knowledge have been identified with plans to minimise their effect.

Activity	2006				2007				2008			
	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4		
<b>Milestones</b>												
Project Reports												
Ethics Application												
Confirmation												
Final Seminar												
Submission												
Conference/Journal Papers												
<b>Course Work</b>												
ITN100 – Research Methodologies												
<b>Research Tasks</b>												
Stage 1: Familiarization												
Investigate MDE background and tools												
Investigate debugging background												
Define research problem												
Complete initial literature review												
Stage 2: Debugging Questions												
Define debugging questions												
Define model-transformation bug taxonomy												
Stage 3: Forensic Debugging												
Analyse debugging questions												
Select debugging questions to address forensically												
Design forensic debugging algorithms for each question												
Implement algorithms using Teikrat and EMF												
Stage 4: Trace Enhancement												
Identify additional trace information												
Implement additional trace output using Teikrat												
Stage 5: Forensic Debugging w Extended Trace												
Select questions that could benefit from extended trace												
Rework existing algorithms to leverage extended trace												
Design new algorithms to leverage extended trace												
Implement new algorithms using Teikrat and EMF												
Solicit feedback from Teikrat community												
<b>Thesis Preparation</b>												
Literature Review												
Thesis Body												
Conclusions, Introduction and Abstract												

Figure 5: Time-line for completion of Masters candidature.

The first issue is the completeness of the proof-of-concept implementation. The primary goals of this research are the theoretical outcomes to be presented in the Thesis. This means that the completeness of the concrete implementation is important, but not critical, to achieving the research goal. The implementation must provide enough support to allow evaluation of the techniques, but all other considerations, such as visualisation and robustness, must be considered secondary. These tasks will only be completed if time allows.

The second issue which must be considered is the availability of the researcher. The completion of this project will require considerable time and effort for a student with a part-time workload. Work and family commitments will have to be closely monitored to ensure that all project milestones can be met on time. The 6 monthly progress reports submitted to the Faculty and Research Students Centre assist in managing this concern.

## **3.8 Budget**

### **3.8.1 Hardware/Software**

This research has minimal implementation costs. All software utilised is open-source software. Tefkat is provided under the Lesser GNU Public License (LGPL) [81]. Eclipse and EMF is provided under the Eclipse Public License (EPL) [82]. The Java Development Kit (JDK) binaries are provided for free under a product and version specific license [83, 84] from SUN Microsystems. Hardware has been provided by both QUT (at Margaret St) and by the researcher (home use) for completion of this research.

### **3.8.2 Travel**

There is a plan in place for at least two conference papers in the scope of the Masters degree. Upon acceptance of papers this shall require travel to the corresponding conference for presentation. This travel shall be applied for and documented in-line with faculty and university policies and procedures [85, 86].

To date, one conference paper [80] has been accepted and travel occurred from 29th of September to the 6th of October in order to present at and attend the MoDELS 2007 conference in Nashville, USA.

## **3.9 Ethical Clearance**

The design of this research has limited ethical implications. The only aspect for ethical consideration is the participation of users in trials of the MDE debugging framework implementation for feedback. Participation in evaluating the software will be at the discretion of each individual or team and the privacy of all users involved will be maintained.

According to QUT policies, the level of human interaction required by this research is classed as Level One (Low Risk). That is, *“human research with no significant risks or ethical issues, e.g. an anonymous questionnaire or an interview involving non-sensitive matters”* [87]. Prior to the commencement of any user involvement a QUT Human Ethics application form is required to be submitted to obtain Level 1 ethical clearance. This application is timetabled for quarter one 2008, to commence interaction in quarter two 2008.

## 4 Work to Date

The previous section has identified the iterations into which this research has been decomposed. There has been significant work completed towards the outlined research. Iterations 1 through 3 have been completed with their outcomes contributing to the conference paper, *Forensic Debugging of Model Transformations* [80], presented at the 10th international conference for Model Driven Engineering Languages and Systems.

This remainder of this section identifies the outcomes achieved so far, for each iteration of the research. The details of this work is left to the attached conference paper, see appendix A.

### 4.1 Iteration 1: Problem Definition

The first iteration of this research involved the completion of the course work requirements, ITN100 - Research Methodologies, as well as the elicitation and refinement of the research problem and corresponding questions. Both of these items contributed to a detailed understanding of the research area and, in turn, helped to provide a clearly defined and tractable research problem.

The research methodologies course work created a number of important outcomes, specifically an initial problem definition, literature review and research plan. These documents, in an updated form, are the basis for this confirmation and articulation report.

The elicitation and refinement of the research problem served as a basis for the later forensic debugging work, helping to define the requirements and actions for the second and third iterations. Sections 1.1 and 1.2 demonstrate the outcomes of iteration 1.

### 4.2 Iteration 2: Defining the Debugging Questions

Following on from the problem definition phase in iteration 1, a set of debugging questions have been defined. Section 3 of the attached paper, appendix A, provides an overview and discussion of these debugging questions. From these questions, a classification of model-transformation bugs has also been established.

Three categories of debugging questions have been identified: logical debugging questions, well-formedness questions and analysis questions. The analysis questions are further broken down into bug smells and information discovery sub-categories.

A number of bug types have been identified. These can be split into two primary categories: bugs which result in valid but incorrect output and bugs which result in invalid output.

The debugging questions and bug classes helped to establish a framework for the next iteration of the research.

### 4.3 Iteration 3: Forensic Debugging

To address the questions defined in iteration 2, two model-transformation specific debugging techniques have been established and investigated. These tech-

niques, static-analysis of the model-transformation artifacts and forensic reenactment, are discussed in section 5 of the attached paper, see appendix A.

Static-analysis techniques are concerned with data-mining and synthesis. In a number of cases the information required to answer a debugging question was available to the developer, however the volume and verbosity of the data prevented efficient use of this information. Automated static-analysis techniques address this by intuitively selecting and filtering the appropriate information.

Forensic reenactment techniques are concerned with inferring additional debugging information that is not immediately available in the model-transformation output. By selectively re-executing limited parts of a model-transformation, as per program slicing techniques, this additional information can be determined without access to the internals of the live transformation engine.

#### **4.4 Iteration 4 and Beyond**

No key outcomes have been achieved for iteration 4 or 5. However, the knowledge gained in the first three iterations has provided a solid base to start investigating the improvements which can be made to the trace information produced by model-transformations and how that enhanced trace information can be leveraged.

## 5 Articulation

The previous sections have outlined the research problem, plan and work to date for confirmation of Masters candidature. This research problem has been highly challenging whilst still being tractable. As the work has progressed a number of future research avenues have been highlighted. Two of these avenues present interesting problems that are highly related and complementary to the current forensic debugging work:

1. Live debugging of declarative model-transformations.
2. Incorporation of test / constraint feedback into debugging algorithms.

This section extends the research problem and plan to incorporate this additional work for articulation to PhD candidature.

### 5.1 Extended Research Problem

The nature and artifacts of model-transformations make forensic debugging techniques extremely effective. However, there are details and complex cases where the forensic techniques are not sufficient. Information such as execution order, intermediate states and other ephemeral data are not available for post-hoc analysis, and due to the complexity and volume of this information it is not practical to make it available.

When a transformation executes, the engine or runtime environment is responsible for building up state information and executing individual steps of the transformation, as it can, to complete the transformation. By interrogating the engine, the execution order and exact state information at each step can be determined. This interaction with the internals of the engine or runtime environment provides the definition for this extended research, live debugging.

Lawley and Steel have presented a live debugging approach for Tefkat [13] using a traditional interactive approach common to imperative programming languages. They highlighted the limitations this approach has with declarative languages, sighting the non-linear execution and re-ordering of terms at runtime as significant limitations. This research aims to build on this experience, utilising the information available in a live environment to develop debugging paradigms and techniques specifically suited to declarative model-transformations.

This aim is further extended to utilize feedback from bug identification mechanisms, such as testing and model constraints, in order to improve both forensic and live debugging algorithms.

### 5.2 Extended Research Questions

To encompass these new facets of the research problem, two additional research questions have been identified to extend those already defined in Section 1.2.

#### 5.2.1 Research Question 3

*To what extent can access to the internals of the transformation engine be leveraged for debugging of declarative model-transformations, and how effective can these debugging techniques be compared with the traditional step-through debugging paradigm?*

This question can be further broken down, into:

- *What additional information can be determined in a live debugging environment compared with the forensic debugging environment?*
- *Given the problems with the step-through debugging paradigm for declarative languages, what alternative techniques could be applied within a live debugging context and how could the step-through paradigm be altered to better support non-linear execution?*

#### **5.2.2 Research Question 4**

*How can testing and constraints in model-driven engineering be leveraged as inputs to provide more effective debugging solutions with a greater level of automation?*

This question can be further broken down, into:

- *What testing and constraint mechanisms are available for model transformations?*
- *What types of feedback do these mechanisms provide?*
- *How can that feedback be utilized to improve the debugging process?*

### **5.3 Revised Research Plan**

The extensions to the proposed research are closely related in content and structure to the original research plan. There is no significant impact on the research methodology used or the evaluation techniques required.

The changes to the research plan encompass additional research iterations on top of those proposed in Section 3.2 as well as a revised time-line for completion.

To address the additional research question put forward in Section 5.2. Three further research iterations shall be undertaken.

1. Addressing debugging questions in a live debugging environment.
2. Alternative live debugging paradigms.
3. Leveraging test and constraint output in debugging of model transformations.

#### **5.3.1 Iteration 6: Addressing Debugging Questions in a Live Environment**

Iteration 6, shall involve analysis of the information available in a live transformation environment and its comparison with that available in a forensic environment. Using this information, the debugging questions, originally defined in iteration 2, will be re-investigated from two angles. Firstly, for the questions that were addressed forensically, to what extent can the addition of live debugging information be leveraged to answer the questions either more concisely, more efficiently or with a greater level of automation. The second angle is to

attempt to address those debugging questions that were difficult to address or not addressed at all by forensic techniques.

This iteration has similar outcomes to iteration 3, except in the context of live debugging. Upon completion of this iteration, the research should identify:

- The types of bugs that are suited to being debugged in a live context.
- A set of approaches for answering debugging questions in a live context.
- Alternate debugging approaches that are suited to live debugging of declarative languages, in particular declarative model-transformations.

The case study and tests from iteration 3 and 5 will be updated and used for evaluation of these live debugging techniques. Communications with the Tefkat users community, established in iteration 5, will continue through the remaining iterations.

### 5.3.2 Iteration 7: Alternate Live Debugging Paradigms

It is expected that iteration 6 will show that traditional live debugging techniques commonly applied to imperative languages such as interactive, step-through debugging do not translate well to languages with non-linear execution paths, such as declarative model-transformations. Iteration 7 aims to investigate alternative live debugging paradigms which better suite declarative languages and more specifically model-transformations. This iteration will encompass research into new debugging approaches as well as investigation into ways to better adapt traditional interactive debugging into a declarative context.

Upon completion of this iteration, the research should identify:

- The characteristics of live debugging approaches that are suited to declarative languages.
- The characteristics of live debugging approaches that are *not* suited to declarative languages.
- New and adapted approaches to live and interactive debugging of declarative languages based upon the lessons learned.

It is intended that iterations 6 and 7 will form the basis of a research paper concerned with live debugging techniques applicable to declarative model transformations.

### 5.3.3 Iteration 8: Model-Transformation Testing and Constraints

The previous iterations consider the debugging problem in isolation. This is adequate in terms of developing new debugging techniques. However to integrate the debugging techniques into a development process and to take full advantage of the information available, the debugging problem must be considered in the context of testing and constraints or bug identification.

Test failures and constraint violations are often the key triggers for the debugging process. They also provide useful contextual information about the problem at hand, which can be used to further enhance the automation of a debugging process[73]. This research iteration involves investigating the relationships of testing and constraints with debugging of model-transformations.

The aim of this iteration is to increase the information available to the debugging environment and provide a higher level of automation in the bug-localisation processes. The outcomes of this iteration will contribute to a fourth research paper addressing model-transformation testing and constraints with respect to debugging.

#### **5.3.4 Revised Time-line**

The increase in scope has a significant impact on the time-line. However this impact is in line with the expectations of effort required for a PhD candidature compared with that required for a Masters candidature. Based on the work completed to date and the estimated effort for each of the future research components, the candidature can be completed within the recommended time-frames.

Figure 6 reflects the updated time-line for completion.

Activity	2005				2006				2007				2008				2009			
	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4		
<b>Milestones</b>																				
Project Reports																				
Ethics Application																				
Confirmation																				
Final Seminar																				
Submission																				
Conference/Journal Papers																				
<b>Course Work</b>																				
ITN100 – Research Methodologies																				
<b>Research Tasks</b>																				
Stage 1: Familiarization																				
Investigate MDE background and tools																				
Investigate debugging background																				
Define research problem																				
Complete initial literature review																				
Stage 2: Debugging Questions																				
Define debugging questions																				
Define model-transformation bug taxonomy																				
Stage 3: Forensic Debugging																				
Analyse debugging questions																				
Select debugging questions to address forensically																				
Design forensic debugging algorithms for each question																				
Implement algorithms using Teekat and EMF																				
Stage 4: Trace Enhancement																				
Identify additional trace information																				
Implement additional trace output using Teekat																				
Stage 5: Forensic Debugging w Extended Trace																				
Select questions that could benefit from extended trace																				
Rework existing algorithms to leverage extended trace																				
Design new algorithms to leverage extended trace																				
Implement new algorithms using Teekat and EMF																				
Stage 6: Live Debugging																				
Identify questions that could not be answered forensically																				
Identify questions that could benefit from live knowledge																				
Design live debugging algorithms for each question																				
Stage 7: Alternate Live Debugging Paradigms																				
Analyse traditional live debugging paradigms																				
Investigate alternate paradigms for declarative languages																				
Implement declarative live debugging using Teekat and EMF																				
Stage 8: Model Transformation Testing and Constraints																				
Analyse existing model testing and constraint mechanisms																				
Identify new testing and constraint mechanisms																				
Investigate how these provide input for debugging																				
Implement this interaction using Teekat and EMF																				
Solicit feedback from Teekat community																				
<b>Thesis Preparation</b>																				
Literature Review																				
Thesis Body																				
Conclusions, Introduction and Abstract																				

Figure 6: Revised time-line for completion of PhD candidature.

## References

- [1] Schmidt, D.: Guest Editor's Introduction: Model-Driven Engineering. *Computer* **39**(2) (2006) 25–31
- [2] Kulkarni, V., Reddy, S.: Introducing MDA in a large IT consultancy organization. *Proceedings of the XIII Asia Pacific Software Engineering Conference* (2006) 419–426
- [3] Selic, B.: The pragmatics of model-driven development. *Software, IEEE* **20**(5) (2003) 19–25
- [4] Kolovos, D., Paige, R., Polack, F.: Model Comparison: A Foundation for Model Composition and Model Transformation Testing. *International Conference on Software Engineering* (2006) 13–20
- [5] Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. *Models in Software Engineering* **4199** (2006) 321–335
- [6] Baresi, L., Ehrig, K., Heckel, R.: Verification of model transformations: A case study with BPEL. *Proc. of the 2nd Symposium on Trustworthy Global Computing, TGC'06* (2006)
- [7] Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the Specification of Model Transformation Contracts. *Proceedings of OCL&MDE* (2004)
- [8] Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. *First International Workshop on Model, Design and Validation* (2004) 29–40
- [9] Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **16**(5) (1994) 1512–1542
- [10] Clarke, E., Grumberg, O., Peled, D.: *Model Checking*. MIT Press (1999)
- [11] Dwyer, M., Hatcliff, J., Robby, R., Pasareanu, C., Visser, W.: Formal Software Analysis Emerging Trends in Software Model Checking. *International Conference on Software Engineering* (2007) 120–136
- [12] Eclipse Foundation: Eclipse Modeling Framework Project (EMF). Available at: <http://www.eclipse.org/modeling/emf/> (2007) accessed February 20th, 2007.
- [13] Lawley, M., Steel, J.: Practical Declarative Model Transformation With Tefkat. *Lecture Notes In Computer Science* **3844** (2006) 139
- [14] Larus, J.: Efficient program tracing. *Computer* **26**(5) (1993) 52–61
- [15] Weiser, M.: Programmers use slicing when debugging. *Communications of the ACM* **25**(7) (1982) 446–452
- [16] 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

- [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] Kokai, G., Harmath, L., Gyimothy, T.: Algorithmic debugging and testing of Prolog programs. *Workshop on Logic Programming Environments* (1997) 14–21
- [19] Paakki, J., Gyimothy, T., Horvath, T.: Effective Algorithmic Debugging for Inductive Logic Programming. *Proc. of the Fourth International Workshop on Inductive Logic Programming (ILP-94)* Bad Honnef/Bonn Germany September (1994) 12–14
- [20] Chan, T., Lakhotia, A.: Debugging program failure exhibited by voluminous data. PhD thesis, University of Southwestern Louisiana (1994)
- [21] Weiser, M.: Program slicing. *Proceedings of the 5th international conference on Software engineering* (1981) 439–449
- [22] Object Management Group: MOF QVT Final Adopted Specification. *OMG Document ptc/2005-11-01*, November (2005)
- [23] Object Management Group: UML 2.0 Object Constraint Language (OCL) Specification. *OMG Document ad/2003-01-07*, January (2003)
- [24] Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. *Software, IEEE* **20**(5) (2003) 36–41
- [25] Object Management Group: Model Driven Architecture - A Technical Perspective. *OMG Document ab/2001-02-01*, February (2001)
- [26] Soley, R., et al.: Model Driven Architecture. *OMG Document omg/2000-11-05*, November (2000)
- [27] Miller, J., Mukerji, J., et al.: MDA Guide Version 1.0.1. *OMG Document omg/2003-06-01*, June (2003)
- [28] Siegel, J., et al.: Developing in OMG’s Model-Driven Architecture. *OMG Document omg/2001-12-01*, December (2001)
- [29] Bock, C.: UML without pictures. *Software, IEEE* **20**(5) (2003) 33–35
- [30] Franz, M.: Code-Generation On-the-Fly: A Key to Portable Software. PhD thesis, Swiss Federal Institute Of Technology (1964)
- [31] Warmer, J.: A Model Driven Software Factory Using Domain Specific Languages. *Lecture Notes in Computer Science* **4530** (2007) 194
- [32] Sprinkle, J.: Model-integrated computing. *Potentials, IEEE* **23**(1) (2004) 28–30
- [33] Sztipanovits, J., Karsai, G.: Model-integrated computing. *Computer* **30**(4) (1997) 110–111

- [34] Weil, F., Mastenbrook, B., Nelson, D., Dietz, P., van den Berg, A.: Automated Semantic Analysis of Design Models. *Model Driven Engineering Languages and Systems* (2007)
- [35] Object Management Group: Architecture-driven modernization roadmap. Available at: <http://adm.omg.org/ADMTFSeptember> 26th, 2006.
- [36] Fleurey, F., Breton, E., Baudry, B., Nicolas, A., Jezequel, J.: Model-Driven Engineering for Software Migration in a Large Industrial Context. *Model Driven Engineering Languages and Systems* (2007)
- [37] MacDonald, A., Russell, D., Atchison, B.: Model-driven development within a legacy system: an industry experience report. *Software Engineering Conference, 2005. Proceedings. 2005 Australian* (2005) 14–22
- [38] Gray, J., Bapty, T., Neema, S., Schmidt, D., Gokhale, A., Natarajan, B.: An Approach for Supporting Aspect-Oriented Domain Modeling. *Proceedings of the Second International Conference on Generative Programming and Component Engineering* (2003) 22–25
- [39] Jayaraman, P., Whittle, J., Elkhodary, A., Gomaa, H.: Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. *Model Driven Engineering Languages and Systems* (2007)
- [40] Object Management Group: Unified Modelling Language: Superstructure, v2.0. *OMG Document formal/2005-07-04*, August (2005)
- [41] Object Management Group: Unified Modelling Language: Infrastructure, v2.0. *OMG Document formal/2005-07-05*, August (2005)
- [42] Bézivin, J., Jouault, F., Kurtev, I., Valduriez, P.: Model-Based DSL Frameworks. *Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA* (2006) 22–26
- [43] van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices* **35**(6) (2000) 26–36
- [44] Crawley, S., Davis, S., Indulska, J., McBride, S., Raymond, K.: Meta-meta is better-better. *Proceedings of the IFIP WG 6* (1997)
- [45] Object Management Group: Meta Object Facility (MOF) Core Specification. *OMG Document formal/2006-01-01*, January (2006)
- [46] Object Management Group: MOF 2.0/XMI Mapping Specification, v2.1. *OMG Document formal/2005-09-01*, September (2005)
- [47] Sendall, S., Küster, J.: Taming Model Round-Trip Engineering. *OOPSLA/GPCE: Best Practices for Model-Driven Software Development* (2004)
- [48] 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)

- [49] Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Systems Journal* **45**(3) (2006) 622
- [50] Mens, T., Czarnecki, K., Van Gorp, P.: A Taxonomy of Model Transformation. *International Workshop on Graph and Model Transformation* (2005)
- [51] Gardner, T., Griffin, C., Koehler, J., Hauser, R.: A review of OMG MOF 2.0 Query - Views - Transformations Submissions and Recommendations towards the final Standard. *MetaModelling for MDA Workshop* (2003)
- [52] Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The Missing Link of MDA. *Lecture Notes in Computer Science* **2505** (2002)
- [53] Tratt, L.: Model transformations and tool integration. *Software and Systems Modeling* **4**(2) (2005) 112–122
- [54] DSTC-IBM-CBOP: MOF 2.0 Query/Views/Transformations, Second revised submission. *OMG Document ad/2004-01-06*, January (2004)
- [55] Aizenbud-Reshef, N., Nolan, B., Rubin, J., Shaham-Gafni, Y.: Model traceability. *IBM Systems Journal* **45**(3) (2006) 516
- [56] 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
- [57] 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
- [58] 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
- [59] Wallace, D., Fujii, R.: Software verification and validation: an overview. *Software, IEEE* **6**(3) (1989) 10–17
- [60] Adrion, W., Branstad, M., Cherniavsky, J.: Validation, Verification, and Testing of Computer Software. *ACM Computing Surveys (CSUR)* **14**(2) (1982) 159–192
- [61] Cleve, H., Zeller, A.: Finding Failure Causes through Automated Testing. *Arxiv preprint cs.SE/0012009* (2000)
- [62] Schmidt, A., Varro, D.: CheckVML: A Tool for Model Checking Visual Modeling Languages. *Uml 2003-The Unified Modeling Language: Modeling Languages and Applications: 6th International Conference, San Francisco, CA, USA, October 20-24, 2003: Proceedings* (2003)
- [63] Varró, D.: Automated formal verification of visual modeling languages by model checking. *Software and Systems Modeling* **3**(2) (2004) 85–113

- [64] Varró, D., Pataricza, A.: Automated formal verification of model transformations. *CSDUML* (2003) 63–78
- [65] Zhao, Y., Fan, Y., Bai, X., Wang, Y., Cai, H., Ding, W.: Towards Formal Verification of UML Diagrams Based on Graph Transformation. *E-Commerce Technology for Dynamic E-Business, 2004. IEEE International Conference on* (2004) 180–187
- [66] Johnson, M.: A software debugging glossary. *ACM SIGPLAN Notices* **17**(2) (1982) 53–70
- [67] McDougall, R., Mauro, J., Gregg, B.: *Solaris (TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall PTR Upper Saddle River, NJ, USA (2006)
- [68] Warren, D., Pereira, L., Pereira, F.: Prolog-the language and its implementation compared with Lisp. *ACM SIGPLAN Notices* **12**(8) (1977) 109–115
- [69] Henderson, F., Conway, T., Somogyi, Z.: *Mercury, an efficient purely declarative logic programming language*. Glenelg, Australia (1995) 499–512
- [70] Shapiro, E.: *Algorithmic Program DeBugging*. MIT Press Cambridge, MA, USA (1983)
- [71] Naish, L.: *Declarative Debugging of Lazy Functional Programs*. Dept. of Computer Science, University of Melbourne (1992)
- [72] Naish, L.: *A Declarative Debugging Scheme*. Department of Computer Science, University of Melbourne (1995)
- [73] Puebla, G., Bueno, F., Hermenegildo, M.: An Assertion Language for Debugging of Constraint Logic Programs. *Proceedings of the ILPS* **97** (1997)
- [74] Shapiro, E.: Algorithmic program diagnosis. *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1982) 299–308
- [75] Tefkat: Tefkat news group. Available at: <http://groups.google.com.au/group/Tefkat> Accessed November 26th, 2007.
- [76] Bezivin, J., Rumpe, B., Schiirr, A., Tratt, L.: Model Transformations in Practice Workshop. Satellite Events at the Models 2005 Conference: MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005: Revised Selected Papers (2006)
- [77] Sun Microsystems, Inc: *Java Standard Edition*). Available at: <http://java.sun.com/javase/> (2008) accessed January 15th, 2008.
- [78] Eclipse Foundation: *Eclipse Platform*. Available at: <http://www.eclipse.org/> (2007) accessed February 20th, 2007.

- [79] Tefkat: Tefkat: The EMF Transformation Engine). Available at: <http://tefkat.sourceforge.net/> (2007) accessed February 20th, 2007.
- [80] Hibberd, M., Lawley, M., Raymond, K.: Forensic Debugging of Model Transformations. Model Driven Engineering Languages and Systems (2007) 580–604
- [81] Free Software Foundation: GNU Lesser General Public License (LGPL). Available at: <http://www.gnu.org/licenses/lgpl.html> (1999) Accessed October 26th, 2006.
- [82] Eclipse Foundation: Eclipse Public License (EPL). Available at: <http://www.eclipse.org/legal/epl-v10.html> (2006) Accessed October 26th, 2006.
- [83] Sun Microsystems, Inc: Java development kit v5.0 binary code license agreement (bcl). Available at: [http://java.sun.com/j2se/1.5.0/jdk-1\\_5\\_0-license.txt](http://java.sun.com/j2se/1.5.0/jdk-1_5_0-license.txt) (2006) Accessed October 26th, 2006.
- [84] Sun Microsystems, Inc: Java runtime environment v5.0 binary code license agreement (bcl). Available at: [http://java.sun.com/j2se/1.5.0/jre-1\\_5\\_0-license.txt](http://java.sun.com/j2se/1.5.0/jre-1_5_0-license.txt) (2006) Accessed October 26th, 2006.
- [85] Queensland University of Technology, Faculty of Information Technology: Grant in Aid. Available at: [http://www.fit.qut.edu.au/research/grant\\_in\\_aid.jsp](http://www.fit.qut.edu.au/research/grant_in_aid.jsp) (2007) Accessed November 3rd, 2007.
- [86] Queensland University of Technology, Finance and Resource Planning: Travel. Available at: <http://www.frp.qut.edu.au/services/travel> (2007) Accessed November 3rd, 2007.
- [87] Queensland University of Technology: Human ethical clearance, what level. Available at: [http://www.research.qut.edu.au/ethics/downloads/forms/human/WHAT\\_LEVEL.pdf](http://www.research.qut.edu.au/ethics/downloads/forms/human/WHAT_LEVEL.pdf) (2006) accessed October 22nd, 2006.

## A Published Work

# 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.

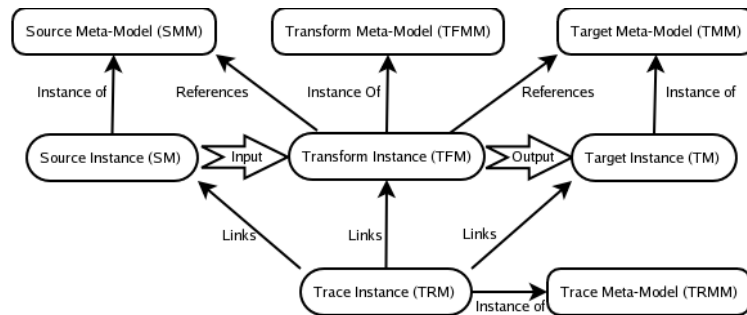
Using these characteristics as a guide, we have utilised the Eclipse Modeling Framework (EMF [7] and the Tefkat [8] model transformation engine. Tefkat uses a declarative approach to model transformations. It has a formal trace model, which links the target, source and transformation. Another important 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.

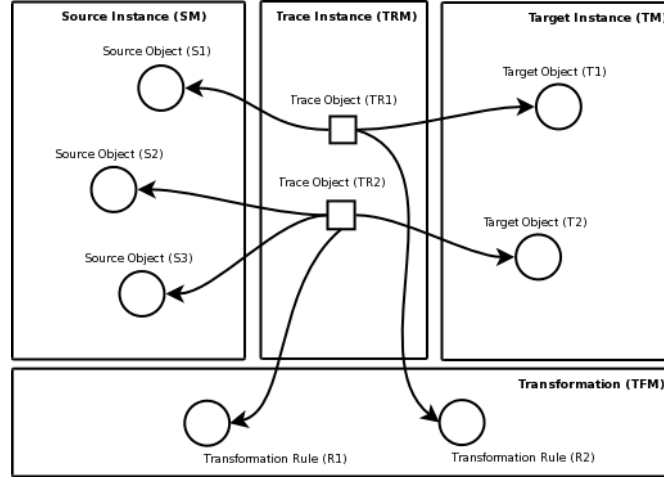


Fig. 2. Model trace environment.

### 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?

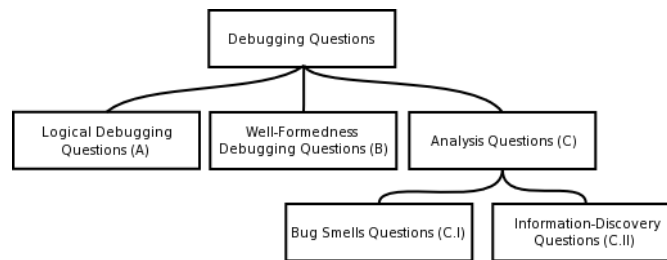
C.II.2 Given a source object, what target objects did it contribute to?

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.

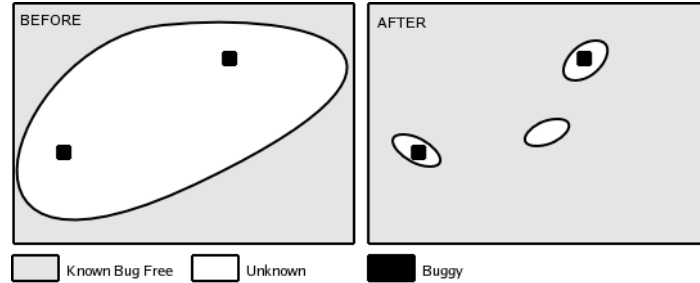
**Well-formedness bugs:** Well-formedness bugs are closely related to the category B debugging questions which result from invalid output. A well-formedness bug occurs when the target model instance does not conform to the target meta-model. Well-formedness bugs are a superset of a number of other bug 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 be 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 in turn help pinpoint the bug's 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<sup>1</sup> 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

---

<sup>1</sup> 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 contributors.** 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), ( $TRM[source-references]$ ), 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.

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

**Source objects that did not contribute to the creation of a target object.** Similarly, 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.

$$y = TRM[source-references] - SM$$

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



```

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 mofInstances
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 mofInstances
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<sup>2</sup>. 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

<sup>2</sup> 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, PATTERNS 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-transformation 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

## 13.2 models 2007 - accepted - forensic debugging

# 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.

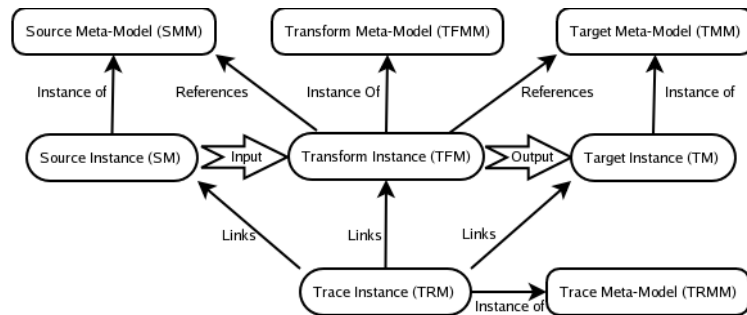
Using these characteristics as a guide, we have utilised the Eclipse Modeling Framework (EMF [7] and the Tefkat [8] model transformation engine. Tefkat uses a declarative approach to model transformations. It has a formal trace model, which links the target, source and transformation. Another important 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.

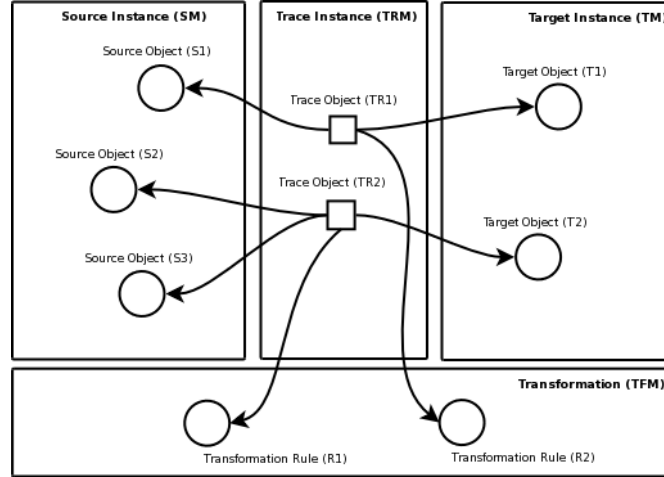


Fig. 2. Model trace environment.

### 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?

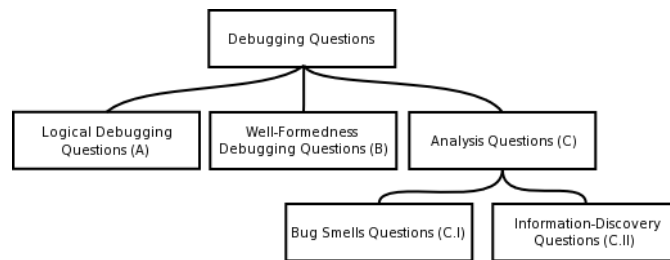
C.II.2 Given a source object, what target objects did it contribute to?

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 be 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 in turn help pinpoint the bug's 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

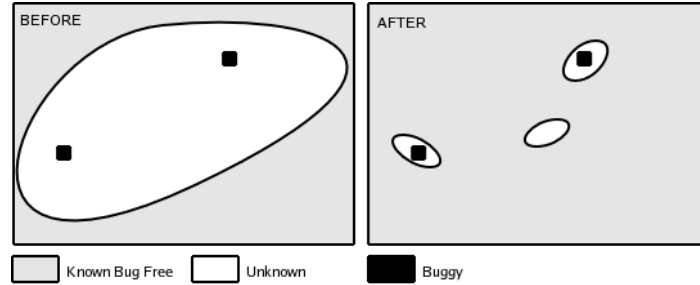


Fig. 4. Before and after localisation of potential problems.

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<sup>1</sup> 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

---

<sup>1</sup> 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 contributors.** 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), ( $TRM[source-references]$ ), 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.

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

**Source objects that did not contribute to the creation of a target object.** Similarly, 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.

$$y = TRM[source-references] - SM$$

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

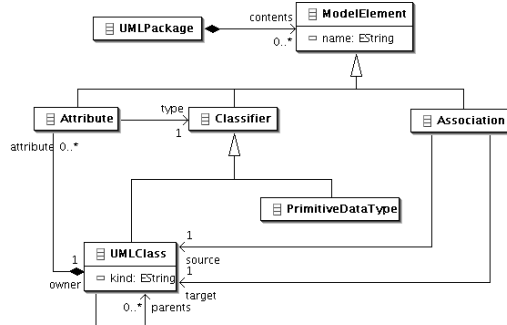


Fig. 6. Simple UML meta-model.

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 mofInstances
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 mofInstances
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<sup>2</sup>. 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

<sup>2</sup> 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, PATTERNS 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-transformation 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

### 13.3 models 2008 - submission - declarative debugging

# Declarative Debugging for Declarative Model Transformations

Mark Hibberd<sup>1</sup>, Michael Lawley<sup>2</sup>, and Kerry Raymond<sup>1</sup>

<sup>1</sup> Queensland University of Technology, Brisbane, Australia  
mt.hibberd@student.qut.edu.au, k.raymond@qut.edu.au

<sup>2</sup> Australian E-Health Research Centre, CSIRO ICT Centre  
michael.lawley@csiro.au

**Abstract.** Declarative model transformation languages provide a powerful, expressive mechanism to define model-to-model transformations. However, this high-level approach to model transformations is currently limited by its dependence on traditional low-level and imperative debugging techniques. This paper proposes that high-level approaches to model-transformation, require high-level debugging techniques. Critical to achieving higher-level debugging is being able to highlight the developer's incorrect assumptions and misinterpretation of the execution model. We exploit data and execution trace to present new visualisation techniques that are able to communicate a more accurate understanding of the transformation execution to the developer. Using these techniques we show how developer's intentions and assumptions can be verified in order to localize model transformations bugs. The approaches are implemented using the Eclipse Modelling Framework and the Tefkat declarative model transformation engine.

## 1 Introduction

Declarative model transformation languages provide a powerful mechanism for defining relationships between data. However, this power comes at the cost of requiring developers to understand a more complex execution model. By their nature, declarative languages have a much lower correlation between the written program and how it is interpreted compared with their imperative counterparts, yet established debugging techniques [1, 2] are limited to low-level approaches adapted from traditional imperative languages.

A bug can be seen as a difference between what is intended and what actually occurs. Based on experience in using and teaching declarative model-transformations, this paper proposes that a developer's ability to localize a bug can be improved by representing aspects of a transformations execution against the static model transformation artifacts. This juxtaposition provides a stronger and more immediate link between the specification of the transformation and its execution; hence allowing developers to more readily verify that their expectations and underlying assumptions were met.

There are a number of approaches to transforming models. Each approach has a number of characteristics [3–6] that affect debugging approaches. This paper is concerned with declarative, domain-specific languages that utilise formal trace models, such as the Relations language defined by OMG’s Queries, Views and Transformations (QVT) Specification [7] and the Tefkat Transformation Engine, which will be used for concrete examples in this paper. General purpose, and imperative languages have their own better-understood debugging techniques which are out of scope for this paper.

The remainder of this paper is organized as follows: Section 2 presents a case for high-level debugging of declarative model-transformation languages and positions this work with respect to debugging in related programming paradigms. Section 3 provides background on the model-transformation environment. Section 4 introduces concepts for summarizing and representing debugging information in a manner that is more closely correlated to the high-level model-transformation artifacts. Section 5 presents visualisation techniques for effectively communicating debugging information to model transformation developers. Section 6 concludes this paper and covers important future work.

## 2 Background

### 2.1 High-Level Debugging

Debugging of declarative languages, including declarative model transformations, is an inherently difficult problem. These languages allow programs to be defined using more expressive constructs, and as a result, this power leads to more complex execution models [8, 9] that do not map linearly to the program source. In terms of writing declarative programs, a more complex execution model is not a problem. The language can provide appropriate abstractions to distance the programmer from the execution model. However, when faced with debugging, the low-level complexities often leak through to the developer due to an impedance mismatch between traditional, low-level, debugging techniques and high-level, declarative transformation languages.

Previous work on debugging in a modelling context, Fujaba’s heap-analysis techniques [1] and Tefkat’s step-wise debugging [2], directly expose low-level execution details. These techniques are more about debugging the transformation *runtime* than the actual transformation. They expose the low-level imperative execution in order to bring the programmer’s understanding and expectations of the transformation in line with the real execution. As noted by Lawley and Steel [2] this is neither effective nor practical. It defeats the purpose of high-level language constructs by forcing the developer to understand the low-level execution model.

Other declarative languages share an identical problem. Naish [10], in particular, emphasizes the need for high-level, declarative debugging mechanisms in functional and logic languages. He highlights that the separation of concerns between correct declaration and execution, and its corresponding benefits, are

lost when utilizing imperative debugging techniques. Developers are faced with understanding complex execution models, which are not closely related to their program source. Further, the execution can be, by requirement, interleaved and in some language implementations, parallelized, making analysis of the procedural steps extremely difficult.

## 2.2 Related Work

There is minimal high-level debugging research specific to model transformations; however, declarative transformations share many traits with other declarative languages. Of particular relevance are pattern and logic-based languages, such as Prolog [11] and Mercury [12]. Debugging research related to these languages introduces a number of potential high-level approaches, such as algorithm [13–16] or deductive [17, 18] debugging as well as program slicing [19–21]. In earlier work [22] we adapted some of these techniques to apply specifically to model transformations.

A key issue in applying any of these debugging techniques in practice is the detailed interactions required to capture the developer’s intent. In the case of general purpose languages, with infinite usage scenarios, the problem is extremely difficult<sup>3</sup>. However, domain specific languages, in this case the domain of model transformations, provide a unique opportunity. The constrained domain allows a set of general debugging questions and corresponding bug types to be extracted [22]. The advantage of such a set of questions is that they can be better understood, referenced and manipulated by developers to express their debugging goals and intent in high-level terms. The most common debugging questions asked of model transformations relate to the quantity of output; why are there so many, why are there not enough, why are there no, objects of type  $T$  in the target?. These questions will be investigated further in Section 5.

## 3 Context for Debugging Declarative Model Transformations

To assist in addressing the debugging questions, model transformations provide a rich, self-descriptive, environment that can be mined for data relevant to bug localization. Commonly available data includes:

- the transformation itself, in the case of this research we are particularly interested in transformations that are described by a meta-model and hence can be easily inspected and reasoned about
- the participating models and meta-models, divided into a source and target extent

---

<sup>3</sup> It is possible to extract generic bug patterns for general purpose languages [23]; however, unlike the domain specific case, it is not practical to define the complete representative set of bugs for all uses of a language, nor is the set of patterns constrained enough that all patterns will be relevant to the majority of programs.

- *data-driven*, or *model-transformation*, trace which links the source, target and transformation
- *execution-driven* trace which maps the procedural steps used to derive the target models from the source models

This paper shall refer to this collective data as the *debugging environment*. The following sections inspect the trace aspects of the debugging environment in more detail.

### 3.1 Data Trace

Data trace, also referred to as a record of transformation [24] or, in a model-transformation specific context, as simply *trace* [7], provides a link between source and target objects together with the relationship(s) that supports their creation. Data trace has primarily evolved as a mechanism to meet the requirements of incremental transformation [7] and round-trip engineering [25]; however, as originally identified by Gerber et al. [26], the relationship information provided by data trace can be used to enable model-transformation specific debugging techniques.

There exists no common formal definition of a data trace model, each tool implements its own variety, with its own level of detail and persistence approach, stored separately or as a part of the target output [4]. However, the QVT specification [7] does outline a number of core requirements for data trace in their Relations transformation language, in particular the requirements to link target objects to the relationships which contributed to its creation as well as the source and ephemeral objects which participated in the realization of the relationship. Similarly, Tefkat defines a trace model which shares these characteristics.

An additional note about data trace, is the concept of implicit trace versus explicit trace. Both QVT and Tefkat have the concepts of implicit and explicit trace. Implicit trace is the compulsory links automatically defined between the target object and its contributors. Explicit trace is a mechanism by which additional output, not in the target model, can be output. Explicit trace objects can also be used as intermediate objects to assist in incrementally building the target model. When discussing data trace, we are referring to the implicit form that can always be made available without modification to the transformation. However, in lieu of support for appropriate implicit trace, tools that only support explicit trace could provide the same data (with modification to the transformation).

### 3.2 Execution Trace

Model transformation trace discussions are generally centered around data trace; however transformations can also produce execution trace. Execution trace maps the actual steps, in an imperative sense, required to perform a transformation. This form of trace is analogous to traditional program tracing [27] used to debug imperative languages.

A key difference to data trace is that execution trace can be accessed in a live (at runtime) manner or in a post-hoc manner by persisting the execution trace. In contrast, data trace is not reliably accessible, due to its mutability, until the transformation is complete. To simplify discussions, the debugging techniques presented in this paper are not concerned with how the execution trace is made available, only that it is. The decision as to which method is more appropriate is left to be made with the knowledge of the time and space trade-offs required for specific tools.

There are two obvious issues with utilizing execution trace for debugging. Firstly, execution trace is very specific to the type of transformation engine and its implementation. Secondly, as established in Section 2.1 it is not appropriate to expose the developer to the details of the execution model. To address this, Section 4 looks at aspects of the execution model, together with the remainder of the debugging environment, that can be exposed in a high-level manner suitable for human interaction.

## 4 Summarizing the Debugging Environment

To approach high-level debugging it is critical to understand what information is important to the user without exposing the unnecessary complexities of the raw data, whose volume and verbosity prevents efficient use. Critical pieces of debugging information become lost in a noise of irrelevant data (for the debugging case). As noted previously [22], a number of debugging questions, in particular the set of *analysis* questions, can be addressed by narrowing the focus of the developer by refining or slicing the data available to a manageable subset.

There are two aspects in addressing this problem, how the raw data can be summarized, which is discussed in this section, and how the summarized data can be visualised, which is discussed in Section 5.

### 4.1 Summarizing Data Trace

Data trace is itself a high-level concept concerned with the relationships between the data and not how they were realized. However, the sheer volume of data trace makes it difficult to work with. It is also problematic for developers to analyze this as an external model, as is normally done with manual debugging approaches. A large context switch occurs between working with the transformation and analyzing the data trace. To address this, the data trace can be summarized from alternate views, where the trace information is described with respect to either the target models, the source models, or the transformation.

**Target View** The debugging story starts with the target view. From the target view a developer is able to verify that their expectations were, or were not, met. This can be facilitated by summary of the target data to help identify bug patterns, and provide hints as to the commonality between the “buggy” target objects. This is achieved by grouping target objects, based upon either

the objects or rules which contributed to their creation. These groupings allow for easier identification of emergent bug patterns; for example, all objects that are missing an expected attribute were created by the same rule.

Using the target view, the developer is able to then refine information on the related source and target views by identifying objects either by their class or attribute values.

**Source View** Links to both the target and transformation can contribute to providing information about the source models.

The data-trace can be used to identify, for each object, whether it contributed to the creation of a final target object, contributed to the creation of some intermittent object, or did not contribute to the output at all.

Static analysis of the transformation can be used to identify which objects have the *potential* to contribute to the output. The delta between those objects which are referenced by the transformation and those that actually contributed to the transformation provides an important insight into potential bugs in the transformation.

**Transformation View** The transformation's core purpose is the creation of a target model; accordingly the most relevant summary data for the transformation is the distribution of object creation. For each rule, for each type of object, the creation data can be summarized as the number of objects that a rule assisted in creating out of the total number of objects for that type. Section 5.3 deals with an example of this technique in more detail.

To be more useful to the developer, it is possible to slice this information based on object type (using class or attribute based equality). When looking at the creation distribution for a single type of object it becomes possible to differentiate between those rules which created 0% of the objects but potentially could of created target object of the desired type and those rules which could not have created the object type.

One important point to note is that for the entire transformation, it is possible (even likely), that the sum of the creation distributions will be greater than 100%, as multiple rules may contribute to the existence of a single target object.

Another important facet is the transformation coverage for a given set of source models. This can be determined by looking at the creation distribution without division based upon the type of object. This information can be used to easily identify which parts of a transformation are not even being evaluated.

## 4.2 Summarizing Execution Trace

As noted in Section 3.2 there is important information to be gathered from the execution trace, however this information must be summarized in a manner that is independent of the transformation engine implementation and isolates the developer from the complexities of the execution model.

When writing a transformation, terms can be described as navigational, those responsible for binding variables, and filters, those which are responsible for selecting a subset of the input for transformation. It is expected that, in general, navigational terms would succeed most (if not all) of the time, and filters would succeed sometimes (but rarely always or never). The execution trace can reveal what actually happened and if the expectations of navigational and filtering terms were met. To summarize the execution trace with respect to the transformation, each term is annotated as one of, always succeeded, always failed, or succeeded a percentage of the time. Section 5.4 demonstrates how this data can be used to provide visual feedback to the developer in order to help localize a bug.

Another problem which often causes issues for developers is the execution order of declarative transformations. Although the declarative paradigm alleviates the concern of execution order from the developer, that is there is no explicit ordering, declarative transformations still imply a partial evaluation order. Partial ordering occurs at two levels; dependencies between rules which imposes stratification, where each strata contains an exclusive sub-set of the transformation rules and is dependent on the (complete) execution of the previous strata; and, dependencies between terms (even in different rules) which imposes ordering within a strata.

The execution trace is able to provide this partial ordering information to assist the developer's understanding of the transformation. For textually represented transformations, such as Tefkat, this information can be communicate to the developer by visually re-order rules within the transformation and terms within a rule to highlight the dependencies. Alternatively, graphical approaches, such as those often used with triple graph grammars [28], could better communicate this dependency information as implicit edges on the graph.

## 5 Visualising the Model Transformation in Execution

Debugging is an inherently interactive process, and how the available information can be presented to the developer is critical to the success of finding the bug. To achieve this, the following section presents user interface (UI) paradigms and techniques to assist in effectively engaging the developer and enabling communication of the debugging questions. Due to the concrete nature of UIs, we utilise the Eclipse Modelling Framework (EMF) [29, 30] and the Tefkat transformation engine [31, 2] to demonstrate these approaches; however, it is important to point out that the techniques presented could be adapted to other declarative transformation tools.

Note that visualisation techniques for all summary data discussed in Section 4 can not be demonstrated due to space constraints; however, the working example will step through the primary techniques to track down a common bug that often occurs in real world transformations.

## 5.1 Working Example

A running example will be used throughout this section. The example, Figure 1, utilizes two small meta-models; a Student meta-model containing a single class representing a **Student** with attributes of name, kind and gpa; and, a Report meta-model containing a single class representing a **Report**, with attributes of name and average. Transformation rules will be introduced as required to discuss specific techniques. Although the example is small, it is sufficient to demonstrate the debugging techniques and is advantageous as the problems can be discussed without being concerned with explaining a complex meta-model context.



Fig. 1. Working Example.

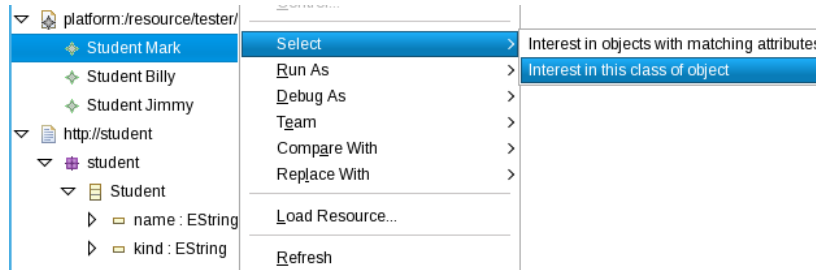
## 5.2 Dealing with Volume

We have identified two primary selection approaches useful to narrowing the focus of the debugging questions.

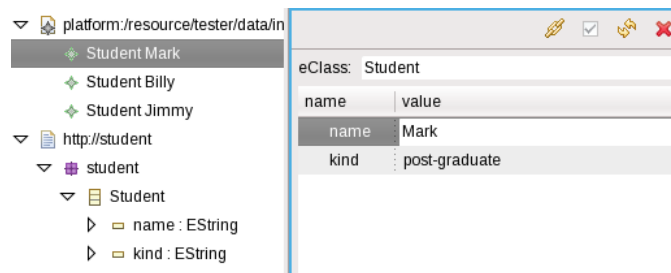
The first selection approach is class based. The UI has been extended to allow any EMF Ecore object to be selected through a right click action. This is achieved by leveraging the Eclipse platforms object contribution model. Figure 2 (a) demonstrates a user registering interest in the class of an object.

The second selection approach is attribute based. An input form allows the user to enter attributes of interest to the debugging question. To assist in this process an existing object, from the source, trace or target models can be selected as a template, pre-populating the form with its attribute values. Figure 2 (b) shows the attribute selection view. Once the user has identified the types of objects that are interesting, either through the object's class or attributes, it is possible to narrow the debugging environment to focus on this selection. It is also noted that not all objects are easily identified by their attributes making this form of selection difficult; however, in practice, the non-identifiable objects are generally navigable from some identifiable object.

The next step in dealing with volume is being able to focus on only the currently relevant data. Folding, a technique of hiding code or data details in an



(a) Class based selection.



(b) Attribute based selection.

**Fig. 2.** Object selection mechanisms.

editor, provides an effective UI analog to slicing techniques. User attention is focused by removing (initially) irrelevant items through folding, and then allowing navigation to related objects by progressively unfolding the results. Folding is achieved by exploiting the underlying tree structure of the transformation and models, where closing a fold is a matter of replacing the children of a branch in the tree with a smaller summarized heading.

Code folding can be exploited to visually represent program slices. Appropriate slicing can occur by allowing the user to pose analysis questions of the debugging environment. Using the running example, the developer is executing a large transformation where there is an issue with the output, in this case too many **Report** objects have been created. Given this, the developer is able to ask the question, “what parts of my transformation contributed to the creation of **Report** instances?”. Static analysis of the transformation is used to determine rules that are capable of creating **Report** objects. Once identified, all unrelated rules can be automatically folded, leaving only rules of interest to the debugging context. Figure 3 compares a folded and unfolded slice of the transformation.

Using these techniques for object selection and visual program slicing enables the user to efficiently ask debugging questions and importantly allows the output be refined to the relevant debugging context. The next section examines the visualisation of data-driven trace to help address more debugging questions.

```

16# RULE ReallyComLexAndIrrelevantRule
17   FORALL Student left, Student right
18   WHERE left.kind = right.kind
19   AND NOT left.name = right.name
20   LINKING RelatedPairsOfStudents
21   WITH primary = left,
22        secondary = right
23 ;
24
25# RULE UnderGradStudentsAverages
26   FORALL Student s
27   WHERE s.kind = "undergrad-student"
28   MAKE Report r
29   SET r.name = s.name,
30       r.average = s.gpa
31 ;
32
33# RULE AnotherReallyComLexAndIrrelevantRule
34   FORALL Student left, Student right
35   WHERE left.kind = right.kind
36   AND NOT left.name = right.name
37   LINKING RelatedPairsOfStudents
38   WITH primary = left,
39        secondary = right
40 ;
41
16# RULE ReallyComLexAndIrrelevantRule
24
25# RULE UnderGradStudentsAverages
26   FORALL Student s
27   WHERE s.kind = "undergrad-student"
28   MAKE Report r
29   SET r.name = s.name,
30       r.average = s.gpa
31 ;
32
33# RULE AnotherReallyComLexAndIrrelevantRule
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55

```

(a) Transformation in unfolded state.      (b) Transformation in folded state.

Fig. 3. Comparison of transformation in unfolded (a) and folded (b) states.

### 5.3 Visualising Data Trace

When writing transformations, the developer knows that some rules embody the “general case”, while other rules are to deal with highly specific situations. Accordingly, the general case rules are expected to create most target objects with the remaining rules responsible for a smaller number of special cases.

Figure 4 shows a set of example transformation rules attempting to build a **Report** for each **Student**. In this case the general case is captured by the rule **StudentsWithGPA**, where it is expected that most students would have a GPA. and the special case is captured by **StudentsWithoutGPA**, where it is possible that some student may not yet have a GPA. After executing these rules, the developer has identified that there are less **Report** instances than **Student** instances, where it was expected that there would be a **Report** for every **Student**. This discrepancy leads the developer to pose the question “why are there not enough **Report**’s in the target model?”.

To address this question, data trace is used to identify the breakdown of how the the buggy target objects were created. In a larger transformation, focus would be narrowed to the relevant rules using techniques described in Section 5.2, for this case we shall just work with the two example rules.

Section 4.1 provides insight into how the data trace can be summarized into more consumable formats, in the case we want to summarize object creation for the specified type, **Report**, with respect to each transformation rule. Figure 5 shows that the general case rule, **StudentsWithGPA** has created all of the **Report** objects. The visual cue provided quickly brings the developers focus to the special case rule which is producing no objects. This does not match the developers expectations, where even though the rule only handles special cases, it is expected that it would support creation of at least *some* of the target model. The rule **StudentsWithoutGPA** requires closer inspection.

```

RULE StudentsWithGPA
  FORALL Student student
  WHERE (student.kind = "under-graduate"
        OR student.kind = "post-graduate")
        AND average = student.gpa
        AND name = student.name
  MAKE Report report
  SET report.name = name,
      report.average = average;

RULE StudentsWithoutGPA
  FORALL Student student
  WHERE student.kind = "future-student"
        AND name = student.name
        AND average = student.gpa
  MAKE Report report
  SET report.name = name,
      report.average =
        append("not-yet-commenced: ", average);

```

Fig. 4. Example transformation rules which produce buggy output.

```

5
6 i 6= RULE StudentsWithGPA Supports 100.0% of Report instances
7   FORALL Student student
8   WHERE (student.kind = "under-graduate"
9         OR student.kind = "post-graduate")
10          AND average = student.gpa
11          AND name = student.name
12   MAKE Report report
13   SET report.name = name,
14       report.average = average;
15
16 i 16= RULE StudentsWithoutGPA Supports 0.0% of Report instances
17   FORALL Student student
18   WHERE (student.kind = "future-student")
19          AND name = student.name
20          AND average = student.gpa
21   MAKE Report report
22   SET report.name = name,
23       report.average =
24         append("not-yet-commenced: ", average);
25

```

Fig. 5. Transformation annotated with creation distribution information.

The debugging question can now be refined to concentrate on a single rule and type of object; “why are there no `Report` instances created by the rule `StudentsWithoutGPA`?”. This refined debugging question presents some interesting challenges. There is no data-driven trace available for non-existent objects; so we must now turn to the execution trace for a more detailed understanding of why this rule is failing.

#### 5.4 Visualising Execution Trace

We observe that non-existent target objects are invariably linked to source patterns not matching what we think they match, that is, the discrepancy between intent and execution. This in turn relates to one or more terms in the transformation rule not doing what we think they do. The consistent failure of a term in a rule expected to produce objects, in this example instances of `Report`, is almost invariably the immediate cause of the object shortage.

```

RULE StudentsWithoutGPA
  FORALL Student student
  WHERE student.kind = "future-student"
        AND name = student.name
        AND average = student.gpa
  MAKE Report report
  SET report.name = name,
      report.average =
        append("not-yet-commenced: ", average);

```

(a)

```

RULE StudentsWithoutGPA
  FORALL Student student
  WHERE student.kind = "future-student"
        AND name = student.name
        AND average = student.gpa
  MAKE Report report
  SET report.name = name,
      report.average =
        append("not-yet-commenced: ", average);

```

(b)

**Fig. 6.** Example transformation annotated with (a) intent (in the mind of the developer) and (b) actual execution.

Execution trace can reveal to us which terms are consistently succeeding, failing, or doing something in-between. The use of colour to decorate the terms

based on their success/failure in the execution trace draws the eye almost instantly to the problem terms. In particular the use of red colouring to indicate always failing terms draws the likely cause of the bug immediately to the developers attention.

Figure 6 (a) shows the buggy transformation rule annotated with the developers intentions<sup>4</sup>. It is clear that the term “`student.kind = "future-student"`” is intended to act as a filter, selecting only objects relevant to this special case rule, and as such it is only expected to succeed *some* of the time. The terms “`name = student.name`” and “`average = student.gpa`” are intended to be navigational only, and are expected to successfully bind the variables every time.

```

RULE StudentsWithoutGPA
  FORALL Student student
  WHERE student.kind = "future-student"
    AND name = student.name
    AND average = student.gpa
    OR average = "undefined"
  MAKE Report report
  SET report.name = name,
    report.average =
      append("not-yet-commenced: ", average);

```

**Fig. 7.** Corrected version of buggy transformation rule.

Now compare this to the actual execution shown in Figure 6 (b). Attention is immediately brought to the always failing term “`average = student.gpa`”. This violates the developer’s expectation that it was a navigational term and should always succeed. Failure of the navigational term indicates that an underlying assumption that all students would have some value for their GPA is incorrect. Figure 7 shows the rule corrected to handle students with an undefined GPA.

## 6 Conclusions and Future Work

We have shown that practical high-level debugging techniques can be used with declarative transformation languages, shielding the developer from a detailed understanding of the underlying transformation engine’s execution strategies.

We demonstrated new debugging techniques that exploit data and execution trace to visualize the logical transformation execution in a “debugging question” dependent manner. Allowing the debugging context to be driven through well

<sup>4</sup> Due to the limitations of black and white printing, the example uses plain, underlined, italicized, and struck-through text to represent the states of did not run, always succeeded, sometimes succeeded and never succeeded, respectively. The real graphical editor, uses plain, green, blue, and red backing colours to highlight the same states with greater affect.

understood domain-specific debugging questions, our approaches were able to deal with the volumes of data that can be produced by a model transformation and present relevant execution summaries in a manner easily understood by developers. Using the visualised execution summary, developers are able to verify, from a high-level, that the intent of the transformation and any underlying assumptions were met.

With the developer able to verify their expectations, the next step is to capture this information for use in testing and automated bug localization. One proposed approach to be investigated in the future, is the ability to capture success rates and object creation over-time resulting in a “typical execution history” of the rule and any variations of actual success rates to expected success rates can be quickly noticed.

Another approach to be considered, is the the ability for developers to annotate their intentions for terms. Similar to determinism which can be expressed by Mercury’s type system [32], the annotations would mark execution expectations for each rule. The classes of expectations would correspond to our term highlighting techniques, where a term would be expected to consistently succeed, fail or succeed only some of the time. For example, identifying navigational terms that should always succeed, or using fuzzy logic techniques to identify terms that succeed *most* or *some* of the time. These annotations could be used by the transformation engine to verify expectations at run-time.

## References

1. Geiger, L., Zündorf, A.: Graph Based Debugging with Fujaba. *Electronic Notes in Theoretical Computer Science* **72**(2) (2002) 112–112
2. Lawley, M., Steel, J.: Practical Declarative Model Transformation With Tefkat. *Lecture Notes In Computer Science* **3844** (2006) 139
3. 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)
4. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. *IBM Systems Journal* **45**(3) (2006) 622
5. Mens, T., Czarnecki, K., Van Gorp, P.: A Taxonomy of Model Transformation. *International Workshop on Graph and Model Transformation* (2005)
6. Mens, T., Van Gorp, P.: A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science* **152** (2006) 125–142
7. Specification, O.: MOF QVT Final Adopted Specification. *OMG Document ptc/2005-11-01*, November (2005)
8. Byrd, L.: Understanding the control flow of Prolog programs. *Proceedings of the 1980 Logic Programming Workshop* (1980) 127–138
9. Pain, H., Bundy, A.: What stories should we tell novice Prolog programmers. *Artificial Intelligence Programming Environments* (1987) 119–130
10. Naish, L.: A declarative debugging scheme. *Journal of Functional and Logic Programming* **3** (1997) 1997
11. Warren, D., Pereira, L., Pereira, F.: Prolog-the language and its implementation compared with Lisp. *ACM SIGPLAN Notices* **12**(8) (1977) 109–115

12. Henderson, F., Conway, T., Somogyi, Z.: Mercury, an efficient purely declarative logic programming language. Glenelg, Australia (1995) 499–512
13. 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
14. Kokai, G., Harmath, L., Gyimothy, T.: Algorithmic debugging and testing of Prolog programs. *Workshop on Logic Programming Environments* (1997) 14–21
15. Paakki, J., Gyimothy, T., Horvath, T.: Effective Algorithmic Debugging for Inductive Logic Programming. *Proc. of the Fourth International Workshop on Inductive Logic Programming (ILP-94)* Bad Honnef/Bonn Germany September (1994) 12–14
16. Shapiro, E.: *Algorithmic Program DeBugging*. MIT Press Cambridge, MA, USA (1983)
17. Naish, L.: *Declarative Debugging of Lazy Functional Programs*. Dept. of Computer Science, University of Melbourne (1992)
18. Naish, L.: *A Declarative Debugging Scheme*. Department of Computer Science, University of Melbourne (1995)
19. Weiser, M.: Program slicing. *Proceedings of the 5th international conference on Software engineering* (1981) 439–449
20. Weiser, M.: Programmers use slicing when debugging. *Communications of the ACM* **25**(7) (1982) 446–452
21. 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
22. Hibberd, M., Lawley, M., Raymond, K.: Forensic Debugging of Model Transformations. *Lecture Notes in Computer Science* **4735** (589–604) 589
23. Hovemeyer, D., Pugh, W.: Finding bugs is easy. *ACM SIGPLAN Notices* **39**(12) (2004) 92–106
24. Miller, J., Mukerji, J., et al.: *MDA Guide Version 1.0.1*. OMG Document omg/2003-06-01, June (2003)
25. Sendall, S., Küster, J.: *Taming Model Round-Trip Engineering*. *OOPSLA/GPCE: Best Practices for Model-Driven Software Development* (2004)
26. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The Missing Link of MDA. *Lecture Notes in Computer Science* **2505** (2002)
27. Larus, J.: Efficient program tracing. *Computer* **26**(5) (1993) 52–61
28. Grunske, L., Geiger, L., Lawley, M.: A Graphical Specification of Model Transformations with Triple Graph Grammars. *Model Driven Architecture Foundations and Applications (ECMDA)* **3748** (2005) 284–298
29. Eclipse Foundation: Eclipse Modeling Framework Project (EMF). Available at: <http://www.eclipse.org/modeling/emf/> (2007) accessed February 20th, 2007.
30. <http://www.eclipse.org>: The Eclipse Modeling Framework (EMF) Overview. Available on: <http://dev.eclipse.org/viewcvs/indextools.cgi/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.html> (2007) accessed February 20th, 2007.
31. Tefkat: Tefkat: The EMF Transformation Engine). Available at: <http://tefkat.sourceforge.net/> (2007) accessed February 20th, 2007.
32. Henderson, F., Somogyi, Z., Conway, T.: Determinism analysis in the Mercury compiler. *Proceedings of the Australian Computer Science Conference* **1** (1996) 337–346

## 13.4 initial sketch of trace paper

# Model, Trace, Debug.

Mark Hibberd<sup>1</sup>, Michael Lawley<sup>2</sup>, Kerry Raymond<sup>3</sup>

<sup>1</sup> Queensland University of Technology, Brisbane, Australia  
e-mail: [mt.hibberd@student.qut.edu.au](mailto:mt.hibberd@student.qut.edu.au)

<sup>2</sup> Australian E-Health Research Centre, CSIRO ICT Centre  
e-mail: [michael.lawley@csiro.au](mailto:michael.lawley@csiro.au)

<sup>3</sup> Queensland University of Technology, Brisbane, Australia  
e-mail: [k.raymond@qut.edu.au](mailto:k.raymond@qut.edu.au)

Received: date / Revised version: date

**Abstract** Model-driven engineering and model transformation approaches have embraced *traceability* as a core principle. This paper explores formal trace models in the context of debugging model transformations. We present two views of model traceability, relational or data trace and execution trace. These views are examined and linked to existing model transformation techniques.

## 1 Outstanding Issues

1. length????
  2. structure???
  3. how much effort should I expend talking generically about trace. and what sort of formalisation should I use.
    - i present trace meta-models for tefkat.  
terminology for execution meta-model  
terms-expressions ?? something else ?? . expressing explicit ordering . addressing iteration - is there a way to slip over this?  
Is it ok to say that i am dealing with an incomplete meta-model - just interested in the artifact w.r.t. debugging?
- ? should i have an example transformation - or discuss in the abstract - meta-models only, get more involved in the how/why wrt debugging.

## 2 Introduction

. a little background on trace . the debugging problem . why trace can be / is useful to debugging .. refs, forensic debugging, missing link papers

## 3 Views of Trace

- ? names – 1. view or something else???
2. data vs relational
    - . data / relational trace .. commonly accepted in model transformation circles, need to start building a case for the aspects suited to debugging. . executional trace .. not as much precedent for formalism here .. tie in with multiple language paradigms, can map concepts of expression evaluation, success/failure across the board - some abstraction, i.e. at the domain of model transformations .. should link/build to a meaty bit of the paper in section 6

## 4 Related Work

. related work .. there is a fair bit on formal data trace, not so much on a formal execution trace . need to draw a tight link between traceability and debugging .. use some of the older papers on program tracing, and informal trace techniques as a basis for this

## 5 Requirements

. where did i come from? / what contributed? . what the state when i was created? . import point: negative trace .. the ability to track the untrackable / things that drop out of the model / missing targets .. helped by execution trace . the ability to identify parts of a model, either things that are readily identifiable or navigable from something that is identifiable.

## 6 Concrete Example

. Tefkat and concrete example of trace meta-models . present the tefkat data trace (refs??? i am sure it has been published somewhere just can't see it at the moment) . introduce an execution trace . discuss import aspects and how/why they are important to debugging.

## 7 Other Concerns

Otheer Space trade off / practability

## 8 Conclusion

some text with citation [1].

## References

1. H. Pain and A. Bundy. What stories should we tell novice Prolog programmers. *Artificial Intelligence Programming Environments*, pages 119–130, 1987.

## Appendix A

# Person to Male-Female Transformation Listing

TODO: include listing from source file, rather than inline.

```
1 TRANSFORMATION unisex2gender : unisex -> gender
2
3 NAMESPACE http://unisex
4 NAMESPACE http://gender
5
6 CLASS Me2Person {
7     Person me;
8     BasePerson person;
9 };
10
11 RULE Male
12     FORALL Person p
13         WHERE p.sex = "m"
14         MAKE Male m
15         SET m.name = p.name ,
16             m.id = p.id
17     LINKING Me2Person
18         WITH me = p ,
19             person = m
20 ;
21
22 RULE Female
23     FORALL Person p
24         WHERE p.sex = "f"
25         MAKE Female f
26         SET f.name = p.name ,
27             f.id = p.id
28     LINKING Me2Person
29         WITH me = p ,
30             person = f
31 ;
32
33 RULE Spouses
34     WHERE Me2Person LINKS me = source , person = source\_person
```

```
35     AND Me2Person LINKS me = spouse, person = spouse\_person
36     AND source.spouse = spouse.id
37     SET source\_person.spouse = spouse\_person
38 ;
```

## Appendix B

# Male-Female to Person Transformation Listing

```
1 TRANSFORMATION gender2unisex : gender -> unisex
2
3 NAMESPACE http://gender
4 NAMESPACE http://unisex
5
6 RULE Male
7   FORALL Male m
8     WHERE IF m.spouse = spouse
9       THEN spouse_id = spouse.id
10      ELSE spouse_id = "n/a"
11     ENDIF
12   MAKE Person p
13     SET p.name = m.name,
14        p.sex = sex,
15        p.id = m.id,
16        p.spouse = spouse_id
17 ;
18
19 RULE Female
20   FORALL Female f
21     WHERE IF f.spouse = spouse
22       THEN spouse_id = spouse.id
23      ELSE spouse_id = "n/a"
24     ENDIF
25   MAKE Person p
26     SET p.name = f.name,
27        p.sex = sex,
28        p.id = f.id,
29        p.spouse = spouse_id
30 ;
```

## Appendix C

# Factored Male-Female to Person Transformation Listing

```
1  TRANSFORMATION gender2unisex : gender -> unisex
2
3  NAMESPACE http://gender
4  NAMESPACE http://unisex
5
6  RULE Male
7    FORALL Male m
8      WHERE spouseId(m, spouse_id)
9      MAKE person(m, "m", spouse_id)
10 ;
11
12 RULE Female
13   FORALL Female f
14     WHERE spouseId(f, spouse_id)
15     MAKE person(f, "f", spouse_id)
16 ;
17
18 PATTERN spouseId(mf, spouse_id)
19   WHERE IF mf.spouse = spouse
20         THEN spouse_id = spouse.id
21         ELSE spouse_id = "n/a"
22         ENDIF
23 ;
24
25 TEMPLATE person(mf, sex, spouse_id)
26   MAKE Person p FROM f(mf)
27   SET p.name    = mf.name ,
28       p.sex     = sex ,
29       p.id      = mf.id ,
30       p.spouse  = spouse_id
31 ;
```

## Appendix D

# Trace Enhancement Code

## Appendix E

# Forensic Debugging Code / Transformations

## Appendix F

# Problem Identification Transformation