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 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

1 Introduction **9**

2 Research Question **11**
- 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

3 Literature Review **18**
- 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

4 Declarative Model Transformations and Tefkat **31**
- 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 . . . 39
  - 4.2.4 Factored Male-Female to Person Transformation . . . 40
  - 4.2.5 Evaluation Order . . . 41
  - 4.2.6 FROM . . . 41

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 ~~Flat~~ Unisex meta-model, representing people in the running example. 34
4.3 Example input for ~~flat~~ unisex meta-model. 34
4.4 ~~Structured~~ Gender meta-model, representing people in the running example. 35
4.5 Example input for ~~structured~~ gender meta-model. 35

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 lifecycle
- 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 lifecycle
- 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

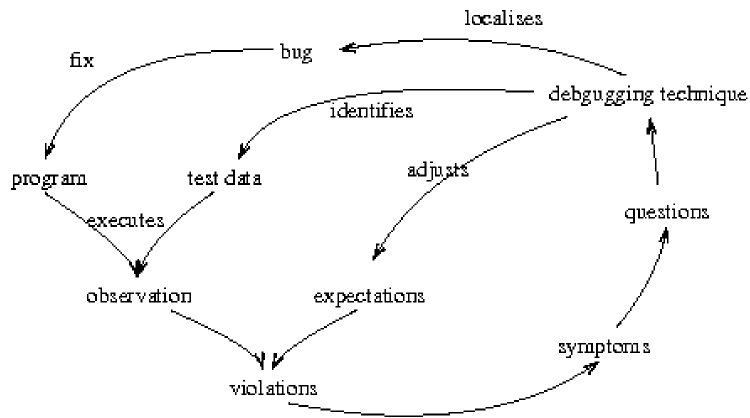

Figure 2.1: Bug lifecycle.

old, see 2.1 for update.

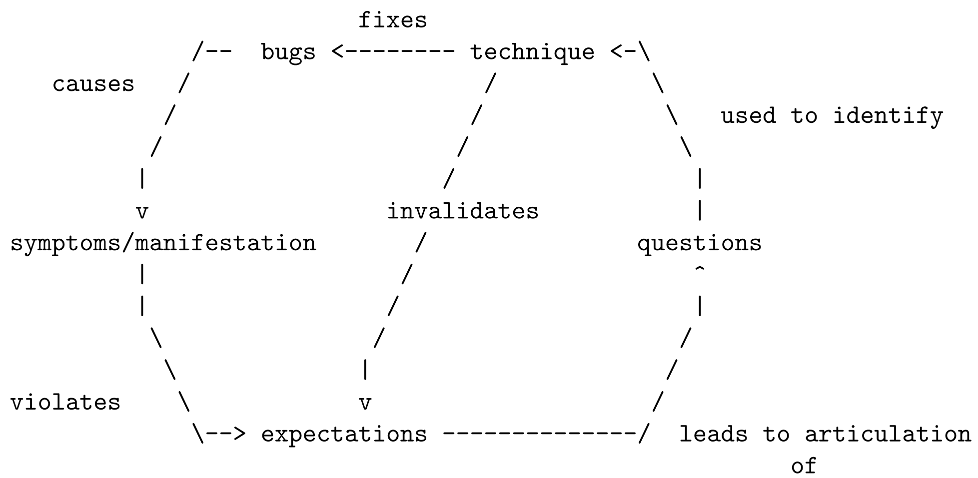

Figure 2.2: Old bug lifecycle.

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 expactations 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:

$$M_s \overset{T}{\mapsto} M_t \overset{V}{\mapsto} \{p_1, p_2, ..., p_n\}$$

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
$\{p_1, p_2, ..., p_n\}$	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 $\{p_1, p_2, ..., p_n\}$ 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:

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

Where:

p	is an identified problem
$\{p_1, p_2, ..., p_n\}$	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
∂M_s	is some part of the source model instance
∂M_t	is some part of the target model instance
∂T	is some part of the transformation
ΔM_s	is a change to the source model instance
ΔT	is a change to the transformation

For all problems, p, in the set of problems$\{p_1, p_2, ..., p_n\}$, the goal is to automatically identify the parts of the source instance, ∂M_s, transformation, ∂T, and target instance, ∂M_t, which describes the location, l, of the problem and a solution, s, describing the changes required to the source model, ΔM_s, and the transformation, ΔT. 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 ro 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 knowledge gained from the debugging process be fed back to reduce 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 debugging 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, together 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 questions from bug symptoms be leveraged to identify an optimal debugging technique or techniques for given violations?"**

> **"To what extent can declaritive focused debugging techniques assist in producing a debugging outcome, and can the general facets of declaritive programming or the specific 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 relevent material here (primarily debugging). Gradually expand and introduce new concepts/literature at the start of each chapter as its relevence 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 relavent. 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 short comings. 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 (MDSD), 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 lifecycle.

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 then 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 come 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 then 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 discusiion, 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 higly 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 correspoindingly 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

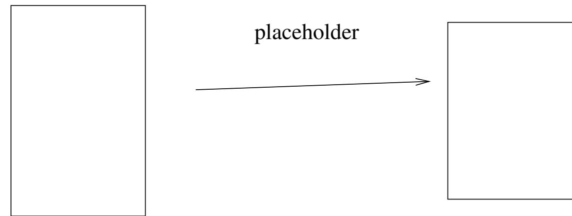

Figure 4.1: Trace Meta-model

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 leafs 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 ~~is intentionally very simple,~~ 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 ~~concisely express the debugging concepts in the following chapters~~ introduce debugging concepts simply and clearly, without concern for ~~having to understand~~ a more complex domain ~~.~~ ~~Once the concepts are clear, we~~ 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 **??** shall introduce a larger, more realistic ~~, example in chapter **??**~~ example to demonstrate the utility of the debugging techniques. ~~The example shall also serve to introduce~~ This larger example will cover a more complete set of Tefkat's ~~syntax~~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 ~~types of *bugs*~~*buggy* transformations.

4.2.1 Meta-Models

The meta-models represent different forms of the same information. This means that we can write total, ~~loss-less~~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:

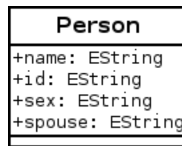

Figure 4.2: ~~Flat~~ Unisex meta-model, representing people in the running example.

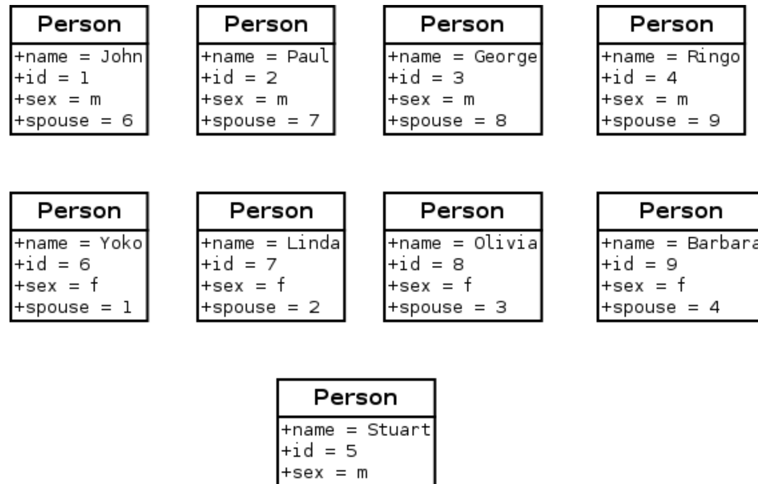

Figure 4.3: Example input for ~~flat~~ unisex meta-model.

- Each person has a name.
- 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 that a reference to the actual spouse. This model will be referred to as the ~~flat~~ unisex model, and is represented by the URI `http://`~~`person`~~`unisex`.

Figure 4.3 shows a valid input model for the ~~flat~~ 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 ~~flat~~ 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

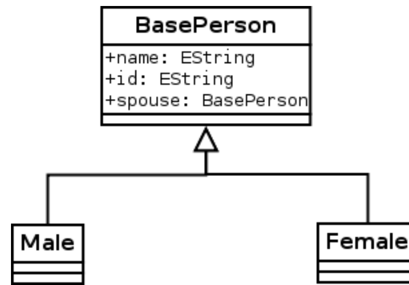

Figure 4.4: ~~Structured~~ Gender meta-model, representing people in the running example.

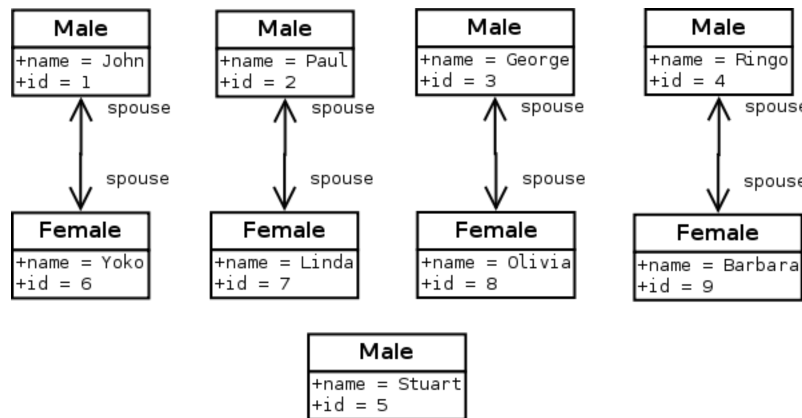

Figure 4.5: Example input for ~~structured~~ gender meta-model.

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 then by the data. This model will be referred to as the ~~structured~~ gender model, and is represented by the URI `http://~~mf~~gender`.

Figure 4.5 shows a valid input model for the ~~flat~~ 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 ~~structured~~ gender model as a source.

4.2.2 Person to Male-Female Transformation

The first transformation takes the ~~flat~~ unisex model and produces an equivalent ~~structured~~ 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 \DIFdelbegin \DIFdel{person2mf : person }\DIFdelend \DIFaddbegin \DI
2  }\DIFdelend \DIFaddbegin \DIFadd{gender
3  }\DIFaddend
4
5  NAMESPACE http://\DIFdelbegin \DIFdel{person
6  }\DIFdelend \DIFaddbegin \DIFadd{unisex
7  }\DIFaddend NAMESPACE http://\DIFdelbegin \DIFdel{mf
8  }\DIFdelend \DIFaddbegin \DIFadd{gender
9  }\DIFaddend
```

In this case we are declaring a transformation named ~~person2mf~~unisex2gender with an input model, ~~person~~unisex, and an output model, ~~mf~~gender. There are two meta-models imported, the input model identified by `http://`~~person~~unisex and the output model identified by `http://`~~mf~~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 ~~of~~ 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. ~~The `FORALL` selects all~~

From an imperative viewpoint, the `FORALL` clause selects elements of a particular type, ~~in this case~~ giving the *impression* of iteration. In this case we are selecting elements of the type `Person` and ~~binds the elements~~ 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 ⇔ {John, Paul, George, Ringo, Yoko, Linda, Olivia, Barbara, Stuart }

WHERE ~~clause is~~ clauses are constructed as a series of terms that ~~are evaluated for~~ 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". ~~Also, the~~ 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 ~~this~~ 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, Lind

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

~~Note that this discussion 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. For example it is possible to evaluate the second sub-term before the first. In this case, the left-hand expression~~ sex ⇔ {John.sex = m, Paul.sex = m, George.sex = m, Ringo.sex = m, Stuart.sex = m }

Because of the constraint between p.sex~~is unbound and the right is bound as a constant~~ and ~~"m"~~sex~~. 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.~~ , 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`. ~~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`:~~

~~RULE Male FORALL Person p WHERE p.sex = "m" MAKE Male m FROM f(p) SET m.name = p.name, m.id = p.id LINKING Me2Person WITH me = p, person = m ;~~

~~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.~~

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.

```
CLASS Me2Person {
  Person me;
  BasePerson person;
};
```

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.

```
RULE Female
  FORALL Person p
   WHERE p.sex = "f"
    MAKE Female f
     SET f.name = p.name,
         f.id = p.id
 LINKING Me2Person
    WITH me = p,
         person = f
;
```

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.

```
RULE Spouses
   WHERE Me2Person LINKS me = source, person = source_person
     AND Me2Person LINKS me = spouse, person = spouse_person
```

```
      AND source.spouse = spouse.id
      SET source_person.spouse = spouse_person
;
```

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 **??** 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 ~~structured~~ gender model and transforming them to `Person` elements in the ~~flattened~~ 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.

```
TRANSFORMATION \DIFdelbegin \DIFdel{mf2person : mf }\DIFdelend \DIFaddbegin \DIFadd
}\DIFdelend \DIFaddbegin \DIFadd{unisex
}\DIFaddend

NAMESPACE http://\DIFdelbegin \DIFdel{mf
}\DIFdelend \DIFaddbegin \DIFadd{gender
}\DIFaddend NAMESPACE http://\DIFdelbegin \DIFdel{person
}\DIFdelend \DIFaddbegin \DIFadd{unisex
}\DIFaddend
```

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.

```
RULE Male
  FORALL Male m
   WHERE IF m.spouse = spouse
         THEN spouse_id = spouse.id
         ELSE spouse_id = "n/a"
         ENDIF
    MAKE Person p
     SET p.name   = m.name,
         p.sex    = "m",
         p.id     = m.id,
         p.spouse = spouse_id
;

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 **??** 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, `PATTERN`s and `TEMPLATE`s, 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:

```
PATTERN spouseId(p, spouse_id)
  WHERE IF p.spouse = spouse
        THEN spouse_id = spouse.id
        ELSE spouse_id = "n/a"
        ENDIF
;
```

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:

```
TEMPLATE person(mf, sex, spouse_id)
    MAKE Person p FROM f(mf)
     SET p.name    = mf.name,
         p.sex     = sex,
         p.id      = mf.id,
         p.spouse = spouse_id
;
```

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

```
RULE Male
  FORALL Male m
   WHERE spouseId(m, spouse_id)
    MAKE person(m, "m", spouse_id)
;

RULE Female
  FORALL Female f
   WHERE spouseId(f, spouse_id)
    MAKE person(f, "f", spouse_id)
;
```

Appendix ?? 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  \DIFadd{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 ;
11 }
```

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.