mel - Model Extraction Language and Interpreter

by

Robert Hackman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

© Rob Hackman 2021
Author’s Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.
Statement of Contributions

This thesis includes material from one published work [22] that was written under the supervision of Dr. Joanne Atlee, who contributed with ideas, discussion, and editing. Exceptions to sole authorship of material are as follows:

Chapter 1 primarily consists of material published in [22], which I co-authored with Dr. Joanne Atlee, Alistair Finn Hackett, and Dr. Mike Godfrey. Most of the paper was written collaboratively.

Chapter 2 in part consists of material published from the same paper [22]. Sections 2.2 and 2.3 were heavily modified and extended from the material presented in the paper by myself for the purposes of inclusion in this thesis.
Abstract

There is a large body of research on extracting models from code-related artifacts to enable model-based analyses of large software systems. However, engineers do not always have access to the entire codebase of a system: some components may be procured from third-party suppliers based on a model specification or their code may be generated automatically from models. Additionally, the development of software systems does not produce only source code as its output. Modern large software systems have various artifacts relevant to them, such as software models, build scripts, test scripts, version control history data, and more. In order to produce a more complete view of a modern software system heterogeneous fact extraction of various artifacts is necessary - not just of source code.

This thesis introduces \texttt{mel}— a model extraction language and interpreter for extracting “facts” from models represented in XMI; these facts can be combined with facts extracted from other system components to form a lightweight model of an entire software system. We provide preliminary evidence that \texttt{mel} is sufficient to specify fact extraction from models that have very different XMI representations. We also show that it can be easier to use \texttt{mel} to create a fact extractor for a particular model representation than to develop a specialized fact extractor for the model from scratch.
Acknowledgements

There are many people without whom this thesis would likely never have come to completion.

First and foremost I must express my immeasurable gratitude towards my supervisor Dr. Joanne Atlee without whose guidance and seemingly unending kindness this thesis would never have been possible. Over the course of my studies Dr. Atlee has taught me an unenumerable amount about research, academics, and the field of computer science. Dr. Atlee also provided me with incredible opportunities which both enriched and enabled my academic career thus far. Finally, and most importantly, I must thank Dr. Atlee for her unwavering support and belief in my ability to complete this work even when I may not have believed myself.

Thank you to Dr. Michael Godfrey for his invaluable advice, thoughts, and insight into various academic endeavours related to and including this thesis, and, of course, for his sense of humour.

Thank you to Dr. Ian Davis who helped guide and enable my early academic exploits, even if they did not enter into this thesis. Dr. Davis’ warmth and kindness also helped to make me feel welcome and at home in an academic environment.

Thank you to Dr. Nancy Day for agreeing to read and evaluate this thesis — I hope it will be as painless an experience as possible.

Thank you to those fellow graduate students whose knowledge and insightful conversations have led me to numerous realizations I would not have otherwise had. Thank you Bryan, Rafael T., Finn, Davood, Sandy, Rafael O., and Parsa.

Thank you to my dear friends Ten Bradley and Dr. Brad Lushman for too much over this (exceedingly) long adventure. Thank you for your friendship, support, and much needed nights of relaxation and fun.

Lastly, thank you to my family for the love and the laughs over the years. Thank you Steven and Stephanie, Leah and James, and Marvin and Pina.
Dedication

This thesis is dedicated to those whom I have missed more than you may ever know these past years:

Eric, Erik, Stefan, Connor, Mitchell, John, Nic, Martin, Tim, Jonas, Andy, Alex, Ashley, Devyn, Kathleen, Sam, and Megan.
# Table of Contents

List of Figures x

List of Tables xviii

1 Introduction 1
   1.1 Thesis Contributions ...................................... 2
   1.2 Thesis Organization ....................................... 3

2 Background 4
   2.1 Fact Extraction ........................................... 4
      2.1.1 Extraction of Lightweight Models ..................... 5
      2.1.2 Heterogeneous Fact Extraction ......................... 6
   2.2 XML and XMI .............................................. 7
      2.2.1 Differences between XML and XMI ....................... 9
      2.2.2 EMF Core (ECore) .................................... 9
   2.3 XML Analysis & Transformation Tools ....................... 11
      2.3.1 XPath and XQuery .................................. 11
      2.3.2 Academic Developed XML Tools ....................... 16

3 Model Extractor Language (mel) 18
   3.1 Requirements of mel ..................................... 18
3.2 Usage of mel ................................................. 20
  3.2.1 Declaration Reference Clauses ...................... 25
  3.2.2 Constraint Clauses .................................. 27
  3.2.3 Compound Productive Clauses ...................... 28
  3.2.4 Optional Clauses .................................. 32
  3.2.5 XM* Non-Basic Identifiers and String Literals ....... 33
  3.2.6 Contents of an XM* Node ............................ 36
  3.2.7 Extracting a Substring of an XM* Attribute ......... 36
  3.2.8 Constraints on XM* Attributes ...................... 39
  3.2.9 The Ternary Operator ............................... 41
  3.2.10 XM* Trees of Elements .............................. 43
  3.2.11 Elide Results of Intermediate Rules ............... 46

3.3 Grammar of mel ............................................. 48
3.4 mint the mel Interpreter .................................. 48
  3.4.1 Usage of mint ........................................ 48

4 Evaluation of mel ............................................. 50
  4.1 Scope of Applicability of mel ............................ 51
    4.1.1 UML StateMachine Diagrams & Statechart Diagrams .... 52
    4.1.2 UML Class Diagrams .................................. 61
    4.1.3 Simulink Block Diagrams .............................. 68
    4.1.4 Arcadia Logical Architecture Diagrams ............... 71
    4.1.5 Feature Models ...................................... 77
  4.2 Ease of Using mel ........................................ 82
    4.2.1 mel Comparison to Custom Extractor .................. 82
    4.2.2 Testing the Development Time of mel Programs ....... 83
    4.2.3 Succinctness of mel .................................. 86
    4.2.4 Development Time of mel Extractors ................. 89
5 Conclusions

5.1 Contributions ......................................................... 93
5.2 Threats and Limitations ............................................. 93
5.3 Future Work .......................................................... 94

References ............................................................. 95

A Extra mel Programs .................................................. 102
   A.1 mel-Generated Extractors for UML Class diagrams .............. 102
   A.2 mel-Generated Extractor for Feature Models ..................... 108
   A.3 mel and XQuery programs written by Subject A .................. 109
   A.4 mel and XQuery programs written by Subject B .................. 119

B Diagrams of Models Used in Studies .............................. 131
   B.1 UML StateMachine Diagrams .................................... 132
   B.2 UML Class Diagrams .............................................. 141
      B.2.1 ArgoUML Class Diagrams .................................... 141
      B.2.2 Eclipse Modelling Tools UML Class Diagrams .............. 144
      B.2.3 MagicDraw UML Class Diagrams ............................. 147
      B.2.4 UMLDesigner UML Class Diagrams ......................... 151
   B.3 Simulink Block Diagrams ........................................ 154
   B.4 Arcadia Diagrams ............................................... 160
   B.5 Feature Models .................................................. 164
List of Figures

2.1 Architecture of a fact-extraction tool pipeline .......................... 6
2.2 A simple Model expressed in XMI. ........................................ 8
2.3 A simple Model expressed in XMI representing relational data using unique identifiers. .................................................. 8
2.4 A simple UML class diagram created with the EMF Core Diagram Editor and a Snippet of the corresponding ECore representation. This shows the entirety of the representation of the two classes and their relationships. The eType and eOpposite attributes demonstrate how ECore encodes references to other data in the Model. (Produced with Eclipse Modeling Tools version 4.19.0). ........................................................... 10
2.5 An XML model with groups of siblings .................................. 12
2.6 Five simple XPath queries, one per line ................................. 12
2.7 XQuery program for generating a sibling relationship ............ 14
2.8 The result of applying the XQuery program in Figure 2.7 to the XML fragment in Figure 2.5. .............................................. 15
3.1 A simple Model expressed in XML. ........................................ 21
3.2 A sample mel program. ....................................................... 21
3.3 A mel program that demonstrates the use and semantics of declaration reference clauses. .................................................. 25
3.4 A Model of a library of books with a variety of authors and genres. ... 28
3.5 A mel program demonstrating the use of constraint clauses. ........ 28
3.6 An XML fragment with several Entity tags that do not each contain all the attributes other Entity tags do. .......................... 29
3.7 A mel program that demonstrates the behaviour of compound clauses.

3.8 The disconnected graph formed by the productive clauses \( A\{id:X, \ att:Y\}, B\{id:Y\}, C\{id:P, \ att:Q\}, D\{id:Q\} \), where each clause is given a vertex, and edges are drawn between vertices whose productive clauses share a bound identifier. The vertices have been labeled with the XM* tag of their corresponding clause, and the edges have been labeled with the bound identifier shared between the productive clauses.

3.9 A Model expressed in XML where Attributes are sometimes present. An owningTeam tag may exist associated with any given component tag, but is not required.

3.10 A mel program using an optional clause, specified by a prefixed @ symbol.

3.11 A simple XML fragment with entity tags with complex Attribute names (e.g., e:type) in XML.

3.12 A toy mel program illustrating the behaviour of double-quote and single-quote enclosed strings.

3.13 A sample XML node that contains string data.

3.14 A simple mel program that makes use of the mel.contents pseudo-Attribute for accessing the string contents of an XM* node.

3.15 A sample XML fragment that shows reference Attributes that have the referenced identifier prefixed with a # character.

3.16 A mel program that makes use of Attribute modifiers to specify a substring of an XM* Attribute to extract.

3.17 A transformation of the mel program shown in Figure 3.5 which includes the same restrictions on result sets without extracting the author and genre as attributes in the result set.

3.18 An alternative to the mel program shown in Figure 3.2 that uses a requirement in the feature declaration to specify that matching compFeatNode XM* nodes must have a userName Attribute.

3.19 A Model representing animals, optionally containing the sound that animal makes as an Attribute.

3.20 A mel program that demonstrates the use of the ternary operator.

3.21 A mel program that demonstrates the use of nested ternary operators.
3.22 An XML fragment that encodes relevant Model information as parental relationships between nodes. ........................................... 43

3.23 Example use of the mel parent operator. ......................................................... 44

3.24 An XML fragment that acts as an exemplar for nested relationships. .......... 45

3.25 Grammar for mel programs: Non-terminal symbols are italicized and TERMINAL symbols are in uppercase. Literals are enclosed in single quotes. "[" denotes alternation, "]...[" encloses optional symbols, and "("...")" encloses a grouping of symbols. "+" denotes zero or more repetitions of the previous symbol or grouping, and "*" denotes one or more repetitions of the previous symbol or grouping. ......................................................... 47

4.1 UML StateMachine diagram of an embedded home alarm system (Rhapsody). 53

4.2 UML StateMachine diagram of Harel’s statechart for a digital watch [23], produced using YAKINDU SCT. ................................. 54

4.3 Snippet of the XMI representation of the Rhapsody StateMachine Model shown in Figure 4.1. Identifiers have been abbreviated and non-referenced Attributes have been omitted. ......................................................... 55

4.4 Snippet of the XMI representation of the YAKINDU StateMachine Model shown in Figure 4.2. Identifiers have been abbreviated, non-referenced Attributes have been omitted, and white space has been adjusted to improve readability. YAKINDU stores all of a transition or state’s triggers, guards, and actions as one contiguous string. ......................................................... 56

4.5 mel program for extracting facts from a UML StateMachine diagram created with Rhapsody. ......................................................... 58

4.6 mel program for extracting facts from a UML StateMachine diagram created with YAKINDU. ......................................................... 59

4.7 UML Class Model of a travel agency (UMLDesigner). ................................. 63

4.8 UML Class Model of a loyalty program of a fictional company. The Model is a recreation, using the Eclipse Modelling Tools, of the Royal and Loyal example from Warmer and Kleppe’s book on the Object Constraint Language [68]. ......................................................... 64

4.9 Snippet of the XMI representation of the UMLDesigner Class diagram shown in Figure 4.7, representing an association relationship. Identifiers have been abbreviated and non-referenced Attributes have been omitted. ........ 65
4.10 Snippet of the ECore representation of the Eclipse Modelling Tools Class diagram shown in Figure 4.8. Identifiers have been abbreviated and non-referenced Attributes have been omitted. ........................... 65

4.11 mel declaration for extracting association facts from a UML Class diagram generated with UMLDesigner. ................................................................. 66

4.12 The mel declaration for extracting association facts from a UML Class diagram generated with the Eclipse Modelling Tools. ................................. 66

4.13 A system of a Simulink Model for a longitudinal flight control algorithm. The blocks with titles beneath them are subsystems for which an entire sub-Model is defined. .................................................. 71

4.14 Snippet of the XML representation of the Simulink Block diagram shown in Figure 4.13. Non-referenced Attributes have been omitted. .................... 72

4.15 mel extractor for Simulink Block diagrams. ........................................... 72

4.16 Arcadia Logical Architecture diagram for an in-flight entertainment system (Capella). ................................................................. 74

4.17 XML Fragment of the Model shown in Figure 4.16. Type Attributes have had their values truncated to fit the page. .......................... 75

4.18 mel program for extracting facts from an Arcadia Logical Architecture diagram. ................................................................................. 76

4.19 Feature Model diagram for an elevator system (FeatureIDE). ............... 78

4.20 Snippet of the XML representation of the Feature Model diagram shown in Figure 4.19. Non-referenced Attributes have been omitted. .............. 79

4.21 Snippet of the XML representation of cross-tree constraints written in FeatureIDE. ................................................................. 80

4.22 mel declarations for extracting children of and nodes in a FeatureModel. 80

A.1 mel program for extracting facts from UML Class diagrams generated with the MagicDraw tool. ..................................................... 104

A.2 mel program for extracting facts from UML Class diagrams generated with the ArgoUML tool. ..................................................... 105

A.3 mel program for extracting facts from UML Class diagrams generated with UMLDesigner. ..................................................... 106
A.4 **mel** program for extracting facts from UML Class diagrams generated with the Eclipse Modelling Tools.

A.5 **mel** program for extracting facts from Feature Model diagram generated with FeatureIDE.

A.6 **mel** program written by Subject A for completion of Task 1 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.

A.7 XQuery program written by Subject A for completion of Task 1 as described in Section 4.2.2.

A.8 **mel** program written by Subject A for completion of Task 2 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.

A.9 XQuery program written by Subject A for completion of Task 2 as described in Section 4.2.2. Line breaks have been removed in some places and added in others to accommodate formatting for this thesis.

A.10 **mel** program written by Subject A for completion of Task 3 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.

A.11 XQuery program written by Subject A for completion of Task 3 as described in Section 4.2.2. Line breaks have been removed in some places and added in others to accommodate formatting for this thesis.

A.12 **mel** program written by Subject A for completion of Task 4 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.

A.13 XQuery program written by Subject A for completion of Task 4 as described in Section 4.2.2. Some line breaks have been added to allow for formatting of this thesis.

A.14 **mel** program written by Subject A for completion of Task 5 as described in Section 4.2.2. Line breaks have been removed in some places and added in others to accommodate formatting for this thesis.

A.15 XQuery program written by Subject A for completion of Task 5 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
A.16 mel program written by Subject B for completion of Task 1 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page. .................................................. 119
A.17 XQuery program written by Subject B for completion of Task 1 as described in Section 4.2.2. .......................................................... 120
A.18 mel program written by Subject B for completion of Task 3 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page. .................................................. 121
A.19 XQuery program written by Subject B for completion of Task 3 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page. .................................................. 123
A.20 mel program written by Subject B for completion of Task 4 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page. .................................................. 124
A.21 XQuery program written by Subject B for completion of Task 4 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page. .................................................. 127
A.22 mel program written by Subject B for completion of Task 5 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page. .................................................. 128
A.23 XQuery program written by Subject B for completion of Task 5 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page. .................................................. 130

B.1 StateMachine diagram for an Arduino stopwatch system (YAKINDU). .... 132
B.2 StateMachine diagram for a polling system meant to be implemented directly on hardware (YAKINDU). ................................. 133
B.3 StateMachine diagram for a blinking LED (YAKINDU). .................... 134
B.4 StateMachine diagram for a Coffee Machine (YAKINDU). ............... 135
B.5 StateMachine diagram for a motion detector (YAKINDU). ................ 136
B.6 StateMachine diagram for a music player (YAKINDU). .................... 137
B.7 StateMachine diagram for a smart home system (YAKINDU). ............ 138
B.8 StateMachine diagram for a traffic light controller (YAKINDU) . . . . . . 139
B.9 StateMachine diagram for an autonomous robot (YAKINDU) . . . . . . 140
B.10 UML Class diagram for a video game named Advent (ArgoUML) . . . . 141
B.11 UML Class diagram for a rotary button system (ArgoUML) . . . . . . 142
B.12 UML Class diagram for a dungeon crawler video game (ArgoUML) . . . 143
B.13 UML Class diagram for a coffee brewing system (Eclipse Modelling Tools). 144
B.14 UML Class diagram for a Java compiler (Eclipse Modelling Tools) . . . . 145
B.15 UML Class diagram for a task management system (Eclipse Modelling Tools). 145
B.16 UML Class diagram for a retail store system (Eclipse Modelling Tools) . 146
B.17 UML Class diagram for Model to code compilation software (Eclipse Modelling Tools). 146
B.18 UML Class diagram for an alarm system (MagicDraw) . . . . . . . . . 147
B.19 UML Class diagram for a library booking system (MagicDraw) . . . . . 148
B.20 UML Class diagram for a music management system (MagicDraw) . . . 149
B.21 UML Class diagram for a parking lot gate system (MagicDraw) . . . . 150
B.22 UML Class diagram for a custom video game engine (UMLDesigner) . . 152
B.23 UML Class diagram for a race management system (UMLDesigner) . . . 153
B.24 Simulink Block diagram for an analog to digital converter quantization algorithm (Simulink). 154
B.25 Simulink Block diagram for the dynamics of air-fuel intake (Simulink). 155
B.26 Simulink Block diagram for an anti-lock brake system (Simulink) . . . . 155
B.27 Simulink Block diagram for a sample anti-windup PID control system (Simulink). 156
B.28 Simulink Block diagram for a bouncing ball (Simulink) . . . . . . . . . 156
B.29 Simulink Block sample diagram for component-based Modelling (Simulink). 157
B.30 Simulink Block diagram for a fault-tolerant fuel control system (Simulink). 157
B.31 Simulink Block diagram for intake airflow estimation and closed loop correction (Simulink). 158
B.32 Simulink Block diagram for a PID tuning system, with Figures B.33 and B.34 (Simulink). 158
B.33 Simulink Block diagram for a PID tuning system, with Figures B.32 and
B.34 (Simulink). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
B.34 Simulink Block diagram for a PID tuning system, with Figures B.33 and
B.32 (Simulink). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
B.35 Arcadia diagram for a train crossing system. . . . . . . . . . . . . . . . . . 160
B.36 A second Arcadia diagram from a train crossing system. . . . . . . . . . . 161
B.37 Arcadia diagram for a security system. . . . . . . . . . . . . . . . . . . . 162
B.38 Arcadia diagram for an unknown system. . . . . . . . . . . . . . . . . . . 163
B.39 Feature Model diagram for a bicycle configurator system (FeatureIDE). . . 164
B.40 Feature Model diagram for a simple car Model (FeatureIDE). . . . . . . . 164
B.41 Feature Model diagram for a desktop searching system (FeatureIDE). . . 165
B.42 Feature Model diagram for an elevator system (FeatureIDE). . . . . . . . 165
B.43 Feature Model diagram for programming expressions expressed as a product
line (FeatureIDE). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
B.44 Feature Model diagram for a simple Hello World program (FeatureIDE). . 166
B.45 Feature Model diagram for the landing gear of a plane (FeatureIDE). . . 167
B.46 Feature Model diagram for a game of Poker (FeatureIDE). . . . . . . . . 167
B.47 Feature Model diagram for a game called TankWar (FeatureIDE). . . . 168
B.48 Feature Model diagram for fictional characters (FeatureIDE). . . . . . . 168
List of Tables

3.1 A summary of the language requirements for mel. ........................................ 19
3.1 A summary of the language requirements for mel. ........................................ 20
3.2 Results of applying the mel Program in Figure 3.2 to the XML fragment in
Figure 3.1 ............................................................................................................. 22
3.3 Results of applying the mel Program in Figure 3.3 to the XML fragment in
Figure 3.1 ............................................................................................................. 26
3.4 Results of applying the first clause of the declaration in Figure 3.7 to the
XML fragment shown in Figure 3.6. Each row represents one fact that would
be extracted and the columns are the values the identifiers would be bound
to. ....................................................................................................................... 30
3.5 Results of applying the second clause of the declaration in Figure 3.7 to the
XML fragment shown in Figure 3.6. Each row represents one fact that
would be extracted. As only one attribute, necessary, is extracted, there
is only that value associated with each fact. ..................................................... 30
3.6 Results of applying the mel program in Figure 3.7 to the XML fragment
shown in Figure 3.6. This is the result of joining the two potential-fact sets
shown in Tables 3.4 and 3.5. ................................................................. 31
3.7 Results of applying the mel Program in Figure 3.10 to the XML fragment
in Figure 3.9. .................................................................................................. 33
3.8 Results of applying the mel Program in Figure 3.12 to the XML fragment
in Figure 3.11 ................................................................................................ 35
3.9 Results of applying the mel Program in Figure 3.14 to the XML fragment
in Figure 3.13 .................................................................................................. 36
3.10 Results of applying the \textit{mel} Program in Figure 3.16 to the XML fragment in Figure 3.15

3.11 Results of applying the \textit{mel} Program in Figure 3.23 to the XML fragment in Figure 3.22

4.1 Necessary facts for extraction from UML StateMachine diagrams.

4.2 Summary of the StateMachine Models considered in this study, including the number of states and transitions in each Model and the total number of entity and relationship facts extracted in the factbase.

4.3 Necessary facts for extraction from UML Class diagrams.

4.4 Summary of the UML Class diagrams considered in this study, showing the number of classes and associations (general associations, aggregations, and compositions) in each Model and the total number of entity and relationship facts extracted in the factbase.

4.5 Necessary facts for extraction from Simulink Block diagrams.

4.6 Summary of the Simulink block diagrams considered in this study, showing the number of blocks and signals in each Model and the total number of entity and relationship facts extracted in the factbase.

4.7 Necessary facts for extraction from Arcadia Logical Architecture diagrams.

4.8 Summary of the Arcadia diagrams considered in this study, showing the number of components, functions, and data flows in each Model and the total number of entity and relationship facts extracted in the factbase.

4.9 Necessary facts for extraction from Feature Models.

4.10 Summary of the Feature Model diagrams considered in this study, showing the number of features and groups (alternative, or, or general groups) in each Mode, and the total number of entity and relationship facts extracted in the factbase.

4.12 Program Sizes of Extractors, where numbers with a single underline are sizes of \textit{mel} 1.1 programs and numbers with a double underline are sizes of \textit{mel} 1.0 programs. All other numbers are sizes of \textit{mel} 1.2 programs.

4.13 Development Time of Extractors. Time in bold denotes the extractor which was written first by the given subject for the Model type.
4.11 A summary of the features added in versions 1.1 and 1.2 (current) of mel (and mint).
Chapter 1

Introduction

As software systems grow increasingly large and complex, they are becoming harder to analyze. Many program-analyses simply do not scale to the sizes of industrial software systems. One strategy to combat this problem is to create a lightweight model of the software system and then analyze the model. For example, compiler technologies and static analyzers can be employed as *model extractors* (e.g., LSME [53], Doop [9], Rigi [52], Frappé [16], eKNOWS Code Model Service [59], Rex [54]) that extract software models (e.g., call graphs, data-flow graphs, dependency graphs) from software artifacts. More generally, extracted models comprise collections of “facts” about a system’s software components — *entities* (e.g., functions, classes, variables), *relationships* (function calls, variable assignments), and *attributes* (e.g., a function is a callback) — that are amenable to analyses expressed as algebraic manipulations [28], graph queries [16, 59], Datalog programs [9], and so on.

The above techniques all assume that the system under analysis comprises various kinds of software entities: source code, object code, build code, configuration files, and other code-related artifacts. However, engineers do not always have access to all of the system’s source artifacts. Some components, libraries, or subsystems may be procured from third parties (e.g., based on a specification Model). Hereafter, we distinguish between *prescriptive and descriptive models* (denoted as capitalized *Models*) that are artifacts produced by engineers as part of the software-development process versus *extracted models* (denoted as lowercase *models*) that are generated from software, Models, and other artifacts. Other components may be generated automatically from Models that are more descriptive and semantically informative than the generated code. Moreover, software developers use software models as an additional artifact when developing and comprehending software systems, ranging from the deep end of Model-driven engineering (MDE) to simply using
Models for communication between engineers or as documentation [30]. While Models may not be completely ubiquitous they certainly are a common artifact of software engineering products and are seemingly here to stay. As such, to create a single coherent model of the entire system, we would need a means to extract facts from Models.

The construction of fact extractors for Models poses a major challenge. There is a wide variety of Model representations: multiple types of Models may be employed, and different types of Models may be expressed in different notations and representations. Even when Models are expressed in eXtensible Markup Language (XML) [10] and Model editors for the same Model types (e.g., UML Model editors) use the XML Metadata Interchange (XMI) [18] standard for expressing metamodels and metadata, the Models’ XMI schema vary considerably. A specialized extractor may need to be created for every pair of Model editor and supported Model type.

This thesis introduces a model extractor language (mel) and mel interpreter (called mint) to support rapid development of fact extractors for Models that are represented in the XML Metadata Interchange (XMI) up to and including the version 2.5.1 standard [18]. mel is a domain-specific language (DSL) for succinctly expressing which facts (e.g., Model elements, relationships, attributes) are to be extracted from a Model. The mel language is designed to allow the user to concentrate on the essential complexity of deciding what information to extract from a Model and leave to mel and mint the accidental complexity of how to pluck that information from an XML/XMI representation.

1.1 Thesis Contributions

The contributions proposed by this thesis are:

- The development and presentation of a domain-specific language mel, and corresponding interpreter mint, for specifying and extracting model-based information from a Model represented in XML.

- An evaluation of the extent to which mel is expressive enough to apply to different Model types, including UML class and state machine diagrams [19], Arcadia Logical Architecture diagrams, [67], Feature Models [43], Simulink Block Diagrams [49].

- An evaluation of whether it is easier to write a mel program to create a fact extractor for a particular Model representation than it is to develop the fact extractor from scratch using existing XML querying tools.
1.2 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 starts with a background on fact extraction, XMI and XML as model representations, and works and tools related prior to the process of XML analysis and transformation.

Chapter 3 introduces mel, specifies its input language, and describes its implementation.

Chapter 4 reports the results of studies that evaluate the effectiveness of mel in easing the creation of new model extractors for Models and the extent of mel’s ability to generalize to a variety of XML/XMI-based Models.

Lastly, Chapter 5 discusses conclusions drawn from the evaluation of mel.
Chapter 2

Background

This chapter discusses the context necessary for understanding mel and its purpose. Mainly, this chapter presents information on fact extraction, XML as a language, and existing tools seeking to aid the transformation or distillation of XML data.

2.1 Fact Extraction

A fact extractor is a program built to analyze a structured software artifact and produce a specific customized set of desired information about that artifact. The data set produced by a fact extractor can be called a factbase - as it is a database of facts about the particular software artifact. A produced factbase also represents a particular model of the software artifact, defined by the software details that the extractor is designed to extract from the artifact.

Most commonly, fact extractors are built to extract facts from source code or even directly from a compiled binary file. Given that a fact extractor is a program built to extract desired information from a software artifact, many commercial static-analysis tools can be considered to contain within them a fact extractor that pulls from the source code the information it requires for its analysis. For example, SciTools Understand [62], a popular IDE that provides static-analysis features, includes an extractor.

As a factbase is just a collection of data about a software artifact, it can have any form. In this thesis, a factbase is a collection of entities and relationships between entities, both of which can be annotated with attributes without any particular formatting.
2.1.1 Extraction of Lightweight Models

We will consider a lightweight model to be a model that abstracts a software system at a high level, including only as much information as is exactly necessary for the desired analyses. The purpose of extracting only exactly as much information as is necessary is to both avoid information overload on the user, as well as to facilitate efficient analysis of very large software systems. There is a rich history of extracting lightweight models from source-code artifacts for a variety of purposes, including the construction of high-level architectural views [8, 51] and the validation of developers’ mental models of a system’s design against the code [53]. At their most basic, the models produced by such tools are based on facts extracted from the source code; the models can be augmented with facts drawn from other development information such as version control meta-data, process metrics, and data from the dynamic instrumentation of the running system. Each extractor understands what information to collect from its respective target, and there is a central model of the system into which this extracted information is integrated. Extracting information from diverse artifacts is important because it improves one’s existing knowledge about the system and permits new kinds of analyses to be performed. In this thesis, we propose a tool to aid in extracting information from software Models, such as UML Class and StateMachine diagrams, Arcadia Logical Architecture diagrams, and Feature Models. The information extracted from such Models can then be included with information extracted from source code to provide a more holistic view of the software. To the best of our knowledge, this has not been done before.

An architecture for a fact extraction tool pipeline might look like Fig. 2.1. To perform extraction at scale, extractors typically first process individual development artifacts producing a set of factbases, one for each source artifact. After the individual artifacts have been processed, the results are typically merged together into a single factbase, where references within one artifact to model elements in other artifacts are resolved. The structuring of the facts — which come from a diverse set of artifacts — into a coherent system model creates a common vocabulary that aids the end-user in phrasing queries and performing analyses over the augmented system model.

Our factbases model information about program entities — such as global variables, function, and files/classes — and the relationships between them, such as containment/ownership, function calls, and global variable accesses. Additionally, both entities and relationships can have attributes, such as function parameter types, the line number within a file where the entity/relationship is located, and if a variable access by a function is read or write.
2.1.2 Heterogeneous Fact Extraction

Source code is not the only relevant artifact produced during software development. Software artifacts can include source code, object code, build code, configuration files, various system Models, version-control log data, and other code-related artifacts. It is not uncommon for extractors of artifacts other than source code to be created in order to build a fuller picture of a piece of software.

Heterogeneous fact extraction is the act of extracting facts from more than one type of software artifact. The goal of heterogeneous fact extraction is to provide a more holistic and complete view of a piece of software. For example Passos and Czarnecki [56] developed an extractor to scrape the Linux kernel Git repository in order to extract data about feature additions and removals [56]. This extractor was built specifically to extract information that could not have been gleaned from source code alone and was used to form a fuller understanding of the evolution of the Linux kernel from a feature-oriented perspective [57].

Creating an extractor requires the developer to both (1) identify the relevant information from the artifact, and (2) write software to find and pluck that information from out
of the artifact. One issue that arises is that the developer of an extractor may not know all
the information users of the extractor may desire from the artifact. The technical challenge
of locating and plucking information from the artifact depends on the type of artifact itself.
If the artifact is source code then the process likely involves walking the abstract syntax
tree of the source code to locate relevant pieces of information. In the case of Passos and
Czarnecki’s work, their extractor needed to execute API requests to obtain the relevant
git log information and then scrape those files for the information they were interested in.
In the case of Software Models represented in an XML format, the effort required is the
traversing of the XML tree to find relevant tags and relationships between them - often
traversing the tree multiple time to resolve references between entities.

This thesis supports heterogeneous fact extraction by providing a language and tool
for the rapid development of Model extractors. mel eases the task of extractor creation by
allowing the developer to focus only on identifying the information they are interested in
and not on the minutiae of how to find and pluck that information from the XML.

2.2 XML and XMI

Many Modelling languages have a graphical representation, but Modelling tools represent
this as a textual representation to ease the storing and sharing of Models. The most com-
mon textual representation for Models is the eXtensible Markup Language (XML) [10].
A further specialization of XML for representing Models has been created by the Object
Management Group (OMG): the XML Metadata Interchange (XMI) standard [18]. The
XMI standard is sufficiently expressive to represent Models belonging to any Model stan-
dard published by the OMG, including UML and SysML diagrams. The purpose of XMI
is to standardize the representation of OMG specified Models allowing for the exporting
and importing of models across compliant tools. Because XMI is a specialization of XML
they share many features/rules. The term XM* will be used when referring to both XML
and XMI.

An XM* representation is a tree of nested tagged nodes, with a single root node. Each
node comprises an opening tag, content, and matching closing tag:

<tagname> content </tagname>

A node’s content can contain text, nested nodes, or both. It is common for attribute values
to also be stored as attributes of a node which must be written in the node’s opening tag,
for example:
When a node conveys all of its information as attributes, it is useful to express it as an empty node that is simply a self-closing tag with attributes and no content. A node is marked as self-closing by prepending its closing > with a forward slash to become />, for example:

```xml
<tagname attr1="Value1" attr2="Value2"/>
```

Seen in Figure 2.2 is a tiny XMI Model that declares a UML Class named “C1”, which has a feature represented as a child node.

```
<xmi:XMI version="2.0" ...>
  <UML:Class name="C1" xmi:type="uml:Class" xmi:id="_1">
    <feature xmi:type="UML:Attribute" xmi:id="_2" name="a1" />
  </UML:Class>
</xmi:XMI>
```

Figure 2.2: A simple Model expressed in XMI.

Relational information in XM* documents is often encoded in one of two ways. The first common encoding is through an ancestor/descendant relationship as seen between the feature and class nodes in Figure 2.2. The second common method is to encode relational data through unique identifier attributes and through references to an identifier attribute of another node. Figure 2.3 demonstrates this alternative method of using unique identifiers to encode the same ancestor/descendant relationship between nodes class and feature that appears in Figure 2.2.

```
<xmi:XMI version="2.0" ...>
  <UML:Class name="C1" xmi:type="uml:Class" xmi:id="_1">
    <feature xmi:type="UML:Attribute" xmi:id="_2" name="a1" parentClass="_1"/>
  </UML:Class>
</xmi:XMI>
```

Figure 2.3: A simple Model expressed in XMI representing relational data using unique identifiers.
2.2.1 Differences between XML and XMI

As XMI is a specialization of XML, any valid XMI document is also a valid XML document whereas the converse is not always true. An XML document is an XMI document only if it conforms to the specifications laid out in the XMI specification [18].

Since XMI is meant to standardize model representations, the primary difference between XMI and arbitrary XML are the restrictions on the tags that may be used. The XMI standard also specifies how tags and their attributes must be used, and what they represent. Effectively XMI is a smaller more controlled subset.

2.2.2 EMF Core (ECore)

The Eclipse Modelling Framework (EMF) is a framework built by the Eclipse Foundation for the modelling of software and generation of code from Models. EMF Core (ECore) [63] is the meta model for describing Models and serializing them to XMI. Many of the Models that are publicly available through the Model database compiled by Hebig, Ho-Quang, Chaudron, Robles, and Fernandez and [25] were created using one of the many Eclipse-based modelling tools, and thus are represented in EMF ECore.

While ECore representation is XMI, ECore differs from most other common XMI representations in one major way: References to existing tags, or declarations that would involve a reference to an existing tag, are written in ECore as an absolute or relative path to the node. Figure 2.4 illustrates this, showing how a bi-directional relationship with labels inventory and store is encoded in the ECore XMI representation. The relationship that class Product has with class ConvenienceStore is stored as an eStructuralFeatures tag. The eType and eOpposite attributes of this eStructuralFeatures tag illustrate ECore’s form of encoding references as paths. That is,

- The store relationship of class Product encodes its target as an eType Attribute, which has value #//ConvenienceStore. The #// refers to the root of a document. If a path to a Document appears on the left of the #, then it refers to the root of that specified Document; if nothing appears on the left, then it the path starts at the root of the current Document. So this particular eType attribute resolves to the ConvenienceStore type defined in this document.

- Secondly, the eOpposite attribute encodes the corresponding relationship in a bi-directional relationship. Here, the eOpposite attribute has the value
Figure 2.4: A simple UML class diagram created with the EMF Core Diagram Editor and a Snippet of the corresponding ECore representation. This shows the entirety of the representation of the two classes and their relationships. The `eType` and `eOpposite` attributes demonstrate how ECore encodes references to other data in the Model. (Produced with Eclipse Modeling Tools version 4.19.0).

`#//ConvenienceStore/inventory`, which refers to the `inventory` relationship within the `ConvenienceStore` class.

- The opposing relationship `inventory` has similar references to the `Product` class and its `store` relationship.
2.3 XML Analysis & Transformation Tools

A wide variety of tools exist to interpret and translate XML documents. Because XMI is a specialization of the XML, these tools implicitly function on XMI documents as well. The World Wide Web Consortium (W3C) defines the XML specification and it has also produced a number of tools for working with XML documents.

This section discusses the W3C tools XQuery [61] and XPath [13] as well as several academic tools for working with XML documents.

2.3.1 XPath and XQuery

XPath and XQuery are tools for querying and working with the contents of an XML file. Because W3C defines the XML and its specification, XPath and XQuery belong to the W3C recommended tool set for working with XML documents.

XPath

XPath is a query language in which the user can write queries to specify matching nodes and attributes in an XML file based on expressions of paths in the XML tree. Simple relationships can be represented in XPath through the use of XPath Axes which allow the user to refer to nodes via their relationship to a current node.

Figure 2.5 provides a simple XML fragment with relational data between vegetables. The figure shows two groups of vegetable types (Greens and Roots) each of which comprises three individual “veggie” child nodes and an additional non-grouped vegetable “Tomato”. Figure 2.6 demonstrates several XPath queries that could be applied to the XML fragment in Figure 2.5 to extract particular pieces of information. The queries’ purposes and behaviours are as follows:

Line 1 This query selects all <veggie> nodes in the XML document. The // syntax specifies a descendant relationship. A query that starts with a // specifies any descendant of the root of the document thereby effectively matching any node in the document. If this query is applied to the XML fragment in Figure 2.5 the result set contains all seven <veggie> nodes in the document.

Line 2 This query selects all <veggie> nodes that are direct children of a <vegetables> node. The / syntax indicates a direct parent-child relationship. Thus the //vegetables
Figure 2.5: An XML model with groups of siblings

```xml
<root>
  <vegetables label="Greens">
    <veggie id="1" label="Asparagus"/>
    <veggie id="2" label="Broccoli"/>
    <veggie id="3" label="Kale"/>
  </vegetables>
  <vegetables label="Roots">
    <veggie id="4" label="Carrot"/>
    <veggie id="5" label="Onion"/>
    <veggie id="6" label="Turnip"/>
  </vegetables>
  <veggie id="7" label="Tomato"/>
</root>
```

Figure 2.6: Five simple XPath queries, one per line

- `//veggie` selects all <veggie> nodes in the document.
- `//vegetables/veggie` selects all <veggie> nodes that are direct children of <vegetables> nodes. If this query is applied to the XML fragment in Figure 2.5, the result set contains all <veggie> nodes in the document except for the one with label “Tomato”.
- `//vegetables/@label` selects all the values of the label attributes of all <vegetables> nodes in the document. If this query is applied to the XML fragment in Figure 2.5, the result set contains the two strings “Greens” and “Roots”.
- `//vegetables[@label="Greens"]/child::veggie` selects all <veggie> nodes that are direct children of <vegetables> nodes that have a label attribute whose value is “Greens”. If this query is applied to the XML fragment in Figure 2.5, the result set contains the three <veggie> nodes with label “Greens”.
- `//vegetables[@label="Roots"]/child::veggie` selects all <veggie> nodes that are direct children of <vegetables> nodes that have a label attribute whose value is “Roots”. If this query is applied to the XML fragment in Figure 2.5, the result set contains the two <veggie> nodes with label “Roots”.
found under the first `<vegetables>` node — that is those with labels “Asparagus”, “Broccoli”, and “Kale”.

**Line 5** This query behaves similarly to the query on Line 4 with the exception being that the `<vegetables>` nodes selected in the first portion of the query have a `label` attribute value of “Roots” instead of “Greens”. If this query is applied to the XML fragment in Figure 2.5, the result set contains the three `<veggie>` nodes found under the second `<vegetables>` nodes — that is those with labels “Carrot”, “Onion”, and “Turnip”.

Such queries are useful and effective for selecting elements in a factbase but not for deriving new facts to include in the factbase. For example, one would not be able to write an XPath query to generate a “sibling” relationship between `<veggie>` nodes in the XML fragment shown in Figure 2.5. However using an XPath processor in conjunction with a general purpose programming language, a programmer could combine query results to generate such a sibling relationship for their factbase.

In conclusion, although XPath is a useful query language for extracting specific nodes or groups of nodes from an XML document, it lacks the innate ability to generate relational data between nodes. The more powerful XQuery can aid a developer in this regard.

**XQuery**

XQuery is a mature general-purpose XML query and transformation language that is designed to transform XML into either other hierarchical representations or user-formatted reports. XQuery was developed after and built on top of XPath. As such XQuery is a superset of XPath and almost any valid XPath query is valid in XQuery as well.

As a more general-purpose language, XQuery allows for much more complex data to be produced than XPath allows and queries can be more complex. For example, the sibling relationship between `<veggie>` nodes in Figure 2.5 can be derived with the XQuery program shown in Figure 2.7. The constructs used in this example are a mix of XPath selectors and common imperative programming concepts; their behaviour is as follows:

**Line 1** defines a variable `$veggies` and assigns its value to the collection of tags returned by the subsequent for loop.

---

1 With small exceptions for old version compatibility modes and expansion of character entity references (e.g. ‘&amp;’ is & in XQuery and remains &amp; in XPath).
let $veggies :=
    for $veggie in //veggie
        return <veg id="{$veggie/@id}" name="{$veggie/@label}" />

let $siblings :=
    for $vegetable in //vegetables
        for $veggie in $vegetable//veggie
            for $v2 in $veggie/following-sibling::veggie
                return <sibling s1="{$veggie/@label}" s2="{$v2/@label}" />
    return <data>
    { $veggies }
    { $siblings }
</data>

Figure 2.7: XQuery program for generating a sibling relationship

Line 2 is a for loop where an iterator variable $veggie is used to iterate through the nodes produced by the XPath selector //vegetables. That is, this for loop iterates over each veggie node in the original document.

Line 3 has a return statement which constructs a new fact with the tag veg, whose id attribute has the value of the veggie node’s id Attribute and name attribute has the value of the veggie node’s label Attribute. As this return statement is within a for loop it constructs one fact for each node iterated by the for loop.

Line 5 defines a variable siblings and assigns its value to the collection of tags returned by the subsequent for loop.

Line 6 is a for loop that iterates over every vegetables node in the original document, binding each to the iterator variable $vegetable.

Line 7 is a nested for loop that iterates over every veggie node that is underneath the vegetables node currently selected by the enclosing for loop in Line 6.

Line 8 is another nested for loop that applies the following-sibling XPath axis to the veggie node currently selected by the enclosing for loop in Line 7. The axis application $veggie/following-sibling::veggie produces all of the subsequent veggie node siblings of the selected node $veggie, and the collection is iterated over by $v2.
Line 9 is a return statement constructs a new fact node with a sibling tag whose s1 attribute is the label of the veggie node currently selected in the for-loop on Line 7 and the s2 attribute is the label of the veggie node currently selected in the for-loop on Line 8.

Lines 10-13 specify the ultimate return value of this XQuery program. As XQuery produces hierarchical data, the produced value is a single data node that encloses the produced veg nodes stored in the variable $veggies and the sibling nodes stored in the variable $siblings. The produced results can be seen in Figure 2.8.

Although XQuery is sufficient as a query language for XM* documents, and any mel program can be approximated in XQuery, mel seeks to abstract the process of fact extraction to a higher level than XQuery does. Particularly, XQuery is not focused on the resolution of relational data. Rather, XQuery allows users to refer to collections of XM* nodes, but also requires that the processing of those nodes be explicitly done by the user. Commonly, this takes the form of writing multiple loops to resolve relationships the user is interested in. The procedure of looping through potential nodes of interest and processing the ones actually relevant to the user’s interests is exactly the type of accidental complexity that mel seeks to relieve the programmer of. Where mel allows the user to state the
relationship they are interested, XQuery requires the user defines programatically how that relationship be resolved.

2.3.2 Academic Developed XML Tools

In addition to the W3C tools, there has been academic interest in developing tools for processing XML documents. In this section several of these relevant works are discussed.

XTQ

XML Tree Query (XTQ) [45] is a declarative XML query language presented by Li, Liu, Zhu, and Ghafoor. XTQ’s syntax is heavily influenced by both XQuery and SQL. XTQ provides a powerful and expressive query language that expands on existing XML query languages at the time of its writing, primarily:

1. XTQ allows one to write expressive patterns to explicitly specify conjunctive, disjunctive, and hierarchical relationships between matching data elements.

2. XTQ’s data filtering mechanism allows for easy filtering on composite data structures (made of many elements from the XML). The claim is that such queries could not be easily written in other languages without the use of multiple sub-queries.

In areas where XTQ succeeds, so too does mel. Additionally, mel seeks to further support extraction of relational data and is more focused on the production of user-defined relationships than XTQ is.

XPathLog

XPathLog, presented by May [50], is a declarative language meant to be used as an extension to XPath. XPathLog focuses on querying and modifying an existing XML document, whereas mel focuses on reasoning about XML in a relational context. Also, mel focuses on operating on and outputting sets of tuples (that is, factbases) from an XML artifact, rather than producing hierarchically structured XML.
Model Transformation

Bhatti and Malik [6] propose a tool for bi-directional transformation between the XML for one Model and the XML for another Model. This work relates to mel in that both seek to transform an XML representation of a Model into a different representation. The tools differ, however, in that mel seeks to extract from an existing Model only the facts the user desires based on expressive queries the user can write, whereas the bi-directional transformation tool translates an entire model from one XML schema to another. The proposed bi-directional framework is preliminary, has been applied only to small examples, and is limited to cases where the tool is provided with the metamodels for both the source and target XML representations.

XML Translation to Databases

Somewhat analogous to mel is the work on translating an XML document into a database. One example is the work on s-XML by Subramaniam, Haw, and Hoong [65], which is a technique of mapping an XML document into a relational database. Although mel also produces a database, mel seeks to produce a database comprising only the information the user is interested in. This distinction is an important difference between mel and tools that produce databases from XML documents. Firstly, these tools offer no actual user-defined processing of the XML data, rather they produce a relational database and the user must then write queries to that database to resolve the facts they are interested in. mel itself functions as both a relational query language and a tool for translating XML documents into databases. Furthermore, the language features of mel are specifically designed to aid in the querying and specifying of relationships of interest as they appear in XML documents. This is not true of general purpose relational databases, and once a translation tool has produced a database for querying the user must write queries about their XML document in a traditional query language with no concept of how data was structured in the XML document. Additionally, the work on s-XML notes a common failure of XML mapping methods to store all relationships between nodes of importance. This problem does not exist in mel, as the important relationships are explicated in the program the user writes. Additionally, the effort expended to write a mel program is not saved by using a tool like s-XML, because the mel program reflects the relational queries the user will write once a mapping tool produces a database for them.
Chapter 3

Model Extractor Language (mel)

In this chapter we introduce mel, which is a small declarative language for specifying fact extraction from Model artifacts, and the mel interpreter called mint. mel is a domain-specific language that allows the user to specify the facts that they want to extract from a Model. Such a specification will vary not only according to the type of Model under study (e.g., class facts from Class diagrams versus state facts from StateMachine diagrams) but also the representation of the Model (i.e., how classes or states are represented textually in a Model) and the user’s needs (not all extractable facts will be relevant to the user’s purpose). The intended scope of applicability of mel is any Model represented in XM*.

One of the challenges is that, although there exists a standard for rendering textual representations of Models, different Modelling tools may employ radically different XMI schema, even for the same notation.

This chapter is organized as follows: Section 3.1 defines the core required features mel must provide in order to satisfy extracting facts from various XM* documents, Section 3.2 introduces the general structure of a mel program and major language features, Section 3.3 displays the full mel grammar and discusses the specific organization of a mel program, and Section 3.4 introduces the mel interpreter mint and explains its usage.

3.1 Requirements of mel

mel was developed using the research methodology of design science [27]. As part of the design science methodology, the required features were identified as part of an iterative development process in which mel programs were developed to extract information from a
variety of Model types. This iterative process led to the identification of several ways that essential information is encoded in the XML representation of Models. These identified encoding methods dictate the requirements of mel, as the language must allow the user to specify the extraction of facts encoded in those ways. Each of these requirements corresponds to a relationship between the XM* document and some desired facts that the developer of a Model extractor would need to identify. Nine requirements for mel are listed in Table 3.1. The satisfaction of these nine requirements is sufficient for mel to extract all the Model information identified in the Models considered in this thesis.

Table 3.1: A summary of the language requirements for mel.

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Refer to XM* nodes by tag</td>
<td>The language must provide a method of referring to nodes in the XM* document by tag type. A reference to an XM* tag type should resolve to all the nodes in the XM* document that have that tag.</td>
</tr>
<tr>
<td>2</td>
<td>Extract values of Attributes of XM* nodes</td>
<td>The language must provide a method to specify which Attributes of an XM* node should represent identifiers or attributes in the factbase.</td>
</tr>
<tr>
<td>3</td>
<td>Refer to pairs of XM* nodes that are related by Attribute values</td>
<td>The language must provide a method to relate two XM* nodes when their specified Attributes have matching values. The reference should resolve to all pairs of such nodes in the XM* document whose Attribute values are equal.</td>
</tr>
<tr>
<td>4</td>
<td>Refer to pairs of XM* nodes that are related by a parent/child relationship</td>
<td>The language must provide a method to relate two XM* nodes (with specified tags) when one node is a child of the other. The reference should resolve to one pair of nodes for each node in the XM* document of (with the specified child tag) that has a parent node (with the specified parent tag).</td>
</tr>
<tr>
<td>5</td>
<td>Refer to pairs of XM* nodes that are related by an ancestor/descendant relationship</td>
<td>The language must provide a method to relate two XM* nodes when one node is a descendant of the other. The reference should resolve to one pair of nodes for each node in the XM* document (with the specified descendant tag) that has an ancestor node (with the specified ancestor tag).</td>
</tr>
</tbody>
</table>
Table 3.1: A summary of the language requirements for mel.

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Constrain results based on the values of Attributes/attributes</td>
<td>The language must provide a method to write constraints on the allowed values for Attributes of referenced XM* nodes or attributes of extracted facts. Specifically, it must allow constraints on the equality/inequality of string values.</td>
</tr>
<tr>
<td>7</td>
<td>Refer to tuples of XM* nodes that are children/descendants of the same XM* node</td>
<td>The language must provide a method to refer to a collection of XM* nodes (with specified tags) that are all children of the same parent node (with a specified tag); or are all descendants (with specified tags) of the same ancestor node (with a specified ancestor tag).</td>
</tr>
<tr>
<td>8</td>
<td>Refer to hierarchies of XM* nodes in the XM* document</td>
<td>The language must provide a method to refer to a series of XM* nodes (with specified tags) that appear in a specified structural hierarchy in the XM* document.</td>
</tr>
<tr>
<td>9</td>
<td>Refer to substrings of XM* Attributes</td>
<td>The language must provide a method to refer to a specified substring of an XM* Attribute value.</td>
</tr>
</tbody>
</table>

It is also worth noting that while the design science methodology used focused on XM* representations of Models, these requirements mean that mel is capable of extracting information from a wide range of XM* documents - not just those that represent software Models. However, mel has been specialized to focus on the types of XM* documents that represent Models, as these were the documents considered in the iterative development process of the language.

### 3.2 Usage of mel

A mel program is a sequence of declarations of fact types: each declaration starts with the name of a fact type to be extracted and the name(s) of its parameter(s) followed by the clauses that specify how facts of that type are constructed from information in the Model. The left-hand side and right-hand side of a declaration are separated by the token “|->”.

Consider the declarations provided in Figure 3.2. A fact type with one parameter is an
entity fact; its parameter is the unique identifier for an extracted entity of that type. A fact type with two parameters is a relationship fact; its parameters refer to the two entities that are being related. On the right-hand side, each clause is one of the following types:

**XM* reference clause:** a reference to XM* nodes in the input Model, as in the first three rules in Figure 3.2.

**Declaration reference clause:** a reference to a declaration previously made in the mel program, as in the fourth rule in Figure 3.2 (discussed further in Section 3.2.1).

**XM* path reference clause:** a reference to specified patterns of XM* nodes that appear in the input Model (discussed further in Section 3.2.10).

**Constraint:** a restriction on the value of of bound identifiers (discussed further in Section 3.2.2).

An XM* reference clause starts with the name of an XM* tag followed by a property list. A property list is a comma-separated list of properties enclosed in curly braces, where each property is an attribute binding or a requirement\(^1\). An attribute binding is the binding of some value to a named identifier in the resultant fact. Specifically, the binding operator, “:”, is a binary operator that binds the value of its left-hand operand to the name provided

\(^1\)Requirements and their syntax are discussed fully in Section 3.2.8
Table 3.2: Results of applying the \texttt{mel} Program in Figure 3.2 to the XML fragment in Figure 3.1

<table>
<thead>
<tr>
<th>Row No.</th>
<th>Fact Type</th>
<th>First Param</th>
<th>Second Param</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>component</td>
<td>CompA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>component</td>
<td>CompB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>component</td>
<td>CompC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>feature</td>
<td>F1</td>
<td></td>
<td>{ label=&quot;Feature 1&quot; }</td>
</tr>
<tr>
<td>5</td>
<td>feature</td>
<td>F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>feature</td>
<td>F3</td>
<td></td>
<td>{ label=&quot;Feature 3&quot; }</td>
</tr>
<tr>
<td>7</td>
<td>featParent</td>
<td>CompA</td>
<td>F1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>featParent</td>
<td>CompB</td>
<td>F2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>featParent</td>
<td>CompA</td>
<td>F3</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>sibling</td>
<td>F1</td>
<td>F2</td>
<td>{ C=&quot;CompA&quot; }</td>
</tr>
</tbody>
</table>

as the right-hand operand. Thus, a binding takes the form $X:Y$, where $X$ is most commonly the name of an XM* Attribute\(^2\) inside of an XM* node in the Model, and $Y$ is the name of one of a fact type’s parameters or is a declaration/reference to an attribute of the fact type.

To understand how \texttt{mel} declarations extract facts from a Model, consider the small pedagogical Model given in Figure 3.1, in which component \texttt{CompA} contains features \texttt{F1} and \texttt{F3}, and component \texttt{CompB} contains feature \texttt{F2}. Suppose that the \texttt{mel} user would like to extract from the Model (1) the set of components, (2) the set of features, (3) information about which components the features belong to, and (4) information about which features reside in the same component. The \texttt{mel} program in Figure 3.2 comprises four definitions that correspond to the four types of facts to be extracted.

Line 1 is one of the most simple declarations that can be written in a \texttt{mel} program.

\texttt{component(C)} declares that the factbase should have entity facts of type \texttt{component}.

The identifier of each entity is bound to the parameter $C$, so that the identifier can be referred to by name in the clauses of this declaration.

\texttt{compFeatNode{component:C}} matches instances of the XM* tag \texttt{compFeatNode}.

The value of the XM* Attribute \texttt{component} of each such node is bound to parameter $C$ (i.e., the identifier of the extracted entity fact). This declaration

\(^2\)As with Model vs. model, we distinguish between XM* Attributes in the input Model from attributes in the extracted model by capitalizing all references to the former.

---

22
requires an identifier so if the Model contained a `compFeatNode` node without a `component` Attribute, it would not be extracted.

**The results** of applying this declaration to the Model shown in Figure 3.1 are shown in the first three rows of Table 3.2.

**Line 2** is similar to the declaration in Line 1, but it includes an additional binding that extracts an attribute associated with `feature`.

`feature(X)` declares that the factbase should have entity facts of type `feature`. The identifier of each entity is bound to the parameter `X`.

`compFeatNode{id:X, userName:label}` matches instances of the XM* tag `compFeatNode`. The value of the XM* Attribute named `id` of each such node is bound to parameter `X` (i.e., the identity of the extracted entity). Additionally, the value of the nodes XM* Attribute `userName` is bound to the attribute `label` associated with the extracted entity. By default, declared attributes are optional. Thus, although a `compFeatNode` node requires an `id` Attribute in order to be matched by this clause, it does *not* require a `userName` Attribute. It is also possible to require that node possess an XM* Attribute in order to satisfy a clause, as shown in Section 3.2.8.

**The results** of applying this declaration to the Model shown in Figure 3.1 are shown in the fourth to sixth and last two rows of Table 3.2. It should be noted that all `feature` entities `F1`, `F2`, and `F3` are extracted, but only `F1` and `F3` have `label` attributes (because the XML node for `F2` does not have a `userName` Attribute); thus entity `F2` is still extracted despite not having a `userName` Attribute.

**Line 3** declares a relationship as opposed to an entity.

`featParent(C, F)` declares that the factbase should have relationship facts of type `featParent`. The first entity in each relationship is bound to the parameter `C` and the second entity is bound to the parameter `F`.

`compFeatNode{component:C, id:F}` matches instances of the XM* tag `compFeatNode`, as before. The `component` and `id` Attributes of each node are bound respectively to the parameters `C` and `F` for each extracted relationship. Because `C` and `F` are parameters of the relationship, only `compFeatNode` nodes that have both `component` and `id` Attributes are matched.

**The results** of applying this declaration to the Model shown in Figure 3.1 are shown in rows 7-9 of Table 3.2.
Line 4 declares another relationship and is the first declaration to use a declaration modifier, refer to previously extracted facts, include a constraint clause, and have multiple clauses. The exact semantics of declarations with multiple clauses (called compound clauses) will be discussed in Section 3.2.3.

\(^\text{sibling}(F, G)\) declares that the factbase should have relationship facts of type sibling. The first entity in each relationship is bound to the parameter \(F\) and the second entity is bound to the parameter \(F\). The prefix \(^\text{\textasciitilde}\) is a declaration modifier that specifies that this declaration should not be commutative. That is, facts of the form sibling \(A \ B\) and sibling \(B \ A\) should not both appear in the factbase. The commutative nature of the relationship may be undesirable by the user, as the cycle formed by it may have negative effects on future analyses depending on the analysis method. As such the \(^\text{\textasciitilde}\) modifier is provided to allow users to specify extraction of only one edge in an otherwise commutative relationship. The prefix \(^\text{\textasciitilde}\) should be used whenever a relationship is non-directed but the clauses do not disallow both facts to be produced; \(\text{mel}\) will extract whichever relationship it sees first while scanning the XM* document and will ignore the second relationship.

\text{featParent}(C, F)\) matches instances of the previously extracted relationship \text{featParent} and binds the first entity in the relationship to attribute \(C\) and the second entity to parameter \(F\).

\text{featParent}(C, G)\) matches instances of the previously extracted relationship \text{featParent} and binds the first entity in the relationship to attribute \(C\) and the second entity to parameter \(G\). If an identifier is used multiple times in the same declaration, it refers to the same value in all instances. Thus, the \(C\) in the first \text{featParent}(C,F)\) clause must match the \(C\) in this \text{featParent}(C,G)\) clause, generates a relationship between entities \(F\) and \(G\).

\(F\neq G\) is a constraint clause which restricts this declaration to only facts where the parameter \(F\) is not the same as the parameter \(G\). That is, the sibling relationship should be non-reflexive and include any relationship instances between a feature and itself. The exact semantics of constraint clauses is discussed further in Section 3.2.2.

The result of applying this declaration to the Model shown in Figure 3.1 is row 10 of Table 3.2.

This example illustrates the basic usage of \text{mel}. At its core, \text{mel} is meant to facilitate the rapid development of queries to extract data from Models, so that the user can concentrate
on the essential complexity of deciding what information to extract from a Model and leave to mel and mint the accidental complexity of how to pluck that information from the XM* representation.

When extracting facts from a textual representation of a Model, the important questions for the user to answer are: “What parts of the model do I care about?” and “What relationships in the model are important?”. Each declaration of a mel program is an answer to one of these two questions and nothing more. A mel user need not write any cumbersome code to parse a model’s textual representation, pluck out the information they care about, and constrain the result set to facts that satisfy certain conditions. Instead the mel user declares facts they are interested in and the clauses necessary for deriving those facts from the Model.

This concludes our introduction to basic mel. Many additional features and intricacies to the language are discussed in the following subsections.

### 3.2.1 Declaration Reference Clauses

A declaration reference clause is a clause that refers to fact type previously defined in the mel program. The two featParent clauses that appear in the sibling declaration (line 4 in Figure 3.2) are examples of declaration reference clauses (referring to the fact type defined in line 3).

```
1 featParent(C, F) |- compFeatNode{component:C, id:F,
2             userName:fLabel};
3 feature(X) |- featParent(owningComponent, X){fLabel:label};
```

Figure 3.3: A mel program that demonstrates the use and semantics of declaration reference clauses.

Specifically, a declaration reference clause binds its parameters to the parameters of the referenced definition.

To exemplify the concept of declaration reference clauses, consider the mel program shown in Figure 3.3, which provides an alternative definition of the feature entity shown in Figure 3.2. In this program, a feature entity is produced for each extracted instance of the featParent relationship, rather than being based on instances of the compFeatNode XM* nodes in the Model. The results of applying this program to the XML fragment
Table 3.3: Results of applying the $\text{mel}$ Program in Figure 3.3 to the XML fragment in Figure 3.1

<table>
<thead>
<tr>
<th>Fact Type</th>
<th>First Param</th>
<th>Second Param</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>featParent</td>
<td>CompA</td>
<td>F1</td>
<td>{ fLabel=&quot;Feature 1&quot; }</td>
</tr>
<tr>
<td>featParent</td>
<td>CompB</td>
<td>F2</td>
<td>{ fLabel=&quot;Feature 2&quot; }</td>
</tr>
<tr>
<td>featParent</td>
<td>CompA</td>
<td>F3</td>
<td>{ fLabel=&quot;Feature 3&quot; }</td>
</tr>
<tr>
<td>feature</td>
<td>F1</td>
<td></td>
<td>{ owningComponent=&quot;CompA&quot; }</td>
</tr>
<tr>
<td>feature</td>
<td>F2</td>
<td></td>
<td>{ owningComponent=&quot;CompB&quot; }</td>
</tr>
<tr>
<td>feature</td>
<td>F3</td>
<td></td>
<td>{ owningComponent=&quot;CompA&quot; }</td>
</tr>
</tbody>
</table>

shown in Figure 3.1 are shown in Table 3.3. This table demonstrates how the binding of parameters of declaration reference clauses works.

In this example the $\text{featParent}$ relationship is declared with the parameter names “C” and “F” and the clause that references this declaration names the parameters “owningComponent” and “X”.

Each produced $\text{feature}$ fact has its identifier, named $X$, bound to the value of the second parameter of the corresponding $\text{featParent}$ fact, as shown in Table 3.3. Additionally, because the first parameter in the declaration reference clause is named $\text{owningComponent}$, each produced $\text{feature}$ fact has an attribute named $\text{owningComponent}$ whose value matches the value of the first parameter of the corresponding $\text{featParent}$ fact that was used to produce it.

Lastly, the $\text{feature}$ facts generated by the original program (in Figure 3.2) had a $\text{label}$ attribute which corresponds to the $\text{userName}$ Attribute of the corresponding XM* node. Because the $\text{feature}$ facts in this program are generated by references to extracted $\text{featParent}$ relationships (rather than from references to XM* data), the $\text{featParent}$ relationship must extract this label information if we want to include it in our $\text{feature}$ facts. This is why the $\text{featParent}$ relationship definition extracts the $\text{userName}$ Attribute of $\text{compFeatNode}$ nodes and binds it to an attribute $\text{fLabel}$ of the relation. Then, the declaration of $\text{feature}$ facts needs a means to access the attributes of referenced facts.

To this end, declaration reference clauses can have property lists; the syntax is identical to that of property lists in XM* reference clauses, except that the properties refer to attributes of the referenced fact rather than to XM* Attributes. Consider, for example, the property list $\{\text{fLabel}:\text{label}\}$ in the $\text{featParent}$ declaration reference clause in
the feature declaration. Because this property list belongs to a featParent declaration reference clause, the identifier fLabel on the left-hand side of the binding operator refers to attributes of previously declared featParent facts. Thus, this property list specifies that the fLabel attribute of each featParent fact, if present, should be used as the label attribute of the corresponding feature fact.

In the subsequent subsections, any newly discussed feature that applies to a property list of XM* reference clauses also applies similarly to a property list of declaration reference clauses. For the sake of brevity, each feature is discussed only with respect to XM* reference clauses, but the reader should recognize that the same feature applies also to the property list of a declaration reference clause (where references to XM* Attributes are replaced with references to attributes of the corresponding facts).

### 3.2.2 Constraint Clauses

A constraint clause restricts the value of a bound identifier, using one of three operators: equality (=), inequality (! =), or the like operator (∼=). At least one operand is the name of an attribute of a produced fact, and the other operand is either another attribute of a produced fact or a string literal. String literals are discussed in more detail in Section 3.2.5. The detailed behaviour of the three operators is as follows.

= An equality constraint evaluates to true if and only if its two operands have exactly the same value; it evaluates to false otherwise.

! = An inequality constraint evaluates to true if and only if its two operands have different values; it evaluates to false otherwise.

∼= A like constraint evaluates to true if and only if its right-hand operand is a string literal that expresses a regular expression pattern and that pattern matches the constraint’s left hand operand.

When a constraint clause is included in a mel declaration, the constraint is applied to each potential fact that could be produced as part of the declaration; if the constraint evaluates to true, then that fact remains in the result set, whereas if the constraint evaluates to false then that fact is removed from the result set. Figure 3.4 demonstrates a Model of a library’s contents. The mel program shown in Figure 3.5 demonstrates use of each of the three constraint operators. The declaration of entities of type SKing includes a constraint
<library>
   <book title="It" author="Stephen King" genre="Horror"/>
   <book title="GEB" author="Douglas Hofstadter" genre="Non-Fiction"/>
   <book title="Misery" author="Stephen King" genre="Horror"/>
   <book title="Great Expectations" author="Charles Dickens"/>
</library>

Figure 3.4: A Model of a library of books with a variety of authors and genres.

Figure 3.5: A mel program demonstrating the use of constraint clauses.

clause specifying that the A attribute of each result (i.e. the author Attribute) must be equal to the string “Stephen King”. The GNovels fact declaration includes the constraint clause specifying that the X parameter (the title Attribute of the book) must be like the string “[gG].*”. That is the GNovels fact set should only include books whose title matches at least one string specified by the regular expression pattern “[gG].*”. Lastly, the NotHorror declaration includes a constraint clause specifying that the G attribute of each result (i.e. the genre Attribute) does not equal the string “Horror”.

3.2.3 Compound Productive Clauses

Each mel declaration is defined after the turnstile symbol ‘ |- ’ by one or more clauses separated by commas. Each clause is either a productive clause (that adds facts to the result set) or a reductive clause (that removes facts from the result sets). Constraints are the most common type of reductive clause. As they are discussed above, in Section 3.2.2, this section focuses on the compounding of productive clauses. There are two well-formedness conditions that must be met by productive clauses in rules that have compound productive clauses:

1. Each productive clause must have at least one bound identifier (parameter or attribute) in common with another productive clause in the declaration.
Figure 3.6: An XML fragment with several Entity tags that do not each contain all the attributes other Entity tags do.

```
<nodes>
  <Entity id="ABC" mayExist="GAR" name="Hello"/>
  <Entity id="GAR" mustExist="HET"/>
  <Entity id="XYZ" mustExist="GAR" mayExist="ABC"/>
  <Entity id="TAB" mustExist="XYZ" mayExist="GAR"/>
  <Entity id="BOO" mustExist="ABC"/>
  <Entity id="KEL" mustExist="ABC" mayExist="HET"/>
</nodes>
```

2. It must be possible to form a connected graph, whose vertices represent the productive clauses of the declaration and whose edges join vertices whose corresponding productive clauses share bound identifiers.

Figure 3.7: A mel program that demonstrates the behaviour of compound clauses.

```
Decl(X) |- Entity{id:X, mustExist:necessary, mayExist:optional},
      Entity{id:necessary};
```

Figure 3.8: The disconnected graph formed by the productive clauses A{id:X, att:Y}, B{id:Y}, C{id:P, att:Q}, D{id:Q}, where each clause is given a vertex, and edges are drawn between vertices whose productive clauses share a bound identifier. The vertices have been labeled with the XM* tag of their corresponding clause, and the edges have been labeled with the bound identifier shared between the productive clauses.

Each productive clause in a mel generates potential facts; but in order to match the full declaration (1) the value of any attribute which occurs more than once in the set of clauses must match the value of all other occurrences of that attribute in the other clauses, and (2) the connectivity between productive clauses (through common attributes) must cover all of
Table 3.4: Results of applying the first clause of the declaration in Figure 3.7 to the XML fragment shown in Figure 3.6. Each row represents one fact that would be extracted and the columns are the values the identifiers would be bound to.

<table>
<thead>
<tr>
<th>X</th>
<th>optional</th>
<th>necessary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>GAR</td>
<td>HET</td>
</tr>
<tr>
<td>GAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XYZ</td>
<td>ABC</td>
<td>GAR</td>
</tr>
<tr>
<td>TAB</td>
<td>GAR</td>
<td>XYZ</td>
</tr>
<tr>
<td>BOO</td>
<td></td>
<td>ABC</td>
</tr>
<tr>
<td>KEL</td>
<td>HET</td>
<td>ABC</td>
</tr>
</tbody>
</table>

Table 3.5: Results of applying the second clause of the declaration in Figure 3.7 to the XML fragment shown in Figure 3.6. Each row represents one fact that would be extracted. As only one attribute, necessary, is extracted, there is only that value associated with each fact.

<table>
<thead>
<tr>
<th>necessary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
</tr>
<tr>
<td>GAR</td>
</tr>
<tr>
<td>XYZ</td>
</tr>
<tr>
<td>TAB</td>
</tr>
<tr>
<td>BOO</td>
</tr>
<tr>
<td>KEL</td>
</tr>
</tbody>
</table>

the productive clauses in the declaration. As a negative example the set of clauses \( A\{id:X, att:Y\}, B\{id:Y\}, C\{id:P, att:Q\}, D\{id:Q\} \) is invalid because the clauses referring to XML tags \( A \) and \( B \) have no connection to the other two clauses; the clauses referring to XML tags \( C \) and \( D \) are similarly disconnected from the other two clauses. That is, the graph that would be formed by these productive clauses is disconnected, as shown in Figure 3.8. The facts generated by compound clauses are effectively the result of computing the intersection over the matching attributes of the facts that would be generated by each productive clause individually if it were the lone clause of a declaration, and then filtering this fact set as per the reductive clauses. Each fact generated has attributes for all the attributes that were bound in any of the productive clauses of the definition.

Consider the result of applying the \( mel \) program in Figure 3.7 to the XML fragment shown in Figure 3.6. The resultant facts should be one \texttt{Decl} fact for each \texttt{Entity} XML
Table 3.6: Results of applying the `mel` program in Figure 3.7 to the XML fragment shown in Figure 3.6. This is the result of joining the two potential-fact sets shown in Tables 3.4 and 3.5.

<table>
<thead>
<tr>
<th>Fact Type</th>
<th>Entity</th>
<th>necessary</th>
<th>optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decl</td>
<td>XYZ</td>
<td>ABC</td>
<td>GAR</td>
</tr>
<tr>
<td>Decl</td>
<td>TAB</td>
<td>GAR</td>
<td>XYZ</td>
</tr>
<tr>
<td>Decl</td>
<td>BOO</td>
<td>ABC</td>
<td></td>
</tr>
<tr>
<td>Decl</td>
<td>KEL</td>
<td>ABC</td>
<td>HET</td>
</tr>
</tbody>
</table>

node that has a `mustExist` Attribute whose value is the same as an `id` Attribute of any `Entity` XML node in the Model.

To understand the behaviour of compound clauses, it helps to consider the potential facts that each clause would extract if it were the sole clause in the declaration. The potential facts that would be extracted by the first clause are shown in Table 3.4: there is one fact for each `Entity` node in the XML fragment, however not all entries have values for the attributes named `optional` and `necessary`.

The potential facts extracted by the second clause are shown in Table 3.5: again there is one fact for each `Entity` node in the XML fragment. However, the second clause binds the `id` Attribute to the `mustExist` attribute in its result, as opposed to the first clause which binds the `mustExist` Attribute to the attribute `necessary` in the result.

The resulting set is produced by computing the equivalent of an inner join of the two collections of potential facts, using the shared attribute’s name as the key to join on. Because the shared attribute is `necessary`, and because `necessary` is bound in the first and second clause to the XM* Attributes `mustExist` and `id` respectively, it follows that the program generates only facts from `Entity` nodes in the XM* whose `mustExist` Attribute corresponds to the `id` of any arbitrary `Entity` node in the XML. As a result there is no fact for the `Entity` nodes with `id` values of `ABC` or `GAR`. The `Entity` with the `id` value `ABC` has no `mustExist` Attribute and thus cannot be joined with any potential facts generated by the second clause. The `Entity` with the `id` value `GAR` has a `mustExist` Attribute, however the value of the attribute is `HET` which does not correspond to the `id` Attribute of any `Entity` node in the XML fragment, and thus cannot be joined with any of the potential results generated by the second clause. This generalizes to compound clauses with more two productive clauses by applying this process across all productive clauses computing the inner join on each of the shared attributes.
3.2.4 Optional Clauses

As discussed in Section 3.2.3, if a mel declaration contains multiple clauses then only facts that satisfy all of the clauses will be produced. Optional clauses allow for extracting specified information only when it is available. An optional clause in mel is a clause that is prefixed with a @ symbol. Optional clauses are particularly useful when trying to extract information about optional attributes related to entities or relationships introduced in another XM* node.

Consider the XML snippet in Figure 3.9, in which some component nodes have an associated owningTeam node that specifies the software team responsible for the component. In this example, the CompA node has an associated owningTeam node while the CompB node does not. Suppose that the user desires to extract all component nodes including the owning team as an attribute when available.

```
1 <SWModel>
2  <component id="CompA"/>
3  <component id="CompB"/>
4  <owningTeam name="DevGroup1" ownedComponent="CompA"/>
5 </SWModel>
```

Figure 3.9: A Model expressed in XML where Attributes are sometimes present. An owningTeam tag may exist associated with any given component tag, but is not required.

```
1 component(C) |- component{id:C},
2         @owningTeam{name:team, ownedComponent:C};
```

Figure 3.10: A mel program using an optional clause, specified by a prefixed @ symbol.

The sample mel program in Figure 3.10 includes both a normal clause (the first clause) and an optional clause (the second rule prefaced by ‘@’). The program extracts all entities and attribute bindings that match all the constraints in all normal clauses (in this case component nodes with Attribute id). The program’s optional rules specify additional component attributes to be extracted from owningTeam nodes whose ownedComponent Attribute value matches the component fact’s id value. The results of apply the mel program in Figure 3.10 to the Model in Figure 3.9 are shown in Table 3.7.
Table 3.7: Results of applying the mel Program in Figure 3.10 to the XML fragment in Figure 3.9.

<table>
<thead>
<tr>
<th>Fact Type</th>
<th>Entity</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity1</td>
<td>CompA</td>
<td>{ team=&quot;DevGroup1&quot; }</td>
</tr>
<tr>
<td>entity1</td>
<td>CompB</td>
<td></td>
</tr>
</tbody>
</table>

### 3.2.5 XM* Non-Basic Identifiers and String Literals

In some cases, an XM* Attribute might have a name containing non-alphanumeric characters which violates the grammar for mel identifiers, which are strictly alphanumeric strings. One common example seen in many Models is for an XM* Attribute name to include colon characters, which also happens to be the mel binding operator. In order to extract Attributes with such names, mel allows the left-hand side of a binding operator to be a string. However, mel prescribes different meanings to double-quote and single-quote enclosed strings, which is important to note. A double-quote enclosed string in mel is a string that represents either an identifier or a string literal, depending on the context. A single-quote enclosed string in mel always represents a string literal and never represents the value of an Attribute or attribute. A double-quote string has the following rules for its usage:

- If a double-quoted string appears as the left-hand operand of a binding operator, it refers to an XM* Attribute with the given name.
- If a double-quoted string appears as the right-hand operand of a binding operator, it refers to the name of an attribute of each produced fact.
- If a double-quoted string appears as either operand in a constraint, then it refers to the name of an attribute of each produced fact if an attribute with the given name exists, otherwise it is the string literal.

The results shown in Table 3.8 illustrate the difference between single-quote and double-quote strings, as well as the semantics of double-quote strings. To clarify, the results we consider each declaration in Figure 3.12 and the corresponding results it produces when applied to the XML fragment shown in Figure 3.11:

**Line 1** In this declaration, the double-quoted string "e:type", which is used as the left-hand operand of a binding operator, is a reference to the XM* Attribute with the
Figure 3.11: A simple XML fragment with entity tags with complex Attribute names (e.g., e:type) in XML.

```
<nodes>
  <entity id="X1" e:type="string" />
  <entity id="x2" e:type="int" />
</nodes>
```

Figure 3.12: A toy mel program illustrating the behaviour of double-quote and single-quote enclosed strings.

```
entity1(X) |- entity{id:X, "e:type":type};
entity2(X) |- entity{id:X, 'e:type':type};
entity3(X) |- entity{id:X, "e:type":type}, "type"!="int";
entity4(X) |- entity{id:X, "e:type":type}, 'type'!="int";
entity5(X) |- entity{id:X, "e:type":type}, "int"!="int";
entity6(X) |- entity{id:X, "e:type":"Fun Type"};
```

same name. As such, this usage binds the value of the XM* Attribute e:type to the attribute type of each produced entity. The result of this is shown in the first two rows of Table 3.8, where both entity1 facts X1 and X2 were extracted with their corresponding type.

Line 2 Because a single-quoted string always resolves to a literal value, the string ‘e:type’ binds the literal value e:type as the value of the type attribute in the result set. The result of this is shown in the third and fourth rows of Table 3.8, where both entity2 facts X1 and X2 have type attributes with value e:type.

Line 3 This declaration is similar to the one on Line 1, except it includes the constraint "type"!="int". In this scenario double-quoted strings are used as the operands to a constraint. As per the semantics of double-quoted strings in a constraint, the string "type" refers to the type attribute of the extracted entity because that attribute exists, and the string "int" refers to the literal string value "int" because no such attribute exists in the extracted entity. Note that this constraint could have equivalently been written as type!="int", which more explicitly specifies that type is an attribute in the result and ‘int’ is a literal value. This constraint disallows any results where the type attribute has the value "int". The result of this is shown in the fifth row of Table 3.8, where only one entity3 fact is extracted: that of X1
Table 3.8: Results of applying the mel Program in Figure 3.12 to the XML fragment in Figure 3.11

<table>
<thead>
<tr>
<th>Fact Type</th>
<th>Entity</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity1</td>
<td>X1</td>
<td>{ type=&quot;string&quot; }</td>
</tr>
<tr>
<td>entity1</td>
<td>X2</td>
<td>{ type=&quot;int&quot; }</td>
</tr>
<tr>
<td>entity2</td>
<td>X1</td>
<td>{ type=&quot;e:type&quot; }</td>
</tr>
<tr>
<td>entity2</td>
<td>X2</td>
<td>{ type=&quot;e:type&quot; }</td>
</tr>
<tr>
<td>entity3</td>
<td>X1</td>
<td>{ type=&quot;string&quot; }</td>
</tr>
<tr>
<td>entity4</td>
<td>X1</td>
<td>{ type=&quot;string&quot; }</td>
</tr>
<tr>
<td>entity4</td>
<td>X2</td>
<td>{ type=&quot;int&quot; }</td>
</tr>
<tr>
<td>entity6</td>
<td>X1</td>
<td>{ Fun Type=&quot;string&quot;}</td>
</tr>
<tr>
<td>entity6</td>
<td>X2</td>
<td>{ Fun Type=&quot;int&quot; }</td>
</tr>
</tbody>
</table>

whose type attribute has the value "string" not "int".

Line 4 This declaration is similar to the one on Line 1, except it includes a constraint ‘type’!="int". A single-quoted string always refers to a string literal. Moreover, the double-quoted string "int" resolves to the literal string value because no attribute in each produced entity is bound to the name int. Thus, this constraint is trivially true constraint, stating that the string ‘type’ must not equal the string ‘int’. Because the constraint evaluates to this declaration is equivalent to the one on Line 1. The result of this is shown in the sixth and seventh rows of Table 3.8, where both X1 and X2 are extracted as entity4 facts, despite what one might confuse for a constraint on the type attribute.

Line 5 This declaration is similar to the one on Line 1, except it includes a constraint "int"!="int". In this case, both double-quoted strings refer to their literal string values because there is no attribute with the name int in each produced entity. Thus, this constraint is trivially false, stating that the string ‘int’ must not equal the string ‘int’. Because the constraint is false, no entity5 facts may be extracted as they all violate the clause. The result of this is shown in Table 3.8 through the absence of any entity5 facts.

Line 6 This declaration is similar to the one on Line 1, except instead of binding the e:type Attribute to the attribute "type" it binds it to the attribute "Fun Type". The result of this is shown in rows eight and nine of Table 3.8, where the attribute of each entity6 fact has the name Fun Type.
Table 3.9: Results of applying the mel Program in Figure 3.14 to the XML fragment in Figure 3.13

<table>
<thead>
<tr>
<th>Fact Type</th>
<th>Entity</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>foo</td>
<td>{ retType=&quot;int&quot; prototype=&quot;int foo(int, int);&quot; }</td>
</tr>
</tbody>
</table>

The dual roles that double-quoted strings can play lead to the seemingly unintuitive semantics described above. As such, the suggested best practice is that double-quoted strings be used only as the left-hand operand of a binding operator when made necessary by the XM* document that the program is applied to. Any other time strings are used, they should be restricted to mean literal values and thus be delimited by single quotes. However, if a mel programmer desires that an attribute in the factbase have a name that is not a valid mel identifier (i.e. any identifier that is not strictly alphanumeric), then they can use a double-quoted string in as the right-hand operand of a binding operator (e.g., as in line 6 of Figure 3.12 to bind an attribute to name "Fun Type" which includes a space). If an attribute's name is non-alphanumeric, then it must be referred to in constraints using strings delimited with double quotes.

Additionally, one may question the benefit of having the ability to bind attributes to string literals as is done in line 2 of Figure 3.12. While the ability to bind attributes to string literals may not appear very useful on its own, it has some limited uses and in Section 3.2.9 we discuss an additional feature of mel that makes effective use of this ability.

3.2.6 Contents of an XM* Node

An XM* node can have content that comprises either nested XM* nodes or string data. Because an XM* node’s content can be string data, which may be pertinent to a user’s needs, mel provides a feature mel.contents that refers to the string-data contents of an XM* node. mel.contents is used as the left-hand operand of a binding operator (as shown in Figure 3.14) to bind the value of the string contents of the given XM* node to the attribute being bound. Effectively mel.contents can be considered the pseudo-Attribute name of the contents of an XM* node. Table 3.9 shows that the extracted prototype attribute of the extract Func has the value of the string contents of the function node shown in Figure 3.13.

3.2.7 Extracting a Substring of an XM* Attribute

36
Figure 3.13: A sample XML node that contains string data.

```xml
<function id="foo" return="int">
  int foo(int, int);
</function>
```

Figure 3.14: A simple mel program that makes use of the `mel.contents` pseudo-Attribute for accessing the string contents of an XM* node.

```mel
Func(X) |- function{id:X, return:retType, mel.contents:prototype};
```

Figure 3.15: A sample XML fragment that shows reference Attributes that have the referenced identifier prefixed with a # character.

```xml
<SWModel>
  <component id="CompA"/>
  <component id="CompB"/>
  <component id="CompC"/>
  <compFeatNode id="F1" userName="Feature 1"
      component="#CompA"/>
  <compFeatNode id="F2" userName="Feature 2"
      component="#CompB"/>
  <compFeatNode id="F3" userName="Feature 3"
      component="#CompA"/>
</SWModel>
```

Figure 3.16: A mel program that makes use of Attribute modifiers to specify a substring of an XM* Attribute to extract.

```mel
featParent(C,F) |- compFeatNode{id:F, component["#(.*)":C]};
```

Figure 3.16: A mel program that makes use of Attribute modifiers to specify a substring of an XM* Attribute to extract.
Table 3.10: Results of applying the mel Program in Figure 3.16 to the XML fragment in Figure 3.15

<table>
<thead>
<tr>
<th>Fact Type</th>
<th>First Param</th>
<th>Second Param</th>
</tr>
</thead>
<tbody>
<tr>
<td>featParent</td>
<td>CompA</td>
<td>F1</td>
</tr>
<tr>
<td>featParent</td>
<td>CompB</td>
<td>F2</td>
</tr>
<tr>
<td>featParent</td>
<td>CompA</td>
<td>F3</td>
</tr>
</tbody>
</table>

Sometimes the value of an XML Attribute contains extraneous characters. A common example are Attributes prefixed with additional characters. Figure 3.15 shows an XML fragment where compFeatNode nodes reference their owning component through the Attribute component, however the values of the Attributes are prefixed with a character #. Ideally, the values of extracted identifiers strip away these extra characters. mel provides a feature called an Attribute modifier which is a regular expression pattern with at least one capturing group. A capturing group in a regular expression pattern is any portion of the string enclosed in parentheses. The value the first capturing group produces when the regular expression is applied to a string is the value that is bound to the attribute. An Attribute modifier is applied to an Attribute by suffixing the Attribute name with an Attribute modifier, in which the regular expression pattern is specified as a string enclosed in square brackets on the left-hand side of a binding operator: \texttt{X[Regex]:Y}. The string \texttt{Regex} represents a regular expression pattern with at least one capturing group. For example, if a binding is \texttt{X["abc([123]*)(.*)"]:Y} and the Attribute \texttt{X} has a value of "12abc323671xyz", then the attribute \texttt{Y} will be bound to the value "323", because the first capturing group used in the Attribute modifier matches any string that comprises only the digits 1, 2, and 3 that immediately follows the string \texttt{abc}.

The syntax and meta-characters of regular expressions that mel supports are those as defined by the ISO C++ 2017 standard [34] and implemented by the the compiler g++ as part of the GNU compiler collection [21].

The featParent declaration in Figure 3.16 includes a \texttt{compFeatNode} clause that modifies the component Attribute with the Attribute modifier "#(.*)" and binds the result to the parameter \texttt{C}. The regular expression pattern "#(.*)" matches a string that begins with the character \# and is then followed by any string, denoted by ".*", which is enclosed in a capturing group. As such, it is the value of the arbitrary string that follows the character \# in the Attribute string that is bound to the parameter \texttt{C}. 

38
3.2.8 Constraints on XM* Attributes.

A user may want to extract a subset of the matching XM* nodes in a Model. Constraints on result attributes are discussed in Section 3.2.2, however these constraints are written on extracted attributes. It is possible a user may instead want to constrain matching XM* node instances based on their Attribute values without necessarily binding those values to attributes in the result. mel supports this concept via Attribute requirements, which specify constraints on an XM* tag’s Attribute values that must be met in order for an instance of the tag to match the fact type declaration. To differentiate between constraint clauses and constraints placed on XM* Attributes, the term constraint is used for constraint clauses and the term requirement is used for requirements placed on XM* Attributes included within an XM* reference clause. For example, the declarations in the program shown in Figure 3.5 can be modified so as not to extract the author and genre of books while still extracting facts related to only the desired subset of XM* nodes.

Requirements can make use of any of the constraint operators introduced in Section 3.2.2. However, when these operators are used within a requirement, the left-hand operand always represents the name of an Attribute in the XM* node and the right-hand operand must always be a string literal. Thus, the semantics of each operator needs to account for the possibility that its left-hand operand refers to an Attribute that does not exist in the XML node:

= Used in a requirement, an equality operation evaluates to true if and only if the value its left-hand operand is exactly equal to the value of its right-hand operand. If the XM* Attribute referred to by the left-hand operand does not exist in the node, the operation evaluates to false, as no string (even the empty string) is equal to a non-existent string.

!= Used in a requirement, an inequality operation evaluates to true if and only if the value its left-hand operand is not exactly equal to the value of its right-hand operand. If
the XM* Attribute referred to by the left-hand operand does not exist in the node, then the operation evaluates to true, as a non-existent string is trivially not equal to any existing string.

\[ \sim = \]

Used in a requirement, a “like” operation evaluates to true if and only if the value of its left-hand operand matches the regular expression pattern of its right-hand operand. If the XM* Attribute referred to by the left-hand operand does not exist in the node, then the operation evaluates to false, as no string (even the empty string) exists within a non-existent string.

In addition to the equality, inequality, and like operators, requirements may also specify Attributes that must or must not appear in the given XM* node in order for it to be a valid fact. These requirements are written using the two unary \texttt{mel} functions named \texttt{mel.exists} and \texttt{mel.nexists}. Such a requirement takes the form of \texttt{mel.exists(attrName)} or \texttt{mel.nexists(attrName)}, where the former evaluates to true if and only if the XM* Attribute named \texttt{attrName} exists in the XM* node, whereas the latter has the inverse semantics.

\begin{verbatim}
component(C) |- compFeatNode{component:C};
feature(X) |- compFeatNode{id:X, mel.exists(userName),
userName:label};
featParent(C,F) |- compFeatNode{component:C, id:F}, feature(F);
^sibling(F,G) |- featParent(C,F), featParent(C,G), F!=G;
\end{verbatim}

Figure 3.18: An alternative to the \texttt{mel} program shown in Figure 3.2 that uses a requirement in the \texttt{feature} declaration to specify that matching \texttt{compFeatNode} XM* nodes must have a \texttt{userName} Attribute.

As an example, the program shown in Figure 3.18 uses the \texttt{mel.exists} function. This program behaves very similarly to the \texttt{mel} program presented in Figure 3.2, with the exception that it extracts \texttt{feature} facts only from \texttt{compFeatNode} nodes that have a \texttt{userName} Attribute. That is, if this program were applied to the XML fragment shown in Figure 3.1 it would not extract the \texttt{feature} fact \texttt{F2} because the \texttt{compFeatNode} node that defines it does not have a \texttt{userName} Attribute. Additionally, the clause \texttt{feature(F)} is added to the \texttt{featParent} declaration to ensure that \texttt{featParent} relationships are extracted only between \texttt{feature} and \texttt{component} facts that already exist in the produced factbase.
An alternative solution, to ensure that featParent relationships are formed only with feature facts that exist would be to include the mel.exists(userName) requirement within compFeatNode clause of the featParent declaration.

3.2.9 The Ternary Operator

The beginning of Section 3.2 specified that property lists may contain attribute bindings or requirements. So far, the attribute bindings shown have only been of the form X:Y where X is either a reference to an Attribute or a string literal. It has not been discussed yet how a user may conditionally bind an attribute to different values.

Consider the Model shown in Figure 3.19. Suppose that the user desires to extract from this Model an entity for each animal that includes as an attribute the sound the animal makes. If there is not a sound Attribute in the XM* for a given animal XM* node, the user would like the sound attribute of the corresponding produced fact to have the value “Who knows?”.

mel provides a ternary operator with the following form

```
requirement ? valIfTrue > valIfFalse
```

where requirement is any valid requirement as described in Section 3.2.8, and valIfTrue and valIfFalse must each be one of:

- A string literal, in single quotes, which evaluates to the exact value of the string literal.
- An identifier corresponding to an XM* Attribute, evaluating to the value of that Attribute. We will see that a ternary always appears as a left-hand operand of a binding operator; thus double-quoted strings appearing in a ternary operator are always interpreted as the name of an Attribute, as discussed in Section 3.2.5.
• A nested ternary, which evaluates as per the semantics defined below.

A ternary expression may appear only as the left-hand operand of a binding operator, or nested within another ternary operation. The value of a ternary operation is the value of its \texttt{valIfTrue} expression if the given requirement is true and is the value of its \texttt{valIfFalse} expression otherwise.

\begin{verbatim}
Animal(X) |- animal{id:X, mel.exists(sound) ? sound > 'Who knows?':noise};
\end{verbatim}

Figure 3.20: A \texttt{mel} program that demonstrates the use of the ternary operator.

The \texttt{mel} program shown in Figure 3.20 shows the use of a ternary operator to conditionally bind the \texttt{noise} attribute of each \texttt{Animal} fact to either the string literal ‘Who knows?’ or to the value of the \texttt{sound} Attribute of the corresponding \texttt{animal} XM* node. The result set of applying this program to the XML fragment shown in Figure 3.19 is three \texttt{Animal} facts each with a \texttt{noise} attribute. The \texttt{Cat} fact has a \texttt{noise} attribute of "meow", the \texttt{Dog} fact has a \texttt{noise} attribute of "bark", and the \texttt{Fox} fact has a \texttt{noise} attribute of "Who knows?". The value of the \texttt{noise} attribute of the \texttt{Fox} fact is a result of the ternary operator producing the string "Who Knows?" because the XML node for \texttt{Fox} has no \texttt{sound} Attribute, making the requirement \texttt{mel.exists(sound)} evaluate to false.

\begin{verbatim}
Animal(X) |- animal{id:X, mel.exists(sound) ?
    sound='bark' ? 'ruff' > sound
    > 'Who knows?':noise};
\end{verbatim}

Figure 3.21: A \texttt{mel} program that demonstrates the use of nested ternary operators.

An example of nested ternary operators is shown in Figure 3.21. In this program, the \texttt{noise} attribute of each extracted \texttt{Animal} fact depends not only on whether the \texttt{sound} Attribute exists, but also on its value if it does exist. If the \texttt{sound} Attribute exists, then it is then compared to the string literal "bark"; if the values are equal then the expression produces value "ruff"; if the values are not equal then the expression produces the value of the Attribute. Alternatively, if the \texttt{sound} Attribute does not exist at all, then the expression produces value "Who knows?". Thus, the difference in the values produced by the programs shown in Figures 3.20 and 3.21 when applied to the XML fragment shown in Figure 3.19 is that the former extracts the \texttt{noise} attribute of the \texttt{Dog} fact as "bark", whereas the latter extracts the same attribute as "ruff".

42
3.2.10 XM* Trees of Elements

So far, the clauses specifying what information to extract from the XM* input document have been limited to XM* reference clauses, which match all corresponding nodes regardless of their locations in the XM* input. However, the tree structure of an XM* document is commonly used to encode meaningful information about the Model it represents. Thus, it is necessary to be able to specify not only individual nodes but patterns of nodes to consider for extraction. mel provides operators for specifying parental (\(\rightarrow\)) or ancestral (\(\Rightarrow\)) relationships between XM* nodes. These operators define paths through the XM* node structure, thus they are collectively referred to as path operators. The path operators can be used to write clauses to specify hierarchical patterns of nodes which must appear in the XM* for a fact to be generated.

**Parent Operator**

The most common relationship among XM* nodes is the *parent* relationship, in which a child XM* node is nested within a parent node. Figure 3.22 shows an XML fragment for a Model that comprises features, components, and an ownership relationship between components and features. A feature’s owning component is represented in the XML fragment by containing each feature node within its owning component node. Thus, this XML fragment represents a Model in which features F1 and F3 belong to component CompA, and feature F2 that belongs to component CompB.

```
<SWModel>
  <component id="CompA">
    <feature id="F1" userName="Feature 1"/>
    <feature id="F3" userName="Feature 3"/>
  </component>
  <component id="CompB">
    <feature id="F2" userName="Feature 2"/>
  </component>
</SWModel>
```

Figure 3.22: An XML fragment that encodes relevant Model information as parental relationships between nodes.
Given the XML fragment in Figure 3.22, a mel declaration to extract a parental relationship featParent between components and their owned features could be written as:

\[
\text{featParent}(C,F) \mid \text{component\{id:C\}->compFeatNode\{id:F\}};
\]

This declaration matches only instances of XM\* component tags that have an XM\* Attribute id and have a child tag compFeatNode with its own XM\* Attribute id. The right-hand operand of a path operator may also be a collection of descendants that each must exist, rather than a single descendant.

\[
\text{Sibling}(F,G) \mid \text{component\{}\text{-}[\text{compFeatNode\{id:F\}},
\text{compFeatNode\{id:G\}}], F!=G;
\]

Figure 3.23: Example use of the mel parent operator.

The sibling relationship between feature facts could be written using path operators as shown in Figure 3.23. The right-hand operand of the parent operator in this example demonstrates how square brackets can be used to specify a collection of descendants that each must exist underneath the left-hand operand of the operator. Thus, the XM\* path clause in the above declaration specifies to look for component tags that have multiple distinct compFeatNode tags as child nodes. When this clause is applied to the XML fragment in Figure 3.22, only combinations of features F1 and F3 are considered because feature F2 does not reside underneath the same component node as F1 and F3 and the constraint F!=G disallows reflexive results that relate features F1, F2, or F3 to themselves. Thus, the only produced relationships are shown in Table 3.11.

Table 3.11: Results of applying the mel Program in Figure 3.23 to the XML fragment in Figure 3.22

<table>
<thead>
<tr>
<th>Relationship Type</th>
<th>First Parameter</th>
<th>Second Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>sibling</td>
<td>F1</td>
<td>F3</td>
</tr>
<tr>
<td>sibling</td>
<td>F3</td>
<td>F1</td>
</tr>
</tbody>
</table>

An important observation of this sibling declaration is that the reference to tags of type component has an empty property list. This example demonstrates how the sibling
Figure 3.24: An XML fragment that acts as an exemplar for nested relationships.

relationship could be extracted even if the component tags have no unique identifier Attributes.

Ancestor Operator

Sometimes a relationship of interest may not be a direct parental relationship but rather an ancestral relationship. The ancestor operator specifies descendental nodes that are nested arbitrarily deep. Consider the XML snippet of nested nodes shown in Figure 3.24.

In this example, we have three features all contained within component CompA but at different levels of nesting in the XML structure:

- **Feature 1** is a directly child of CompA.
- **Feature 2** is nested in a featureCollection tag, which itself is nested underneath an ownedFeatureSet tag, which is finally a child of the component CompA.
- **Feature 3** is nested only underneath a featureSet tag.

Because the ancestor operator matches XM* nodes of arbitrary depth of nesting, the relationship between components and their respective features can be extracted with a single mel declaration:

```mel
def featureOf(C,F) | - component{id:C}=>compFeatNode{id:F};```
Specifying Paths

The path operators are so-named because they can be combined to specify more complicated paths in the XM* structure, than just parent-child or ancestor-descendent relationships.

Given the XML snippet shown in Figure 3.24, consider the case that one wants to extract only features that are contained within an `ownedFeatureSet` node that itself is a child of a `component` node:

1. `featureOf(C,F) |- component{id:C}`
2. `->ownedFeatureSet{}`
3. `=>compFeatNode{id:F};`

The above declaration of `featureOf` combines the use of a parent operator with an ancestor operator to specify a path expression. Specifically the clause looks for `component` nodes that have a child node with an `ownedFeatureSet` tag, then searches for any `compFeatNode` nodes that are descendants (at any depth) of that `ownedFeatureSet` node.

3.2.11 Elide Results of Intermediate Rules.

The `mel` user may want to declare and extract facts that are not output to the result set. For example, some facts may be extracted for the sole purpose of easing the definition of more complicated facts. `mel` supports intermediate declarations, which are distinguished with the prefix ".", whose results are elided from the result set. As an example, suppose that the `mel` user would like to extract information about sibling features that lie in the same `component`, but not have intermediate facts about features and their parent relations appear in the final result set. If the `featParent` declaration in Figure 3.2 were replaced by the following declaration

1. `featParent(C,F) |- compFeatNode{component:C, id:F};`

then the results of applying this modified `mel` program to the XML Model from Figure 3.1 would be the results in Table 3.2, but without the contents of rows 7-9.
program ::= decl+
  decl ::= [mods] name ('params') ['-' clauses ']' 
  mods ::= ( '.' | '^' | '$' )*
  params ::= name [';' name ]
  clauses ::= clause ( ';' clause )*
  clause ::= ['@'] xmlRef | ['@'] declRef | ['@'] path | constraint | ['@'] fnCall
xmlRef ::= id ('{ propList '}')
  declRef ::= name ('params') '{' propList '}'
  path ::= ['@'] xmlRef ('->' | '=>') paths
constraint ::= ( id [strMod] | literal ) op ( id [strMod] | literal )
fnCall ::= ('mel.parent'|'mel.ancestor') op ( xmlRef ',' xmlRef ')
propList ::= binding | attreq | propList ';' ( binding | attreq )
attreq ::= lvalue [strMod] op literal | eReq '{' lvalue '}'
eReq ::= 'mel.exists' | 'mel.nexists'
binding ::= lvalue [strMod] ':' id | ternary ':' id
ternary ::= attreq '?' ternOption '>' ternOption
ternOption ::= ternary | lvalue [strMod] | literalStr
op ::= '=' | '!=' | '~='
paths ::= ['@'] xmlRef | path | ['@'] listofPaths '
listofPaths ::= ( ['@'] xmlRef | path ) ( ',' ['@'] xmlRef | path )*
lvalue ::= id | 'mel.contents'
strMod ::= [' literal ']
  id ::= name | string
name ::= ALPHA ( ALPHA | DIGIT | '_')*
literal ::= string | literalStr
string ::= "" CHAR* '"'
literalStr ::= '"' CHAR* '"'

Figure 3.25: Grammar for mel programs: Non-terminal symbols are italicized and TERMINAL symbols are in uppercase. Literals are enclosed in single quotes. "|" denotes alternation, "["..."]" encloses optional symbols, and "("...")" encloses a grouping of symbols. "*" denotes zero or more repetitions of the previous symbol or grouping, and "+" denotes one or more repetitions of the previous symbol or grouping.
3.3 Grammar of mel

The full grammar for mel is provided in Figure 3.25. A program is a sequence of declarations. Each declaration defines either an entity type (with one parameter) or a relationship type (with two parameters). Declarations can be prefaced with modifiers which modify their behaviour. A declaration prefaced with modifier ‘.’ denotes an elided fact type whose instances will not be part of the output; but the instances can be used to identify other extracted facts that are part of the output. Prefacing a declaration with the modifier ‘~’ declares that the relationship type cannot be commutative (i.e., cannot relate an entity X to entity Y as well as relate Y to X). In such cases, mel removes the second of the commutative instances from the result set. Lastly, prefacing a declaration with the modifier ‘$’ specifies that duplicate facts (i.e. facts with the same values for their parameters) should not be extracted, in this case mel selects the first such found fact.

Each declaration is followed by a series of clauses that correspond to the four types of clauses discussed earlier in Section 3.2.

3.4 mint the mel Interpreter

As with any programming language there needs to be a method of executing programs written in the language. The solution to executing mel programs is the mel interpreter mint. mint is implemented in C++ using GNU Bison [44] for parsing, Fast Lexical Analyzer Generator (Flex) [44] for tokenizing, and the RapidXML library [36] for parsing XML text. C++ was chosen for its highly efficient code, access to the aforementioned parsing libraries, and because it is more architecture-independent than assembly.

3.4.1 Usage of mint

mel programs are executed by providing them as input to the mel interpreter (mint) along with the additional input(s) of the XM* file(s) to process. A mel program and the XM* file are respectively analogous to the intensional database (IDB) and extensional database (EDB) of a Datalog program. mint is used as a simple command-line interface tool whose usage is of the form

$ mint <melProgram> <XM*File> [XM*Files]... [-n] [-o output]
where the first argument is the \texttt{mel} program the user has written, the second argument is an XM* file to be extracted from, and the user can optionally provide additional XM* files if the Model they are extracting from comprises several files. The optional flag (-n) denotes that the output format is comma-separated value (CSV) files, one for all the nodes and one for all the relationships, by default \texttt{nodes.csv} and \texttt{edges.csv} in the current working directory respectively. The other optional (-o) argument specifies the name of the file to which the resultant factbase is output. If -n and -o \texttt{fileName} are used in conjunction then the two output CSV files corresponding to edges and nodes respectively will be \texttt{fileNameEdges.csv} and \texttt{fileNameNodes.csv}. If neither of the flags are used then \texttt{mint} outputs the resultant factbase to the standard output stream.
Chapter 4

Evaluation of mel

In this chapter, we discuss the evaluations performed on mel and the methods of the experiments performed. The evaluation of mel focuses on two criteria:

1. The extent to which mel generalizes to XM*-based representations of Models and to desired extractions

2. The effectiveness of mel in easing the creation of new fact extractors for Models

In order to test mel against a wide variety of Model types, we looked for publicly available Models that exercise complex features in their respective Modelling languages; that were generated by Modelling tools commonly used in education, research, or industry [1, 31, 37, 42]; and whose XM* representations varied from each other. Finding suitable Models for the study was a challenge because of the dearth of publicly available sophisticated Models for a majority of the Model types. Researchers Hebig, Ho-Quang, Chaudron, Robles and Fernandez [25] provide a solid database for UML diagrams, although the vast majority of the Models available from, the database are UML Class Diagrams. In the end, we sourced Models from sample Models provided with Modelling tools and from public GitHub repositories.

The Model types considered in this evaluation are UML StateMachine and Class diagrams [19], Simulink Block diagrams [49], Arcadia Logical Architecture diagrams [67], and Feature Models [43]. With the exception of a particularly structured feature constraints in Feature Models, we found that mel was sufficiently expressive to generalize to all five Model types. The data sets used for these studies are available in the repository https://github.com/Roshack/ModelsUsedWithmel which includes for each Model
an image of at least one diagram from the Model, the XM* representation of the Model, and the factbase produced when the corresponding mel-generated extractor was applied to it. Additionally, our studies on the effectiveness of mel in easing the creation of new fact extractors produced generally favourable results in terms of development time and succinctness of the programs developed.

4.1 Scope of Applicability of mel

Our first study evaluated whether mel is expressive enough to extract facts of interest from different Model types with varying XM* representations. For each Model type we discuss the types of facts extracted, the different editors and representations that were considered, and the correctness tests of the extractors written in mel when applied to the Models of that type.

In order to test the correctness of both the mel-generated extractors, and the interpreter mint itself, we sampled facts for each Model considered:

1. Twenty facts (or all facts, if there were fewer than twenty in the Model) were randomly selected from the visual depiction of each Model of the Model type being assessed. These 20 facts were then confirmed to exist in the factbase produced by applying the mel-generated extractor to the XM* representation of the Model.

2. Twenty facts were randomly selected from the factbase produced by applying the mel-generated extractor to the XM* representation of a Model. These facts were then confirmed to exist in the XM* of the Model and in the visual depiction of the Model.

The above process was applied to all the Models of each type considered. The remainder of this subsection will discuss the different Model types considered. For each Model type, we discuss (1) the information that must be extracted from the Model to produce a factbase that represents a complete model of a Model of that type; (2) the number of Models considered, the editors we used to produce them and an overview of the differences in the textual representations between the editors; (3) the mel-generated extractors written for each Model/editor pair; and lastly (4) the results of applying the mel-generated extractors to the Models.
Table 4.1: Necessary facts for extraction from UML StateMachine diagrams.

<table>
<thead>
<tr>
<th>Entities</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Machines</td>
<td>name</td>
</tr>
<tr>
<td>States</td>
<td>name; entry, exit, and internal actions/guards</td>
</tr>
<tr>
<td>Regions types</td>
<td>name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationships</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub state</td>
<td></td>
</tr>
<tr>
<td>Containment of state by region</td>
<td></td>
</tr>
<tr>
<td>Containment of region by state</td>
<td></td>
</tr>
<tr>
<td>Concurrent regions</td>
<td></td>
</tr>
<tr>
<td>Transitions</td>
<td>triggers, guards, and actions</td>
</tr>
</tbody>
</table>

4.1.1 UML StateMachine Diagrams & Statechart Diagrams

UML StateMachine diagrams are adaptations of the statechart formalism proposed in 1987 by Harel [23]. Statecharts extend traditional state machines with hierarchical states, typed variables, concurrent submachines, and the communication of events between concurrent submachines. We differentiate between the mathematical concept of a state machine and the class of UML StateMachine diagrams through the capitalization of the latter. StateMachine diagrams were selected for this study because (1) the data they model is distinct from other Model types considered, and (2) they feature prominently in academic works across a wide variety of applications [14, 39, 66].

Information to be Extracted

For this study, we identified several key pieces of information to extract as facts from UML StateMachine diagrams, shown in Table 4.1 This set of facts is our attempt at extracting a comprehensive model of a UML StateMachine diagram without omitting any details. Thus, an extractor would need to be able to extract all of the information listed in Table 4.1 to be considered successful.

Models & Tools Considered

As subject StateMachine diagrams, we considered Models generated by either IBM Rhapsody (version 8.4) [32] or YAKINDU Statechart Tools (SCT) [35]. Rhapsody was selected
because it is used in industry and studied in research on Model-driven engineering [37]. YAKINDU was selected because it serializes its Model representations quite differently from how Rhapsody serializes Models; also YAKINDU provides a plethora of thirty sample Models.

The UML StateMachine shown in Figure 4.1 is for a component of a hypothetical home alarm system written in C++. It comes from a sample Rhapsody project provided as part of the IBM Rhapsody version 8.4. A YAKINDU produced model is also shown in Figure 4.2 which includes much more detail. For example, the YAKINDU Model includes examples of actions that occur while residing in, entering, or exiting a state.

Snippets of XML from these two Models are shown in Figures 4.3 and 4.4 respectively. While the two Model representations share similarities, as do most representations of the same Model type, they also have some important differences. One major difference is in
Figure 4.2: UML StateMachine diagram of Harel’s statechart for a digital watch [23], produced using YAKINDU SCT.

how the two editors serialize data about transition triggers, guards, and actions as well as state actions and their corresponding guards. Figure 4.3 shows how Rhapsody encodes the transition’s triggers, guards, and actions through a particular set of relationships:

**Triggers** are represented by a trigger node that is nested underneath its corresponding transition node. The trigger node contains the identifier of a packagedElement node. The corresponding packagedElement node has an xsi:type of uml:SignalEvent that is found elsewhere in the XML hierarchy. The uml:SignalEvent has a signal Attribute whose value corresponds to the identifier of a different packagedElement whose xsi:type is uml:Signal. Within that second packagedElement, the value of the trigger is stored within the name Attribute.

**Guards** are represented as the value Attribute of a specification node, which resides underneath a ownedRule node, which in turn resides underneath the node for the transition in question.
Figure 4.3: Snippet of the XMI representation of the Rhapsody StateMachine Model shown in Figure 4.1. Identifiers have been abbreviated and non-referenced Attributes have been omitted.

**Effects** are represented as the contents of a **body** node, which resides underneath an **effect** node, which in turn resides underneath the node for the **transition** in question.

As shown in Figure 4.4, YAKINDU instead chooses to represent a transition’s trigger, guard, and action all together as a single string. Each transition is stored as an **outgoingTransition** node whose **specification** Attribute contains all the information about that transition other than its source and target states. The **specification** Attribute takes the form of the literal value displayed in the Model including line feed characters. This XML snippet also shows how all of a state’s actions are also encoded as one large string.

Rhapsody’s and YAKINDU’s representations place different requirements on the respective extractors that can analyze them. An extractor for Rhapsody diagrams must be able to sufficiently specify and resolve relational connections to pluck out a transition’s trigger, guard, and action. Conversely, an extractor for YAKINDU Models need not resolve any relation to find a trigger’s information, but be able to parse relevant substrings from a string Attribute.
Figure 4.4: Snippet of the XMI representation of the YAKINDU StateMachine Model shown in Figure 4.2. Identifiers have been abbreviated, non-referenced Attributes have been omitted, and white space has been adjusted to improve readability. YAKINDU stores all of a transition or state's triggers, guards, and actions as one contiguous string.
Overall, a total of eleven StateMachine diagrams were used in evaluating \texttt{mel}. Ten of these were generated using YAKINDU and the remaining one was generated using Rhapsody.

\texttt{mel} Extractors for Rhapsody and YAKINDU Models

The \texttt{mel} extractors written for StateMachine diagrams produced by Rhapsody and YAKINDU can be seen in Figures 4.5 and 4.6, respectively. The structures of these two programs further illustrates the differences in how Rhapsody and YAKINDU encode their Models. Examining the \texttt{Transition} declaration in Figure 4.5 reveals that extracting all the relevant information takes eight compound clauses, five of which involve parent operators - meaning that at least \textit{thirteen} nodes need to be resolved and connected to generate each \texttt{Transition} fact. The \texttt{Transition} declaration of Figure 4.6 represents the other extreme: it is defined using a single clause involving a parent operator, meaning that only two nodes need to be resolved and connected to generate each \texttt{Transition} fact. However, this \texttt{Transition} declaration makes use of several non-trivial regular expressions to resolve the attributes it would like to produce for each fact.

The two extremes of Rhapsody and YAKINDU Model representations demonstrate that the expressiveness of \texttt{mel} is sufficient both to resolve complex interconnected relationships as well as resolve complex mappings of XML Attributes to fact attributes.
Figure 4.5: mel program for extracting facts from a UML StateMachine diagram created with Rhapsody.
Figure 4.6: mel program for extracting facts from a UML StateMachine diagram created with YAKINDU.
Table 4.2: Summary of the StateMachine Models considered in this study, including the number of states and transitions in each Model and the total number of entity and relationship facts extracted in the factbase.

### Results of StateMachine Diagram Extraction

Table 4.2 summarizes the Rhapsody-generated StateMachine Model and the ten YAKINDU-generated StateMachine Models that we considered. For each of the eleven StateMachine Models used in this study, Graphical representations of diagrams from the Models and their XMI representations can be found in the online repository\(^1\). Additionally, the graphical representations of the diagrams are included in Appendix B.

Using the process described at the beginning of Section 4.1, we applied our two mel-generated extractors to the eleven StateMachine Models in our evaluation set produced result sets without error. We manually examined the generated facts and found that there were no sampled Model facts that did not exist in the generated factbase, plus the facts’ attributes all had appropriate values. Similarly, none of the extracted facts sampled were found to be extraneous to the Model.

\(^1\)https://github.com/Roshack/ModelsUsedWithmel
4.1.2 UML Class Diagrams

UML Class diagrams are perhaps the most well-known type of software Model and have found to be the diagram most commonly reported to be used in industry [31]. It is for this reason that mel’s applicability to UML Class diagrams is considered. Additionally, due to the popularity of UML Class diagrams, there exists an incredibly large number of distinct tools for generating them, each supporting different Modelling constructs and employing their own XM* representations. Thus, UML Class diagrams and tools provide good evidence of the potential need for specialized extractors for every distinct tool.

Information to be Extracted

The full set of information to be extracted from UML Class diagrams is reported in Table 4.3. It is important to note that each tool may support only a subset of the facts and attributes listed in Table 4.3. For example, UML Class diagrams created with the Eclipse Modelling Tools cannot denote a class as being final, meaning that no such attribute can be extracted. For each of the facts and attributes listed, if it is supported by the Modelling tool, then it should be extracted from the Model.

Models & Tools Considered

UML Class diagrams are the most easily available Model type and we consider diagrams from each of several tools:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Number of Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMLDesigner [55]</td>
<td>3</td>
</tr>
<tr>
<td>MagicDraw [33]</td>
<td>4</td>
</tr>
<tr>
<td>Eclipse Modelling Tools [63]</td>
<td>6</td>
</tr>
<tr>
<td>ArgoUML [60]</td>
<td>2</td>
</tr>
</tbody>
</table>

UMLDesigner, MagicDraw, and ArgoUML all store their diagrams in a representation approximating a OMG UML 2.X standard [19], thus their representations are all fairly similar to one another. Consider the UMLDesigner and Eclipse Modelling Tools diagrams shown in Figures 4.7 and 4.8, respectively. The UMLDesigner Model is for a fictional Travel Agency system, which is a sample UML Class diagram project provided with the UMLDesigner version 9.0 [55]. The Model generated with the Eclipse Modelling Tools is sourced from the UML model database [25] and is a recreation of a pedagogical example from Warmer
Table 4.3: Necessary facts for extraction from UML Class diagrams.

<table>
<thead>
<tr>
<th>Entities</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>name, abstract, final, and interface*</td>
</tr>
<tr>
<td>Interfaces</td>
<td>name</td>
</tr>
<tr>
<td>Enumeration types</td>
<td>name, visibility, constness, type,</td>
</tr>
<tr>
<td></td>
<td>multiplicities, and concreteness</td>
</tr>
<tr>
<td>Fields</td>
<td>name, visibility, return type, and</td>
</tr>
<tr>
<td></td>
<td>concreteness</td>
</tr>
<tr>
<td>Methods</td>
<td>name and type</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
</tr>
</tbody>
</table>

*Depending on the language interfaces may be their own entity, or a class may define an interface

and Kleppe’s book on the Object Constraint Language [68]. The two Models have different XMI representations: UMLDesigner serializes its Models according to the OMG UML standard version 2.1, and the Eclipse Modelling Tools serialize Models according to the EMF ECore specification and file format.

The most notable difference between the OMG XMI and the ECore representations of UML Class diagrams is in their methods of referencing other nodes. Documents adhering to the OMG XMI specification refer to other nodes through unique identifier Attributes. ECore documents, as discussed earlier in Section 2.2.2, refer to other nodes through strings that represent paths in the XMI structure.

Figures 4.9 and 4.10 demonstrate the two distinct methods of resolving references. The association relationship in OMG representation is determined by considering the type Attribute of the two ownedEnd nodes that are children of the packagedElement whose
Figure 4.7: UML Class Model of a travel agency (UMLDesigner).

**xmi:type** is **uml:Association**. In ECore representation, the association relationship is perhaps more intuitive to read, because the two-way relationship is defined as an embedded XML node in both of the two **eStructuralFeatures** nodes. The first node defines the relationship from class **ServiceLevel** to the class **Membership** (resolved by the value of the **eType** Attribute). The opposing relationship is both mentioned by path in the first **eStructuralFeatures** node, but also defined itself as an **eStructuralFeatures** child node of the **Membership** class node.

**mel Extractors for UML Class Diagrams**

The **mel** extractors for UMLDesigner class diagrams and Eclipse Modelling Tools class diagrams comprise mostly simple **mel** declarations, so we do not discuss them in detail. The full programs for each are located in the Appendix in Figures A.3 and A.4 respectively.

Instead we will consider just the extraction of **Association** facts expressed as two **mel** declarations, shown in Figures 4.11 and 4.12. The interesting detail about these two declarations is that the bindings of the **target** attribute of each association are not all that different despite the distinct methods of referencing entities. One might expect that the extractor for ECore Models would need to resolve the entire path to determine the **target** of an association, however it can be resolved with a single attribute binding. This is because the regular expression selectors that **mel** supplies allow the user to easily extract
the identifier, which is needed to find the target class (the $\text{targ}$ binding in Line 4 of Figure 4.12). Thus, we do not need to use the path operators to extract the $\text{target}$ and are able to resolve it with a single binding similar to the resolution of the $\text{target}$ for UMLDesigner Models. The similarity to the declaration for extraction associations from UMLDesigner Models can be seen in Figure 4.11, in which $\text{type}$ Attributes in the $\text{ownedEnd}$ nodes are used to resolve the bindings of $S$ and $E$ (Lines 6 and 8 of Figure 4.11).

**Results of UML Class Diagram Extraction**

Each $\text{mel}$-generated extractor was able to extract all the relevant facts from each UML Class diagram that we produced using each of the four tools: UMLDesigner, MagicDraw, Eclipse Modelling Tools, and ArgoUML. It is also worth noting the differences between the $\text{mel}$ programs for UMLDesigner, MagicDraw, and ArgoUML are relatively small leading to very rapid development for the second-, third-, and fourth-developed extractors because we were being able to reuse most of the facts already defined.

Using the process described at the beginning of Section 4.1, we applied our $\text{mel}$-generated extractors to fifteen subject models, summarized in Table 4.4. For each of
Figure 4.9: Snippet of the XMI representation of the UMLDesigner Class diagram shown in Figure 4.7, representing an association relationship. Identifiers have been abbreviated and non-referenced Attributes have been omitted.

Figure 4.10: Snippet of the ECore representation of the Eclipse Modelling Tools Class diagram shown in Figure 4.8. Identifiers have been abbreviated and non-referenced Attributes have been omitted.
Figure 4.11: mel declaration for extracting association facts from a UML Class diagram generated with UMLDesigner.

Figure 4.12: The mel declaration for extracting association facts from a UML Class diagram generated with the Eclipse Modelling Tools.
<table>
<thead>
<tr>
<th>Modelling Tool</th>
<th>Name</th>
<th>Classes</th>
<th>Associations</th>
<th>Entities</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>Advent</td>
<td>45</td>
<td>42</td>
<td>177</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>Zork</td>
<td>10</td>
<td>90</td>
<td>52</td>
<td>163</td>
</tr>
<tr>
<td></td>
<td>Coffee Machine</td>
<td>11</td>
<td>7</td>
<td>14</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>Java Language</td>
<td>9</td>
<td>25</td>
<td>10</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>MakeItHappen</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Royal and Loyal</td>
<td>15</td>
<td>54</td>
<td>38</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>Store</td>
<td>55</td>
<td>73</td>
<td>65</td>
<td>305</td>
</tr>
<tr>
<td></td>
<td>XUml Compiler</td>
<td>61</td>
<td>118</td>
<td>70</td>
<td>214</td>
</tr>
<tr>
<td>Eclipse Modelling Tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Alarmas</td>
<td>6</td>
<td>3</td>
<td>124</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>MediaTeka</td>
<td>4</td>
<td>4</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>Parking Lot</td>
<td>14</td>
<td>10</td>
<td>41</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>Library</td>
<td>5</td>
<td>6</td>
<td>32</td>
<td>43</td>
</tr>
<tr>
<td>MagicDraw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLAY Engine</td>
<td>27</td>
<td>26</td>
<td>32</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>Rennspur</td>
<td>17</td>
<td>17</td>
<td>94</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>Travel Agency</td>
<td>8</td>
<td>16</td>
<td>43</td>
<td>55</td>
</tr>
</tbody>
</table>

Table 4.4: Summary of the UML Class diagrams considered in this study, showing the number of classes and associations (general associations, aggregations, and compositions) in each Model and the total number of entity and relationship facts extracted in the factbase.
the 15 Class diagrams used in this study. Graphical representations of diagrams from the Models and their XMI representations can be found in the online repository\textsuperscript{2}. Additionally, the graphical representations of the diagrams are included in the Appendix B.

We manually examined 20 Model elements randomly selected from each subject Model and verified that each element was correctly extracted in the factbase with the proper details and that none of the facts randomly selected from each extracted factbase were determined to be extraneous or incorrect in their details.

4.1.3 Simulink Block Diagrams

Simulink was considered due to its popularity as a modelling and code-generation tool that sees wide use in industry particularly in safety-critical systems and industries requiring strict adherence to specified standards [29]. Additionally, the inclusion of Simulink Block diagrams in the study improves its analysis of scope of applicability, because Simulink Models have the most distinct textual representation in that XML tags typically have no Attributes for referencing related nodes, instead referencing related nodes by its location in the XML tree structure.

Information to be Extracted

The information to be extracted from Simulink Block diagrams is shown in Table 4.5. We extract blocks which have many different types for example gain, signal generator, or subsystem blocks which contain nested diagrams. The two relationships we are interested in extracting are the flow of signals between blocks and the containment hierarchy of blocks that occurs when subsystem blocks comprise nested blocks.

Models Considered

The only tool considered for these diagrams is the MATLAB Simulink tool [48], as Simulink modelling notation is proprietary. Ten Simulink Models were used in the experiment. A single system of one of the Models considered is shown in Figure 4.13.

Analogous to how state chart diagrams allow for nested state machines, Simulink Block diagrams allow for nested subsystem blocks. An example of a subsystem block is the block labeled “Aircraft Dynamics Model” in Figure 4.13; the details of this subsystem block are

\textsuperscript{2}https://github.com/Roshack/ModelsUsedWithmel
Table 4.5: Necessary facts for extraction from Simulink Block diagrams.

<table>
<thead>
<tr>
<th>Entities</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks</td>
<td>name and type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationships</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block/port ownership</td>
<td>port name</td>
</tr>
<tr>
<td>Signal flow</td>
<td>signal label</td>
</tr>
<tr>
<td>Subsystem block/block ownership</td>
<td></td>
</tr>
</tbody>
</table>

depicted in a separate Block diagram. A sample of the XML representation of the diagram shown in Figure 4.13 is shown in Figure 4.14. This XML snippet shows the signal flow from the q, rad/sec port in the Aircraft Dynamics Model subsystem to the Controller subsystem. To detect such signal flows mel must be able to refer to two children of a parent node that has no Attributes with which refer to it.

**mel Extractor for Simulink Block diagram**

The mel extractor for Simulink Block diagrams shown in Figure 4.15 demonstrates how the mel path operators can be used to denote relationships through a parent node with no Attributes. This is simply a result of the inherent meaning of the path when the right hand operand is a list, which denotes a collection of XML nodes that are all underneath the same ancestor/parent node. For example, the signal relationship on Line 6 of Figure 4.15 specifies that a signal relationship’s parameters are bound to the string contents of two P nodes which have Name Attributes of Src and Dest. There exist many such tags in the XML document, so it is important that the parent operator can be used to specify that these two tags are underneath the same Line node. If applied to the XML fragment shown in Figure 4.14 this would extract the relationship between block 3 and 33, but would not combine the values extracted from these P nodes with those extracted from other P nodes in the document, as they do not reside under the same (Attribute-free) Line tag.

**Results of Simulink Block Diagram Extraction**

A summary of the Simulink Models that we considered in this study can be seen in Table 4.6. For each of the 10 Block diagrams used in this study Graphical representations of diagrams from the Models and their XMI representations can be found in the online
<table>
<thead>
<tr>
<th>Name</th>
<th>Blocks</th>
<th>Signal Flow</th>
<th>Entities</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC Quantization Algorithm</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Air-Fuel Intake Dynamics</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Aircraft Longitudinal Flight Control</td>
<td>64</td>
<td>54</td>
<td>64</td>
<td>69</td>
</tr>
<tr>
<td>Anti-Windup PID Control Example</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>Bouncing Ball Model</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Component Based Modelling</td>
<td>14</td>
<td>8</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Fault-Tolerant Fuel Control System</td>
<td>21</td>
<td>20</td>
<td>21</td>
<td>29</td>
</tr>
<tr>
<td>Intake Airflow Estimation and Closed-Loop Correction</td>
<td>24</td>
<td>25</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>Modeling an Anti-Lock Braking System (ABS)</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>24</td>
</tr>
<tr>
<td>PID Controller Tuning</td>
<td>41</td>
<td>33</td>
<td>41</td>
<td>67</td>
</tr>
</tbody>
</table>

Table 4.6: Summary of the Simulink block diagrams considered in this study, showing the number of blocks and signals in each Model and the total number of entity and relationship facts extracted in the factbase.
Figure 4.13: A system of a Simulink Model for a longitudinal flight control algorithm. The blocks with titles beneath them are subsystems for which an entire sub-Model is defined.

repository\(^3\). Additionally, the graphical representations of the diagrams are included in the Appendix B. Using the process described at the beginning of Section 4.1, we applied our mel-generated extractor to each Simulink Model and produced result sets without error. No facts considered from the Models were missing in the generated factbases, and the generated factbases did not include extraneous facts.

4.1.4 Arcadia Logical Architecture Diagrams

Arcadia Logical Architectures diagrams from Thales’ Capella modelling tool were chosen for another distinct textual representation. Capella saves its Models in an XMI file that

\(^3\)https://github.com/Roshack/ModelsUsedWithmel
Figure 4.14: Snippet of the XML representation of the Simulink Block diagram shown in Figure 4.13. Non-referenced Attributes have been omitted.

Figure 4.15: mel extractor for Simulink Block diagrams.
Table 4.7: Necessary facts for extraction from Arcadia Logical Architecture diagrams.

<table>
<thead>
<tr>
<th>Entities</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>name</td>
</tr>
<tr>
<td>Components</td>
<td>name, is actor, is human</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationships</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data flow between functions</td>
<td>data label</td>
</tr>
<tr>
<td>Data flow between components</td>
<td>data label</td>
</tr>
<tr>
<td>Functions containing functions</td>
<td></td>
</tr>
<tr>
<td>Components containing functions</td>
<td></td>
</tr>
</tbody>
</table>

adheres to the OMG XMI standard 2.0 but is written using a custom metamodel defined for the tool.

**Information to be Extracted**

Arcadia Logical Architecture diagrams specify abstract components, functions, and data they communicate between them. As the Logical Architecture diagrams create an entire view of the logical behaviour of a system, they include actors in the system as components. Because of this, Component facts include attributes to specify whether or not they are an actor and if they are human, as shown in Table 4.7.

**Models Considered**

Four Arcadia Logical Architecture diagrams were considered for this part of the evaluation. One of the subject Models used is shown in Figure 4.16. In Arcadia Models, the entity types are mostly denoted by colour:

- The darker blue boxes are components
- The lighter blue boxes represent actors
  - If an actor has a stick figure symbol then it is also represents a human
- The green boxes represent functions
  - Function ownership is shown in other diagrams of the Architecture
• The lines between functions and functions, and components and components, represents the flow of data
  – The green text beside a line is the data being sent

A sample of the XML representation of Arcadia diagrams can be found in Figure 4.17

**mel Extractor for Arcadia Logical Architecture diagram**

The **mel**-generated extractor for Arcadia Logical Architecture diagrams is shown in Figure 4.18. This program contains relatively short and small declarations, because Arcadia Models have a simple XMI representation and noes have identifying tag Attributes — calling for the type of identifier references and relational reference that **mel** excels at.

**Results of Arcadia Logical Architecture Diagram Extraction**

Due to the smaller number of subject Models for evaluating **mel** on Arcadia Logical Architecture diagrams, the correctness-testing process described at the beginning of Section 4.1
Figure 4.17: XML Fragment of the Model shown in Figure 4.16. Type Attributes have had their values truncated to fit the page.

<table>
<thead>
<tr>
<th>Name</th>
<th>Components</th>
<th>Functions</th>
<th>Dataflows</th>
<th>Entities</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level Crossing Traffic Control</td>
<td>18</td>
<td>274</td>
<td>323</td>
<td>615</td>
<td>391</td>
</tr>
<tr>
<td>OCP Security Project</td>
<td>1</td>
<td>92</td>
<td>92</td>
<td>175</td>
<td>92</td>
</tr>
<tr>
<td>In-Flight Entertainment System</td>
<td>23</td>
<td>219</td>
<td>262</td>
<td>504</td>
<td>310</td>
</tr>
<tr>
<td>Zorro</td>
<td>1</td>
<td>41</td>
<td>29</td>
<td>71</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 4.8: Summary of the Arcadia diagrams considered in this study, showing the number of components, functions, and data flows in each Model and the total number of entity and relationship facts extracted in the factbase.
Figure 4.18: mel program for extracting facts from an Arcadia Logical Architecture diagram.
was modified to select forty, instead of twenty, Model elements or facts for assessing correctness. This means that we randomly selected 40 elements (or all, if fewer than 40 were present) from each Model and checked for their existence in the factbase. We also then selected 40 random facts from the factbase and verified they were present in the Model with the appropriate Attributes. This process found the elements sampled from each Model were present in their corresponding factbases, and the facts sampled from each factbase were present in their corresponding Model. For each of the 4 Arcadia diagrams used in this study Graphical representations of diagrams from the Models and their XMI representations can be found in the online repository. Additionally, the graphical representations of the diagrams are included in the Appendix B.

### 4.1.5 Feature Models

Feature Models were included in our assessment of mel because they are considerably different from the types of Models considered so far. The tool FeatureIDE version 3.6.2 [43, 40] is a well supported tool produced by FOSD researchers. FeatureIDE serializes Models according to its own XML format that does not adhere to any XMI standard, offering another unique XML representation for our assessments.

**Information to be Extracted**

The information to be extracted was the information about the features and the relationships between them. As such, the only entities extracted were features themselves, as shown in Table 4.9. Relationships between facts are either implicit in the Model (e.g., two features being alternative sub-features of the same parent feature are implicitly mutually exclusive) or explicit in the Model (e.g., a declared global constraint involving features that are not siblings).

**Models Considered**

Ten Feature Models were considered, one of which is shown in Figure 4.19. It is important to note the distinction between an Or group and an Alternative group, where the former is an inclusive or and the latter is exclusive or of the parent node’s child nodes. As

\[https://github.com/Roshack/ModelsUsedWithmel\]

77
Table 4.9: Necessary facts for extraction from Feature Models.

<table>
<thead>
<tr>
<th>Entities attributes</th>
<th>Feature is abstract, is mandatory, name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationships</td>
<td>Feature containment</td>
</tr>
<tr>
<td></td>
<td>Mutual exclusion of features</td>
</tr>
<tr>
<td></td>
<td>Mutual inclusion of features</td>
</tr>
<tr>
<td></td>
<td>Implication</td>
</tr>
<tr>
<td></td>
<td>Disjunction</td>
</tr>
<tr>
<td></td>
<td>Conjunction</td>
</tr>
</tbody>
</table>

Figure 4.19: Feature Model diagram for an elevator system (FeatureIDE).

Such, when extracting facts from a Feature Model, any features that are underneath the same Alternative must be considered to be mutually exclusive. Similarly, any features that are mandatory children of a group must be considered to be required together.

The XML representation that FeatureIDE uses mimics the tree-structure of the Feature Model exactly. This can be seen in Figure 4.20, which shows the top level Elevator feature and its five children. Because children nodes can be any of feature, or and, or alt, mel extractors can be verbose. However, the real cause for concern is the implementation of the cross-tree constraints.

Figure 4.21 demonstrates how cross-tree constraints are encoded in FeatureIDE’s textual representation. This XML snippet represents two logical expressions:

\[
\text{DirectedCall} \implies \text{ShortestPath} \\
\text{Behaviour} \implies (\text{CallButtons} \land \text{DirectedCall}) \lor (\text{Elevator} \land \text{FIFO})
\]
Figure 4.20: Snippet of the XML representation of the Feature Model diagram shown in Figure 4.19. Non-referenced Attributes have been omitted.

The way these expressions are represented poses a problem for mel. Although mel can handle expressions with a finite depth of nesting, it is unable to effectively handle arbitrarily nested expressions. In order to handle expressions of this type, which effectively define the grammar for logical expressions, a mel program would need to implement a logical expression parser - of which mel is not capable as it is not intended to be a Turing-complete language. Because mel seeks to abstract the accidental complexity of Model extraction, it is just meant to be a language that allows the user to declare the relevant relationships between an XM* document and a corresponding desired factbase; mel is not meant to be a language for arbitrary computation. However, as all of the Models considered had no degree of nesting deeper than three we were able to write a mel extractor which could extract all of the constraints in the sample Models.

**mel Extractor for Feature Model diagrams**

The extractor that was written for Feature Models is quite long, due to requiring many alternative rules for resolving children features. As such the program is displayed in the Appendix as Figure A.5. The set of declarations that defines the relationship between and nodes and their children is shown in Figure 4.22. This declaration demonstrates how the presence of many alternative XML tag types to denote the same abstract meaning can potentially lead to a combinatorial explosion of declarations.
Figure 4.21: Snippet of the XML representation of cross-tree constraints written in FeatureIDE.

Results of Feature Model Extracton

We applied mel to the ten Feature Models summarized in Table 4.10. While our mel-generated extractor was fully capable of extracting information from the Feature Tree, the extractor could only extract nested constraint expressions of finite depth. As our sampled Models had no constraints nested deeper than three levels, our mel-generated extractor that extracts constraints up to three levels deep was able to extract all of the facts from the sampled Models. Additionally, each of the facts randomly selected from each extracted factbase were present in the Model itself.

Figure 4.22: mel declarations for extracting children of and nodes in a FeatureModel.
<table>
<thead>
<tr>
<th>Name</th>
<th>Features</th>
<th>Group</th>
<th>Entities</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bike Configurator</td>
<td>54</td>
<td>12</td>
<td>55</td>
<td>154</td>
</tr>
<tr>
<td>Car</td>
<td>16</td>
<td>6</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>Desktop Searcher</td>
<td>16</td>
<td>8</td>
<td>22</td>
<td>32</td>
</tr>
<tr>
<td>EPL-AHEAD</td>
<td>12</td>
<td>4</td>
<td>23</td>
<td>43</td>
</tr>
<tr>
<td>Elevator</td>
<td>8</td>
<td>8</td>
<td>21</td>
<td>30</td>
</tr>
<tr>
<td>Hello World</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Landing Gear</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Poker</td>
<td>10</td>
<td>4</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>TankWar</td>
<td>31</td>
<td>11</td>
<td>37</td>
<td>62</td>
</tr>
<tr>
<td>Variable Characters</td>
<td>9</td>
<td>7</td>
<td>15</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 4.10: Summary of the Feature Model diagrams considered in this study, showing the number of features and groups (alternative, or, or general groups) in each Mode, and the total number of entity and relationship facts extracted in the factbase.
4.2 Ease of Using mel

The second criterion of our evaluation of mel involved assessing our hypothesis that it is easier to create a fact extractor for a Model using mel than using other technologies. Our assessment comprises two studies. The first study compares a mel-generated extractor to an existing specialized extractor for Arcadia Logical Architecture Models built using Python, which was created previously for a separate project; the comparison criterion is succinctness of the code (as a proxy for ease of programming).

The second study compares the time it takes for a user to create a fact extractor using mel versus the time it takes using the XQuery programming language; the comparison criteria are succinctness of the code and the development time (both as proxies for ease of programming). The execution times of the programs was not considered as a criterion because it is in the order of milliseconds on all examples, and is not relevant to how a language eases the programmers task. In each case, the outputs of both programs were compared against each other to verify that they extracted the same facts. Additionally, we examined the output of the extractors developed by the subjects against their target Models to verify the correctness of their extractors. We sampled twenty random facts from each target Model and verified that they existed in the extracted factbase. We also examined twenty random facts in each extracted factbase and compared them to the corresponding Model to verify their existence and the values of their attributes.

4.2.1 mel Comparison to Custom Extractor.

This study assessed the brevity of a mel program compared to that of an existing special-purpose fact extractor for Capella Models that was developed for a previous project. We use the brevity of a programming language as a proxy for measuring a language’s support for a programming task and the effort of programming. The latter extractor was developed using Python, which was chosen for the previous project because of its built-in library for parsing XML and its strong support for string processing. Both extractors were developed by the author of mel and this thesis. While times were not recorded, the development time of the mel extractor was considerably shorter; however, this comparison is entirely facile as the initial design of mel was built around generalizing the writing of this particular extractor - so by the time that mel was defined the extractor had already been conceived.

We could not use development time as a comparison criterion because the Python extractor was pre-existing and the time taken to develop it had not been recorded.
Both extractors take as input an Arcadia Logical Architecture diagram generated by Capella [20] and output the set of facts detailed in Table 4.7.

The sizes of the resulting programs are provided in Table 4.12. The size of the mel program is an order of magnitude smaller than the Python program, even when we control for mel’s automatic formatting of results by removing from the Python program the code responsible for formatting the output. This result is unsurprising: brevity is a natural side effect of most domain-specific languages. Still, the result is reported because a negative result was possible and would imply a failure in the design of the language.

4.2.2 Testing the Development Time of mel Programs.

This study evaluated mel with respect to the time it takes to create a fact extractor. Two user subjects were used, both of whom are experts in programming language theory. The user subjects shall henceforth be referred to as Subject A and Subject B. Neither subject was involved with the development of mel. We chose XQuery [61] to be the competing technology in this study because of its facilities for parsing XML and manipulating extracted values and its ease of use. Prior to the writing of this thesis, a conference paper on mel was published [22] which includes some evaluation by Subject A on two prior versions of mel. The three versions of mel (and corresponding mint) are version 1.0, version 1.1, and version 1.2 (which is the current version). We summarize the additions made by versions 1.1 and 1.2 in Table 4.11. The development of new versions of mel based on the results of prior studies is indicative of the design science research methodology [27] which was employed as part of this thesis.

Each evaluation was a set of tasks given to the subject. Each task was to create an extractor in both mel and XQuery for a specified Model. The subjects were given instructions to time their development of both extractors, as well as steps to follow to minimize favourable bias for either language. There were three steps to each experiment:

1. Examine the specified Model to determine the tags, relationships, and Attributes in the XML that were relevant to the facts they were asked to extract for the task.

2. Write their first extractor, timing their development time.

3. Write their second extractor, timing their development time.

In order to mitigate the likely bias that the second extractor for each Model type would be easier to write, the subjects were primarily asked to write their mel-generated extractor
first. Of the five tasks assigned to Subject A they were asked to write the XQuery extractor first in only one task. Similarly, of the four tasks given to Subject B they were asked to write the XQuery extractor first in only one task. This was so that if a favourable bias did exist it would be in XQuery’s favour.

We first discuss the experimental setup and then discuss the results of the experiments in Sections 4.2.3 and 4.2.4.

**Experiment Set One (mel 1.0)**

The first set of experiments was performed by Subject A for the inclusion in a conference paper [22]. The subject was asked to create mel and XQuery fact extractors for relevant facts from two Model representations:

Task1: Arcadia Logical Architecture diagram (Capella) - Figure 4.16

- **Entities**: functions, components, data entities
- **Relationships**: data flows between functions

Task2: UML Class diagram (UMLDesigner) Figure 4.7

- **Entities**: classes, attributes
- **Relationships**: classes’ attributes, associations, compositions
- **Attributes**: attributes’ type, visibility, constness, multiplicity

For each Task, the subject first analyzed the XML to determine which tags were relevant to the extraction. Additionally for each of these tasks, the subject wrote the mel program first and the XQuery program second. It is possible, for any Task, creating and debugging the first program helped the subject gain a better understanding of the XML, easing the development of the second program.

**Experiment Set Two (mel 1.1)**

In the second set of experiments, Subject A was provided with a third Model and a new version of mel (version 1.1) to test.

Task3: UML StateMachine diagram (Rhapsody) Figure 4.1:
In this task Subject A was asked to write their XQuery program first and mel program second.

Experiment Set Three (mel 1.2)

In the third set of experiments Subject A wrote extractors for two additional Models and a second subject, Subject B, wrote extractors for those same two Models, as well as for the Models originally provided to Subject A in Tasks 1 and 3.

The two new models were a UML Class diagram in ECore format, produced with the Eclipse Modelling Tools, and a Simulink Block diagram. Due to the length of time that had passed since Subject A had used mel or XQuery, and the fact that Subject B had not used either language, both subjects were given a set of facts to first extract from a UML Class diagram as a warm-up for both languages. Once they had completed that task, they were given an additional Model that included more details and were asked to time how long it took them to modify their extractors to extract the two additional facts. The reasons for the warm-up and subsequent followup task were two-fold: first to help the Subjects to familiarize themselves with each language, and secondly to provide some cursory insight into how extensible programs of each language were. The Simulink Block diagram test was completed the same as the tests from the first three experiments with Subject A.

Warm-up: UML Class diagram (ECore) Figure B.14

- **Entities**: classes, attributes
- **Relationships**: classes' attributes, associations, compositions
- **Attributes**: attributes' type, visibility, constness, multiplicity, classes' being abstract or final

Task4: UML Class diagram (ECore) Figure 4.8

- **Entities**: methods
• **Relationships:** classes’ methods

• **Attributes:** isDerived attribute for methods and isDerived attribute for fields

Task 5: Simulink Block diagram Figure 4.13

• **Entities:** blocks, ports

• **Relationships:** signal flow between blocks

• **Attributes:** names of signals

Both subjects performed their warm-up task followed by Task 4 and then Task 5, at which point Subject A was finished. Subject B then completed a task with the same setup as Task 1 (Arcadia) above, and finished by completing Task 3 (StateMachine) from above. In each of these tasks, both subjects properly separated XML analysis time from extractor development. All tasks were performed with the creation of the `mel` program first, except for when Subject B completed Task 3 in which they wrote their XQuery extractor first as Subject A was originally asked to write their XQuery extractor first for the same Model in the second experiment.

### 4.2.3 Succinctness of `mel`

In general, `mel` saw favourable performance over XQuery with respect to succinctness. The results shown in Table 4.12 demonstrate this.
Table 4.12: Program Sizes of Extractors, where numbers with a single underline are sizes of \texttt{mel} 1.1 programs and numbers with a double underline are sizes of \texttt{mel} 1.0 programs. All other numbers are sizes of \texttt{mel} 1.2 programs.

<table>
<thead>
<tr>
<th></th>
<th>Arcadia</th>
<th>UMLDesigner</th>
<th>ECore</th>
<th>StateChart</th>
<th>Simulink</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject A Lines</td>
<td>\texttt{mel}</td>
<td>6</td>
<td>38</td>
<td>36</td>
<td>31</td>
<td>52</td>
</tr>
<tr>
<td>XQuery</td>
<td>33</td>
<td>47</td>
<td>36</td>
<td>39</td>
<td>38</td>
<td>193</td>
</tr>
<tr>
<td>Words</td>
<td>\texttt{mel}</td>
<td>58</td>
<td>143</td>
<td>208</td>
<td>204</td>
<td>223</td>
</tr>
<tr>
<td>XQuery</td>
<td>115</td>
<td>207</td>
<td>277</td>
<td>178</td>
<td>312</td>
<td>1089</td>
</tr>
<tr>
<td>Chars</td>
<td>\texttt{mel}</td>
<td>434</td>
<td>1482</td>
<td>1709</td>
<td>1630</td>
<td>1618</td>
</tr>
<tr>
<td>XQuery</td>
<td>122</td>
<td>2016</td>
<td>2515</td>
<td>1763</td>
<td>2623</td>
<td>10039</td>
</tr>
<tr>
<td>Subject B Lines</td>
<td>\texttt{mel}</td>
<td>9</td>
<td>N/A</td>
<td>23</td>
<td>26</td>
<td>11</td>
</tr>
<tr>
<td>XQuery</td>
<td>38</td>
<td>N/A</td>
<td>83</td>
<td>77</td>
<td>45</td>
<td>243</td>
</tr>
<tr>
<td>Words</td>
<td>\texttt{mel}</td>
<td>22</td>
<td>N/A</td>
<td>70</td>
<td>60</td>
<td>35</td>
</tr>
<tr>
<td>XQuery</td>
<td>97</td>
<td>N/A</td>
<td>282</td>
<td>208</td>
<td>119</td>
<td>706</td>
</tr>
<tr>
<td>Chars</td>
<td>\texttt{mel}</td>
<td>493</td>
<td>N/A</td>
<td>1358</td>
<td>1348</td>
<td>616</td>
</tr>
<tr>
<td>XQuery</td>
<td>1175</td>
<td>N/A</td>
<td>3061</td>
<td>2547</td>
<td>1056</td>
<td>7839</td>
</tr>
</tbody>
</table>

For each program written by Subject B, the \texttt{mel}-generated extractor was considerably shorter than the XQuery program in terms of numbers of lines, words, and characters. The results for Subject A are slightly more mixed in terms of numbers of lines and words; but in total character count (including whitespace characters), Subject A’s \texttt{mel} programs are each more succinct than their XQuery counterparts.

Inspection of Subject A’s programs suggests that the main reason their \texttt{mel} programs had similar or even larger number of lines and words is due to stylistic spacing choices taken by Subject A. For their own reasons of readability, Subject A chose to separate most tokens by spaces, increasing total word and character count. Additionally, Subject A chose to place most clauses on their own line and also placed closing square brackets on their own line when they closed a collection of tags that served as the right-hand operand to a path operator. These stylistic choices are important considerations in evaluating the succinctness of \texttt{mel} as the Subject was not instructed to do this, suggesting that they did it to improve readability for themselves. Because readability directly affects the usability of a programming language, the increased length due to Subject A’s style choices should
not be ignored as inconsequential to program length, as without them the programs may have been harder to write for the Subject.

Even with the stylistic choices made by Subject A, each \texttt{mel} program still has fewer total lines than their XQuery counterparts other than Subject A’s ECore programs, which have equal number of lines, and Subject A’s Simulink programs, in which the \texttt{mel} program is larger than the XQuery program. Totalling the line counts for each Subject shows that Subject A’s \texttt{mel} programs were an average of 11\% lines fewer than their XQuery counterparts, while Subject B’s \texttt{mel} programs were an average of 71\% lines fewer than their XQuery counterparts.

The word counts of the Subjects’ programs show similar results, with each \texttt{mel} program having fewer total words than its corresponding XQuery program, except for Subject A’s StateChart programs. Totalling the word counts for each Subject shows that Subject A’s \texttt{mel} programs were an average of 23\% words fewer than their XQuery counterparts, while Subject B’s \texttt{mel} programs were an average of 73\% words fewer than their XQuery counterparts.

Lastly, the character counts weigh heavily in favour of \texttt{mel}, with not a single XQuery program taking fewer characters than its \texttt{mel} counterpart. Totalling the character counts for each Subject shows that Subject A’s \texttt{mel} programs were an average of 31\% characters fewer than their XQuery counterparts, while Subject B’s \texttt{mel} programs were an average of 51\% characters fewer than their XQuery counterparts.

There are a few potential reasons for the differences in program sizes. The most likely reason is that there are several cases where \texttt{mel} hides the accidental complexity of searching and retrieving data from the XM* document that matches relational declarations, where XQuery does not. One example of this, seen repeatedly in both Subjects’ programs, is in the resolutions of hierarchical structures of nodes. Processing relationships between specified nodes in a hierarchical structure typically required at least one for loop, if not nested for loops in the corresponding XQuery program. This repetitive search and retrieval of desired nodes from the XM* document is the type of accidental complexity that \texttt{mel} seeks to hide from the user. In a \texttt{mel} program, the user need not worry about \textit{how} to resolve the relationships they are interested in, but simply state what those relationships are. In fact, resolving any relationship between two nodes in XQuery demonstrates this accidental complexity. In order to search for pairs of nodes that share common Attribute values the user must again write explicit for loops that iterate over the possible nodes and produce results for those that match the desired relationship. Once again, \texttt{mel} resolves these relationships implicitly for the user, who needs only state the relationship itself.
4.2.4 Development Time of \texttt{mel} Extractors

Table 4.13: Development Time of Extractors. Time in bold denotes the extractor which was written first by the given subject for the Model type.

<table>
<thead>
<tr>
<th>Model</th>
<th>Subject A</th>
<th>Subject B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\texttt{mel} Time</td>
<td>XQuery Time</td>
</tr>
<tr>
<td>Arcadia</td>
<td>46m</td>
<td>75m</td>
</tr>
<tr>
<td>UMLDesigner Class Diagram†</td>
<td>53m</td>
<td>43m</td>
</tr>
<tr>
<td>ECore Class Diagram‡</td>
<td>6m 15s</td>
<td>8m 3s</td>
</tr>
<tr>
<td>StateChart</td>
<td>30m</td>
<td>54m</td>
</tr>
<tr>
<td>Simulink</td>
<td>29m 33s</td>
<td>30m 54s</td>
</tr>
<tr>
<td>Total</td>
<td>164m 48s</td>
<td>210m 57s</td>
</tr>
</tbody>
</table>

† For this task Subject A forgot to analyze the XML before writing extractors, as such the \texttt{mel} time is conflated with the XML deduction time.
‡ The task involving ECore Class diagrams was not the development of the entire extractor, but rather the addition of two fact declarations to the extractors that the Subjects had previously developed.

Table 4.13 shows the development time taken by each Subject to create each extractor. Subject A spent less time writing their extractors in \texttt{mel} than in XQuery for four of the five tasks given to them. Of the four tasks that Subject B completed, they spent less time on their \texttt{mel} programs than their XQuery programs in two tasks, and vice versa on the other two. However, there are some details of note about the tasks in which Subjects took longer to write their \texttt{mel} programs than their XQuery programs.

Subject A forgot to spend time analyzing the XML structure before beginning development for their \texttt{mel}-generated extractor for UML Class diagrams produced with UMLDesigner. Given that Subject A recorded their XML deduction time for Arcadia Logical Architecture diagrams and StateChart diagrams as 26 minutes and 33 minutes respectively, it is reasonable to assume that the fact that it took 10 minutes longer for them to develop their \texttt{mel} extractor for UMLDesigner Models than their XQuery program was at least partly due to time spent interpreting the XML.

Subject B provided their thoughts on why their \texttt{mel} program for Arcadia Logical Architecture diagrams took much longer to develop than their XQuery program: (1) they forgot the semantics of compound clauses and had to go review the documentation they
were provided, and (2) they misspelled an XML reference in their program. Two possible causes of Subject B’s first suggested slowdown could be due to the learning curve with a new language or it could indicate that the behaviour of compound clauses in mel are unintuitive or difficult to work with. In the case of Subject B’s second suggested slowdown, mint actually includes a feature to help debug misspelled XML references, however the Subject was unaware of this feature and missed it entirely. The feature in question produces a colourful warning when an XML reference clause produces zero results, suggesting that the programmer may have misspelled something. The warnings mint produce are printed to the standard error stream, but are printed before the program output is printed to standard output. Because the warnings were printed before the output of the program Subject B never noticed them. Had Subject B been aware of mint’s warning features, they could have deliberately checked for any warnings, which they believe would have saved them time.

Overall, both subjects spent less total time developing their mel programs than their XQuery programs. Subject A spent just under 165 minutes developing all of their mel programs and just under 211 minutes developing all of their XQuery programs. Thus, on average Subject A spent roughly 21% less time on developing their mel programs. Similarly, Subject B spent 96 minutes developing their mel programs and 122 minutes developing their XQuery programs for an average of 21% less time spent developing their mel programs. Also, it is not insignificant that users spent less time on average developing mel programs than time on developing XQuery programs, despite the fact that the majority of the tasks were biased in favour of XQuery because the Subjects wrote their mel programs first. However, we must also consider the possibility that the learning effort exerted in the first task left each Subject with less energy to expend on the second task, potentially leading to slower development of the second extractor.

There are a few potential reasons for the differences in development times. Once again, we believe that efficiencies in development time offered by mel are primarily the result of the way mel hides accidental complexities from the user. Additionally, both of our Subjects described particular issues in debugging mistakes in XQuery programs in contrast to mel. Subject A specifically mentioned that the optional-clauses feature of mel were particularly useful in debugging, as they could toggle clauses on/off easily to find the mistake that was causing their declaration to produce incorrect results. In contrast, while debugging XQuery programs Subject A had a more difficult time factoring out their for loops into let bindings to isolate errors. Subject A also mentioned that mel excels over XQuery at resolving relationships between cross-tree nodes: that is, resolving relationships between nodes with matching Attribute values regardless of their location in the XM* document — due to how mel hides the accidental complexity of walking the tree from the user. Subject
B also expressed the same belief, however they characterized this as mel being easier at expressing joins. Again, a join in XQuery would take the form of a for loop wherein the programmer resolves the join’s relationships themselves, an explicit resolution of the accidental complexity of extracting relational data from an XM* document.
Table 4.11: A summary of the features added in versions 1.1 and 1.2 (current) of mel (and mint).

<table>
<thead>
<tr>
<th>Version Number</th>
<th>Features Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1.1</td>
<td>Added warning messages to mint when:</td>
</tr>
<tr>
<td></td>
<td>1. a declaration reference refers to an undeclared name</td>
</tr>
<tr>
<td></td>
<td>2. no instances of an Attribute name referenced in an XML reference are found in the corresponding XML document</td>
</tr>
<tr>
<td></td>
<td>3. a declaration produced zero facts.</td>
</tr>
<tr>
<td>Version 1.2</td>
<td>• Introduced the path operators, deprecating the mel.parent and mel.ancestor functions†</td>
</tr>
<tr>
<td></td>
<td>• Introduced constraint clauses</td>
</tr>
<tr>
<td></td>
<td>• Introduced regular-expressions for use with Attribute modifiers, which previously matched only strings between a user-defined prefix and suffix</td>
</tr>
<tr>
<td></td>
<td>• Introduced the like operator (∼=)</td>
</tr>
<tr>
<td></td>
<td>• Introduced single-quote string literals and the ability to bind attributes to string literals</td>
</tr>
<tr>
<td></td>
<td>• Introduced the ternary operator</td>
</tr>
</tbody>
</table>

† — These functions are still present in the mel grammar shown in Figure 3.25, but not discussed in Chapter 3. This is because these functions are deprecated in favour of the path operators. An application of mel.parent or mel.ancestor behave the same as a single use of the parent or ancestor operators respectively.
Chapter 5

Conclusions

This chapter discusses the contributions made in this thesis, as well as the limitations of the work, and finally the possibilities of future work.

5.1 Contributions

This thesis provides as its primary contributions the development of the Model Extraction Language \texttt{mel} and its corresponding interpreter \texttt{mint}. As a language \texttt{mel} has been used to extract facts from UML Class and StateMachine diagrams, Arcadia Logical Architecture diagrams, Simulink Block diagrams, and Feature Models. Additionally, the Models analyzed came from a wide variety of tools with varying XM* schemata which demonstrates a wider applicability of \texttt{mel}. Evaluations show promising results with respect to the applicability of \texttt{mel} to Models represented in XM*; and to \texttt{mel}'s ability to ease the development of fact extractors. Experimental results found that \texttt{mel} was at least as effective as a language for extracting information from XML documents as the W3C recommended tool XQuery. Additionally, on average, the written \texttt{mel} programs were shorter and took less development time than their XQuery counterparts.

5.2 Threats and Limitations

Although evaluations of \texttt{mel} produced generally positive results, there are some limitations to the language worth noting and threats to the validity of the experimental results to discuss.
The one notable limitation of mel as a language was found when writing an extractor for Feature Models. While the facts represented within the feature tree did not pose a problem to mel, the arbitrary cross-tree constraints did. Due to the fact that the cross-tree constraints took the form of arbitrarily nested logical expressions, mel was unable to resolve any constraint to a fact. The mel extractor written was able to resolve constraints so long as they were limited to a depth of one. Although this is a limitation of mel, it is outside of the scope of the language. The arbitrarily nested constraints are written in a way that would require the extractor to simulate a recursive parser for logical expressions. Since mel, much like Datalog which it is influenced by, is not a Turing-complete language it cannot achieve such behaviour.

The most notable threat to the evaluation is that the evaluation of mel’s ability to ease extractor development time was performed by only two subjects on a combined total of Five Models. This small sample size could mean that mel’s performance can be explained by statistical noise rather than by the features of the language. Additionally, both subjects are personal acquaintances of the author of this thesis which may have subconsciously biased their evaluation. It is worth noting, however, that both subjects are academics and would not intentionally bias the results of the experiment.

5.3 Future Work

Future work extending this thesis could include, but is not limited to, (1) extending the features of mel to support more Model representations: more XML representations, as in the constraints of Feature Models, or generalizing the back-end of mel so that XML reference and path reference clauses can be dispatched to other representations as well. (2) Extending the evaluations of mel through applying it to additional Model types, Model examples, industrial case studies, and the performing of a full user study.
References


[40] Christian Kästner, Thomas Thümm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented


Appendix A

Extra mel Programs

A.1 mel-Generated Extractors for UML Class diagrams

```plaintext
Class(X) |- packagedElement{"xmi:type"="uml:Class", name:label, "xmi:id":X,
   mel.nexists(ID)};

Enum(X) |- packagedElement{"xmi:type"="uml:Enumeration", name:label, "xmi:id":X,
   mel.nexists(ID)};

EnumValues(E,V) |- packagedElement{"xmi:type"="uml:Enumeration", name:enum, "xmi:id":E}
   ->ownedLiteral{"xmi:id":V, name:enumValue, value:value};

InheritsFrom(Parent,Child) |- packagedElement{"xmi:id":Child}
   ->generalization{general:Parent};

Interface(X) |- packagedElement{"xmi:type"="uml:Interface", name:name, "xmi:id":X,
   mel.nexists(ID)};

Field(X) |- packagedElement{"xmi:type"="uml:Class"}
   ->ownedAttribute{"xmi:type"="uml:Property", "xmi:id":X, name:label,
   visibility:visibility, isReadOnly:const, isStatic:static,
   type:typeID, mel.exists(type), mel.nexists(association),
   mel.nexists(ID)};
   @packagedElement{"xmi:id":typeID, name:type};

Field(X) |- packagedElement{"xmi:type"="uml:Class"}
   ->ownedAttribute{"xmi:type"="uml:Property", "xmi:id":X, name:label,
   visibility:visibility, isReadOnly:const,
```
Field(X) |- packagedElement{"xmi:type"="uml:Class"}
->ownedAttribute{"xmi:type"="uml:Property", "xmi:id":X, name:label,
visibility:visibility, isReadOnly:const,
mel.nexists(type), mel.nexists(association),
mel.nexists(ID)}
->type{}=>referenceExtension{referentPath[".*::(\[^:]*$\)\]:type};

FieldOf(C,A) |- packagedElement{"xmi:type"="uml:Class", "xmi:id":C, mel.nexists(ID)}
->ownedAttribute{"xmi:id":A, mel.nexists(association)};

Method(X) |- ownedOperation{"xmi:id":X, name:name, visibility:visibility,
mel.nexists(ID)}
->@ownedParameter{"xmi:type"="uml:Parameter", direction="return",
name:returnName, type:TID},
@packagedElement{"xmi:type"="uml:Class", name:returnType, "xmi:id":TID};

MethodOf(Class, Method) |- packagedElement{"xmi:id":Class, mel.nexists(ID)}
->ownedOperation{"xmi:id":Method},
Class(Class), Method(Method);

Parameter(X) |- ownedParameter{"xmi:type"="uml:Parameter", "xmi:id":X, name:name,
type:TID, direction!="return", mel.nexists(ID)},
@packagedElement{"xmi:type"="uml:Class", name:type, "xmi:id":TID};

ParameterOf(Fn,P) |- ownedOperation{"xmi:id":Fn}->ownedParameter{"xmi:id":P},
Method(Fn), Parameter(P);

Composition(Owner,Ownee) |
packagedElement{"xmi:id":Owner}
->ownedAttribute{"xmi:id":ownerRelID, "xmi:type"="uml:Property", name:ownerLabel,
association:AID, aggregation=composite}
=>[@lowerValue{value:ownerLowerMult}, @upperValue{value:ownerUpperMult}]
packagedElement{"xmi:id":Ownee}
->ownedAttribute{"xmi:id":owneeRelID, "xmi:type"="uml:Property",
name:owneeLabel, association:AID}
=>[@lowerValue{value:owneeLowerMult}, @upperValue{value:owneeUpperMult}]
packagedElement{"xmi:type"="uml:Association", "xmi:id":AID, name:compLabel},
owneeRelID!=ownerRelID;
Figure A.1: mel program for extracting facts from UML Class diagrams generated with the MagicDraw tool.
Figure A.2: mel program for extracting facts from UML Class diagrams generated with the ArgoUML tool.
Figure A.3: mel program for extracting facts from UML Class diagrams generated with UMLDesigner.
Figure A.4: mel program for extracting facts from UML Class diagrams generated with the Eclipse Modelling Tools.
A.2 \textit{mel}-Generated Extractor for Feature Models

```
Feature(X) |- struct{}=>feature{abstract:isAbstract, mandatory:isMandatory, name:X};
Feature(X) |- and{abstract:isAbstract, mandatory:isMandatory, name:X};
Feature(X) |- or{abstract:isAbstract, mandatory:isMandatory, name:X};
Feature(X) |- alt{abstract:isAbstract, mandatory:isMandatory, name:X};
FeatureOf(A,B) |- and{name:A}->and{name:B};
FeatureOf(A,B) |- and{name:A}->alt{name:B};
FeatureOf(A,B) |- and{name:A}->or{name:B};
FeatureOf(A,B) |- and{name:A}->feature{name:B};
FeatureOf(A,B) |- alt{name:A}->and{name:B};
FeatureOf(A,B) |- alt{name:A}->alt{name:B};
FeatureOf(A,B) |- alt{name:A}->or{name:B};
FeatureOf(A,B) |- alt{name:A}->feature{name:B};
FeatureOf(A,B) |- or{name:A}->and{name:B};
FeatureOf(A,B) |- or{name:A}->alt{name:B};
FeatureOf(A,B) |- or{name:A}->or{name:B};
FeatureOf(A,B) |- or{name:A}->feature{name:B};
^MutuallyExclusive(A,B) |- Feature(A), Feature(B), A!B,
  alt{name:parent},
  FeatureOf(parent,A), FeatureOf(parent,B);
^RequiredTogether(A,B) |- Feature(A){isMandatory:M}, Feature(B){isMandatory:M},
  M='true', A!B, and{name:parent},
  FeatureOf(parent,A), FeatureOf(parent,B);
Implies(A,B) |- imp{}->[var{mel.contents:A}], [var{mel.contents:B}], A!B;
Disjoint(A,B) |- disj{}->[var{mel.contents:A}], [var{mel.contents:B}], A!B;
Conjoint(A,B) |- conj{}->[var{mel.contents:A}], [var{mel.contents:B}], A!B;
```

Figure A.5: \textit{mel} program for extracting facts from Feature Model diagram generated with FeatureIDE.
A.3 mel and XQuery programs written by Subject A

```
function(id) |- ownedFunctions { id: id, name: name }
component(id) |- ownedSystemComponents { id: id, name: name }
dataFlow(from, to) |- 
    ownedFunctionalExchanges { source["#%": source, target["#%": target,
        name: name },
    parent(inputs { id: target, name: inputName },
        ownedFunctions { id: from, name: inputFunction }),
    parent(outputs { id: source, name: outputName },
        ownedFunctions { id: to, name: outputFunction });
```

Figure A.6: mel program written by Subject A for completion of Task 1 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
Figure A.7: XQuery program written by Subject A for completion of Task 1 as described in Section 4.2.2.
class(id) |- packagedElement { "xmi:id": id, name: name, "xmi:type" = "uml:Class" };

classContains(class, attr) |
   class(class),
   mel.parent(packagedElement { "xmi:id": class }, ownedAttribute { "xmi:id": attr });

attribute(id) |
   mel.parent(ownedAttribute { "xmi:id": id, name: label },
              type { href ["pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#%"]: type }),
   mel.parent(ownedAttribute { "xmi:id": id }, lowerValue { value: lowerBound }),
   mel.parent(ownedAttribute { "xmi:id": id }, upperValue { value: upperBound });

association(from, to) |
   mel.parent(
      packagedElement { "xmi:id": associationId },
      ownedEnd { type: from, name: fromName, aggregation != "composite" }),
   mel.parent(
      packagedElement { "xmi:type" = "uml:Association",
                      "xmi:id": associationId,
                      name: associationName,
                      navigableOwnedEnd: navigableOwnedEnd
      },
      ownedEnd { type: to, name: toName, "xmi:id": navigableOwnedEnd });

composition(from, to) |
   mel.parent(
      packagedElement { "xmi:id": associationId },
      ownedEnd { type: from, name: fromName, aggregation = "composite" }),
   mel.parent(
      packagedElement { "xmi:type" = "uml:Association",
                      "xmi:id": associationId,
                      name: associationName,
                      navigableOwnedEnd: navigableOwnedEnd
      },
      ownedEnd { type: to, name: toName, "xmi:id": navigableOwnedEnd });

Figure A.8: mel program written by Subject A for completion of Task 2 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
xquery version "3.1";

declare namespace xmi = "http://schema.omg.org/spec/XMI/2.1";

for $doc in doc("db/agency.uml")
let $classes :=
  for $class in $doc//packagedElement[@xmi:type = "uml:Class"]
  return <class id="{ $class/@xmi:id }" name="{ $class/@name }" />
let $attributes :=
  for $attr in $doc//ownedAttribute
  let $type := replace($attr/type/@href, "^.*#", "")
  let $lowerBound := $attr/lowerValue/@value
  let $upperBound := $attr/upperValue/@value
  return <attribute id="{ $attr/@xmi:id }" name="{ $attr/@name }" type="{ $type }"
    lowerBound="{ $lowerBound }" upperBound="{ $upperBound }" />
let $classAttrs :=
  for $class in $doc//packagedElement[@xmi:type = "uml:Class"]
  for $attr in $class/ownedAttribute
  return <classAttr classId="{ $class/@xmi:id }" attrId="{ $attr/@xmi:id }" />
let $associations :=
  for $assoc in $doc//packagedElement[@xmi:type = "uml:Association"]
  let $fromEnd := $assoc/ownedEnd[@xmi:id = tokenize($assoc/@memberEnd, "\\s")][1]
  let $toEnd := $assoc/ownedEnd[@xmi:id =
    tokenize($assoc/@navigableOwnedEnd, "\\s")][last()]
  where not(exists($fromEnd/@aggregation)) or $fromEnd/@aggregation != "composite"
  return <association
    id="{ $assoc/@xmi:id }"
    name="{ $assoc/@name }"
    from="{ $fromEnd/@type }"
    to="{ $toEnd/@type }" />
let $compositions :=
  for $assoc in $doc//packagedElement[@xmi:type = "uml:Association"]
  let $fromEnd := $assoc/ownedEnd[@xmi:id = tokenize($assoc/@memberEnd, "\\s")][1]
  let $toEnd := $assoc/ownedEnd[@xmi:id =
    tokenize($assoc/@navigableOwnedEnd, "\\s")][last()]
  where $fromEnd/@aggregation = "composite"
  return <composition
    id="{ $assoc/@xmi:id }"
    name="{ $assoc/@name }"
    from="{ $fromEnd/@type }"
    to="{ $toEnd/@type }" />
return <data> { $classes } { $attributes } { $classAttrs } { $associations } { $compositions } </data>

Figure A.9: XQuery program written by Subject A for completion of Task 2 as described in Section 4.2.2. Line breaks have been removed in some places and added in others to accommodate formatting for this thesis.
Figure A.10: meL program written by Subject A for completion of Task 3 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
xquery version "3.1";

declare namespace xmi = "http://schema.omg.org/spec/XMI/2.1";

for $doc in doc("homealarm.xmi")
let $stateMachines :=
  for $stm in $doc//ownedBehavior[@xmi:type = "uml:StateMachine"]
  return <statemachine id="{$stm/@xmi:id}" name="{$stm/@name }"/>

let $regions :=
  for $region in $doc//region[@xmi:type = "uml:Region"]
  let $stateMachine := if( exists($region/@stateMachine) )
    then $region/@stateMachine else
    $region/ancestor::region[exists(.//@stateMachine)]/@stateMachine
  return <region id="{$region/@xmi:id }" name="{$ region/@name }"
    stateMachine="{$ stateMachine }"/>

let $states :=
  for $state in $doc//subvertex[@xmi:type = "uml:State"]
  return <state id="{$ state/@xmi:id }" name="{$ state/@name }"
    region="{$ state/parent::region/@xmi:id }"/>

let $stateContainss :=
  for $state in $doc//subvertex[@xmi:type = "uml:State"]
  for $child in $state/region/subvertex[@xmi:type = "uml:State"]
  return <stateContains outer="{$ state/@xmi:id }" inner="{$ child/@xmi:id }"/>

let $transitions :=
  for $trans in $doc//transition[@xmi:type = "uml:Transition"]
  let $eventId := $trans/trigger/@event
  let $signalEvent := $doc//packagedElement[@xmi:type = "uml:SignalEvent"
    and @xmi:id = $eventId]
  let $signal := $doc//packagedElement[@xmi:type = "uml:Signal"
    and @xmi:id = $signalEvent/@signal]
  return <transition id="{$ trans/@xmi:id }" source="{$ trans/@target }"
    target="{$ trans/@target }" trigger="{$ signal/@name }"
    guard="{$ trans/ownedRule/specification/@value }"
    effect="{$ trans/effect/body/text() }"/>

return <data>
{ $stateMachines } { $regions }
{ $states } { $stateContainss } { $transitions }
</data>

Figure A.11: XQuery program written by Subject A for completion of Task 3 as described in Section 4.2.2. Line breaks have been removed in some places and added in others to accommodate formatting for this thesis.
Figure A.12: mel program written by Subject A for completion of Task 4 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
xquery version "3.1";
let $doc := doc("royal.xml")
let $class :=
for $class in $doc // eClassifiers [ @xsi:type = "ecore:EClass" ]
return <class name="{$class/@name}" abstract="{if($class/@abstract = "true")
  then "true" else "false"}/>
let $memberOf :=
for $class in $doc // eClassifiers [ @xsi:type = "ecore:EClass" ]
for $member in $class / eStructuralFeatures [ @xsi:type = "ecore:EAttribute" ]
return <memberOf className="{$class/@name}" name="{$member/@name}"
  type="{$member/@eType}" />
let $superClassOf :=
for $class in $doc // eClassifiers [ @xsi:type = "ecore:EClass" and @eSuperTypes ]
let $superClassName := analyze-string($class/@eSuperTypes, "#//(\w+)" ) / fn:match / fn:group [ @nr = "1" ] / text()
return <superClassOf className="{$class/@name}"
  superClassName="{$superClassName}" />
let $associationOrComposition :=
for $class in $doc // eClassifiers [ @xsi:type = "ecore:EClass" ]
for $assoc in $class / eStructuralFeatures [ @xsi:type = "ecore:EReference" ]
let $targetName := analyze-string($assoc/@eType, "#//(\w+)" ) / fn:match / fn:group [ @nr = "1"] / text()
let $lowerMult := if(exists($assoc/@lowerBound)) then $assoc/@lowerBound else "0"
let $upperMult := if($assoc/@upperBound = "-1") then "*" else $assoc/@upperBound
let $derived := if($assoc/@derived = "true") then "true" else "false"
return if($assoc/@containment = "true")
then <composition from="{$class/@name}" to="{$targetName}" label="{$assoc/@name}" lowerMult="{$lowerMult}" upperMult="{$upperMult}" derived="{$derived}" />
else <association from="{$class/@name}" to="{$targetName}" label="{$assoc/@name}" lowerMult="{$lowerMult}" upperMult="{$upperMult}" derived="{$derived}" />
let $methodOf :=
for $class in $doc // eClassifiers [ @xsi:type = "ecore:EClass" ]
for $method in $class / eOperations
let $returnType := analyze-string($method/@eType, "#//(\w+)" ) / fn:match / fn:group [ @nr = "1"] / text()
return <methodOf class="{$class/@name}" name="{$method/@name}"
  returnType="{$returnType}" />
let $paramOf :=
for $method in $doc // eClassifiers [ @xsi:type = "ecore:EClass" ] / eOperations
for $param in $method / eParameters
let $type := analyze-string($param/@eType, "#//(\w+)" ) / fn:match / fn:group [ @nr = "1"] / text()
return <paramOf method="{$method/@name}" name="{$param/@name}" type="{$type}" />
return <results> {$class} {$memberOf} {$superClassOf} {$associationOrComposition} {$methodOf} </results>

Figure A.13: XQuery program written by Subject A for completion of Task 4 as described in Section 4.2.2. Some line breaks have been added to allow for formatting of this thesis.
Figure A.14: *mel* program written by Subject A for completion of Task 5 as described in Section 4.2.2. Line breaks have been removed in some places and added in others to accommodate formatting for this thesis.
xquery version "3.1";

let $doc := doc("simulink.xml")
let $signalGenerator :=
    for $block in $doc // Block [ @BlockType = "SignalGenerator" ]
    return <signalGenerator id="{$block/@SID}" name="{$block/@Name}" />
let $sum :=
    for $block in $doc // Block [ @BlockType = "Sum" ]
    return <sum id="{$block/@SID}" name="{$block/@Name}" />
let $gain :=
    for $block in $doc // Block [ @BlockType = "Gain" ]
    return <gain id="{$block/@SID}" name="{$block/@Name}" />
let $subSystem :=
    for $block in $doc // Block [ @BlockType = "SubSystem" ]
    return <subSystem id="{$block/@SID}" name="{$block/@Name}" />
let $block := ($signalGenerator,$sum,$gain,$subSystem)
let $port :=
    for $block in $doc // Block
    for $port in $block / Port
    return <port blockId="{$block/@SID}" name="{$port/P[@Name="Name"]/text()}">
    number="{$port/P[@Name="PortNumber"]/text()}">
    let $portRel :=
        for $line in $doc // Line
        let $fromBlockId := analyze-string($line/P[@Name="Src"]/text(), "(\d+)#")
        / fn:match / fn:group [@nr = "1"] / text()
        let $toBlockId := analyze-string($line/P[@Name="Dst"]/text(), "(\d+)#")
        / fn:match / fn:group [@nr = "1"] / text()
        let $fromPortIdx := analyze-string($line/P[@Name="Src"]/text(), ".\*:\/(\d+)")
        / fn:match / fn:group [@nr = "1"] / text()
        let $toPortIdx := analyze-string($line/P[@Name="Dst"]/text(), ".\*:\/(\d+)")
        / fn:match / fn:group [@nr = "1"] / text()
        for $p1 in $port[@blockId = $fromBlockId and @number = $fromPortIdx]
        for $p2 in $port[@blockId = $toBlockId and @number = $toPortIdx]
        return <portRel fromBlockId="{$fromBlockId}" toBlockId="{$toBlockId}"$
    / fromPortId="{$fromPortIdx}" toPortId="{$toPortIdx}" />
    return <results> {$signalGenerator} {$sum} {$gain} {$subSystem}$
    {$port} {$portRel} {$blockRel} </results>

Figure A.15: XQuery program written by Subject A for completion of Task 5 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
A.4  
mel and XQuery programs written by Subject B

```scheme
Function(X) |- ownedArchitectures{name="Logical Architecture"}
=>ownedFunctions{"xsi:type"[".*:(.*)"]="LogicalFunction", name:X};

Component(X) |- ownedArchitectures{name="Logical Architecture"}
=>ownedLogicalComponents{"xsi:type"[".*:(.*)"]="LogicalComponent",
name:X};

Flow(X, Y) |- ownedFunctionalExchanges{source['#(.*)']:SOURCE, target['#(.*)']:TARGET,
ownedFunctions{id:SOURCE, name:X},
ownedFunctions{id:TARGET, name:Y};
```

Figure A.16: mel program written by Subject B for completion of Task 1 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
let $part1 :=
<part1>{
  for $x in doc("in-flight-ent-system")//ownedArchitectures
  where $x/@name="Logical Architecture"
  for $y in $x//ownedFunctions
  where analyze-string($y/@xsi:type, "[-a-zA-Z0-9_-]+$")/fn:match = "LogicalFunction"
  return <function name="{$y/@name}"/>
}
</part1>

let $part2 :=
<part2>{
  for $x in doc("in-flight-ent-system")//ownedArchitectures
  where $x/@name="Logical Architecture"
  for $y in $x//ownedLogicalComponents
  where analyze-string($y/@xsi:type, "[-a-zA-Z0-9_-]+$")/fn:match = "LogicalComponent"
  return <component name="{$y/@name}"/>
}
</part2>

let $part3 :=
<part3>{
  for $x in doc("in-flight-ent-system")//ownedFunctionalExchanges
  let $src := analyze-string($x/@source, "[-a-zA-Z0-9_-]+$")/fn:match
  let $tgt := analyze-string($x/@target, "[-a-zA-Z0-9_-]+$")/fn:match
  for $y in doc("in-flight-ent-system")//ownedFunctions
  where $y/@id = $src
  for $z in doc("in-flight-ent-system")//ownedFunctions
  where $z/@id = $tgt
  return <flow source="{$y/@name}" target="{$z/@name}" name="{$x/@name}"/>
}
</part3>

return <ans>
{$part1}
{$part2}
{$part3}
</ans>

Figure A.17: XQuery program written by Subject B for completion of Task 1 as described in Section 4.2.2.
StateMachine(X) |- ownedBehavior{"xmi:type"="uml:StateMachine", name:X};

Region(X, Y) |- ownedBehavior{"xmi:type"="uml:StateMachine", name:X}
  =>region{"xmi:type"="uml:Region", name:Y};

Owns(X, Y) |- ownedBehavior{"xmi:type"="uml:StateMachine", name:X}
  =>region{"xmi:type"="uml:Region", name:Y};

Owns(X, Y) |- subvertex{name:X}
  =>region{"xmi:type"="uml:Region", name:Y};

State(X, Y) |- ownedBehavior{"xmi:type"="uml:StateMachine", name:X}
  =>subvertex{"xmi:type"="uml:State", name:X};

Contains(X, Y) |- ownedBehavior{"xmi:type"="uml:StateMachine"}
  =>subvertex{"xmi:type"="uml:State", name:X}
  =>subvertex{"xmi:type"="uml:State", name:Y};

Transition(X) |- transition{"xmi:type"="uml:Transition", name:X, target:TARGET, source:SOURCE}
  ->[
    @trigger{event:EVENT},
    @ownedRule{}->specification{value:guard},
    @effect{}->body{mel.contents:effect}
  ],

@packagedElement{"xmi:type"="uml:SignalEvent", "xmi:id":EVENT, signal:SIGNAL},
@packagedElement{"xmi:type"="uml:Signal", "xmi:id":SIGNAL,
  name:trigger},
subvertex{"xmi:type"="uml:State", "xmi:id":SOURCE},
subvertex{"xmi:type"="uml:State", "xmi:id":TARGET};

Figure A.18: mel program written by Subject B for completion of Task 3 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
xquery version "3.1";
declare namespace xmi="http://schema.omg.org/spec/XMI/2.1";
let $part1 :=
<part1>
  for $x in doc("HomeAlarmWithPorts.xmi")//ownedBehavior
  where $x/@xmi:type="uml:StateMachine"
  return <StateMachine name = "{$x/@name}"/>
</part1>

let $part2 :=
<part2>
  for $x in doc("HomeAlarmWithPorts.xmi")//ownedBehavior
  where $x/@xmi:type="uml:StateMachine"
  for $y in $x//region
  return <Region machine ="{$x/@name}" name="{$y/@name}"/>
</part2>

let $part3 :=
<part3>
  for $x in doc("HomeAlarmWithPorts.xmi")//(ownedBehavior | subvertex)
  for $y in $x//region
  return <Owns state="{$x/@name}" region="{$y/@name}"/>
</part3>

let $part4 :=
<part4>
  for $x in doc("HomeAlarmWithPorts.xmi")//ownedBehavior
  where $x/@xmi:type="uml:StateMachine"
  for $y in $x//subvertex
  where $y/@xmi:type="uml:State"
  return <State machine="{$x/@name}" name="{$y/@name}"/>
</part4>

let $part5 :=
<part5>
  for $x in doc("HomeAlarmWithPorts.xmi")//ownedBehavior
  where $x/@xmi:type="uml:StateMachine"
  for $y in $x//subvertex
  where $y/@xmi:type="uml:State"
  for $z in $y//subvertex

Figure A.19: XQuery program written by Subject B for completion of Task 3 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
Figure A.20: mel program written by Subject B for completion of Task 4 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
xquery version "3.1";
let $part1 :=
<part1>
{
for $x in doc("royalandloyal")//eClassifiers
where $x/@xsi:type = "ecore:EClass"
let $abstract := if (exists($x/@abstract)) then data($x/@abstract) else "false"
let $interface := if (exists($x/@interface)) then data($x/@interface) else "false"
return <class name="{data($x/@name)}" abstract="{$abstract}" interface="{$interface}"/>
}
</part1>
let $part2 :=
<part2>
{
for $x in doc("royalandloyal")//eClassifiers
where $x/@xsi:type="ecore:EClass"
for $y in $x/eStructuralFeatures
where $y/@xsi:type="ecore:EAttribute"
let $type := analyze-string(data($y/@eType), "[-a-zA-Z0-9_]+")
let $derived := if (exists($y/@derived)) then data($y/@derived) else "false"
return <membership class="{data($x/@name)}" member="{data($y/@name)}" type="{data($type/fn:match)}" derived="{$derived}"/>
}
</part2>
let $part3 :=
<part3>
{
for $x in doc("royalandloyal")//eClassifiers
where $x/@xsi:type="ecore:EClass"
let $st := analyze-string(data($x/@eSuperTypes), "[-a-zA-Z0-9_]+")
return if (exists($x/@eSuperTypes)) then<superclass sub="{data($x/@name)}" super="{data($st/fn:match)}"/> else ()
}
</part3>
let $part4 :=
<part4>
{
for $x in doc("royalandloyal")//eClassifiers
where $x/@xsi:type="ecore:EClass"
for $y in $x/eStructuralFeatures
where $y/@xsi:type="ecore:EReference"
where not(exists($y/@containment))
let $upper := if (exists($y/@upperBound)) then if (data($y/@upperBound) = "-1")
then "*" else data($y/@upperBound) else "1"
let $lower := if (exists($y/@lowerBound)) then data($y/@lowerBound) else "0"
let $derived := if (exists($y/@derived)) then data($y/@derived) else "false"
return <association class="{data($x/@name)}" other="{data($y/@name)}" lower="{$lower}" upper="{$upper}" derived="{$derived}"/>
}
</part4>

let $part5 :=
<part5>
for $x in doc("royalandloyal")//eClassifiers
where $x/@xsi:type="ecore:EClass"
for $y in $x/eStructuralFeatures
where $y/@xsi:type="ecore:EReference"
where $y/@containment = "true"
let $upper := if (exists($y/@upperBound)) then if (data($y/@upperBound) = "-1") then "*" else data($y/@upperBound) else "1"
let $lower := if (exists($y/@lowerBound)) then data($y/@lowerBound) else "0"
let $derived := if (exists($y/@derived)) then data($y/@derived) else "false"
return <containment class="{data($x/@name)}" other="{data($y/@name)}" lower="{$lower}" upper="{$upper}" derived="{$derived}"/>
}
</part5>

let $part6 :=
<part6>
for $x in doc("royalandloyal")//eOperations
let $type := analyze-string(data($x/@eType), "[-a-zA-Z0-9_]+")
return <method name="{$x/@name}" returnType="{$type/fn:match}"/>
</part6>

let $part7 :=
<part7>
for $x in doc("royalandloyal")//eOperations
for $y in $x/eParameters
let $type := analyze-string(data($y/@eType), "[-a-zA-Z0-9_]+")
return <hasParam methodName="{$x/@name}" paramName="{$y/@name}" type="{$type/fn:match}"/>
</part7>
return <ans>{$part1}
{$part2}
{$part3}
{$part4}
{$part5}
{$part6}
Figure A.21: XQuery program written by Subject B for completion of Task 4 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
Figure A.22: mel program written by Subject B for completion of Task 5 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
let $part1 :=
<part1>
  {
    for $x in doc("system_root")//Block
    return <Block type="{$x/@BlockType}" name="{$x/@Name}"/>
  }
</part1>

let $part2 :=
<part2>
  for $x in doc("system_root")//Block//Port/P
  where $x/@Name = "Name"
  return <Port name="{data($x)}"/>
</part2>

let $part3 :=
<part3>
  for $x in doc("system_root")//Block
  for $y in $x//Port/P
  where $y/@Name = "Name"
  return <Owns Block="{$x/@Name}" Port="{data($y)}"/>
</part3>

let $part4 :=
<part4>
  for $x in doc("system_root")//Block
  for $y in doc("system_root")//Line
  for $z in $y/P
  where $z/@Name = "Src" and analyze-string(data($z),"^[0-9]+")/fn:match = $x/@SID
  for $w in $y/(if (Branch) then Branch/P else P)
    (:for $w in $y/P:)
  where $w/@Name = "Dst"
  for $u in doc("system_root")//Block
  where analyze-string(data($w),"^[0-9]+")/fn:match = $u/@SID
  return <Flow Src = "{$x/@Name}" Dst="{$u/@Name}"/>
</part4>
return <ans>
{$part1}
{$part2}
{$part3}
{$part4}
</ans>
Figure A.23: XQuery program written by Subject B for completion of Task 5 as described in Section 4.2.2. Some additional line breaks have been added to allow the program to fit on the page.
Appendix B

Diagrams of Models Used in Studies
B.1 UML StateMachine Diagrams

Figure B.1: StateMachine diagram for an Arduino stopwatch system (YAKINDU).
Figure B.2: StateMachine diagram for a polling system meant to be implemented directly on hardware (YAKINDU).
Figure B.3: StateMachine diagram for a blinking LED (YAKINDU).
Figure B.4: StateMachine diagram for a Coffee Machine (YAKINDU).
Figure B.5: StateMachine diagram for a motion detector (YAKINDU).
Figure B.6: StateMachine diagram for a music player (YAKINDU).
Figure B.7: StateMachine diagram for a smart home system (YAKINDU).
Figure B.8: StateMachine diagram for a traffic light controller (YAKINDU).
Figure B.9: StateMachine diagram for an autonomous robot (YAKINDU).
Figure B.10: UML Class diagram for a video game named Advent (ArgoUML).
Figure B.11: UML Class diagram for a rotary button system (ArgoUML).
Figure B.12: UML Class diagram for a dungeon crawler video game (ArgoUML).
B.2.2 Eclipse Modelling Tools UML Class Diagrams

Figure B.13: UML Class diagram for a coffee brewing system (Eclipse Modelling Tools).
Figure B.14: UML Class diagram for a Java compiler (Eclipse Modelling Tools).

Figure B.15: UML Class diagram for a task management system (Eclipse Modelling Tools).
Figure B.16: UML Class diagram for a retail store system (Eclipse Modelling Tools).

Figure B.17: UML Class diagram for Model to code compilation software (Eclipse Modelling Tools).
Figure B.18: UML Class diagram for an alarm system (MagicDraw).
Figure B.19: UML Class diagram for a library booking system (MagicDraw).
Figure B.20: UML Class diagram for a music management system (MagicDraw).
Figure B.21: UML Class diagram for a parking lot gate system (MagicDraw).
B.2.4 UMLDesigner UML Class Diagrams
Figure B.22: UML Class diagram for a custom video game engine (UMLDesigner).
Figure B.23: UML Class diagram for a race management system (UMLDesigner).
B.3 Simulink Block Diagrams

Figure B.24: Simulink Block diagram for an analog to digital converter quantization algorithm (Simulink).
Air-Fuel Intake Dynamics

Figure B.25: Simulink Block diagram for the dynamics of air-fuel intake (Simulink).

Modeling an Anti-Lock Braking System (ABS)

Figure B.26: Simulink Block diagram for an anti-lock brake system (Simulink).
Figure B.27: Simulink Block diagram for a sample anti-windup PID control system (Simulink).

Figure B.28: Simulink Block diagram for a bouncing ball (Simulink).
Figure B.29: Simulink Block sample diagram for component-based Modelling (Simulink).

Figure B.30: Simulink Block diagram for a fault-tolerant fuel control system (Simulink).
Figure B.31: Simulink Block diagram for intake airflow estimation and closed loop correction (Simulink).

Figure B.32: Simulink Block diagram for a PID tuning system, with Figures B.33 and B.34 (Simulink).
Figure B.33: Simulink Block diagram for a PID tuning system, with Figures B.32 and B.34 (Simulink).

Figure B.34: Simulink Block diagram for a PID tuning system, with Figures B.33 and B.32 (Simulink).
B.4 Arcadia Diagrams

Figure B.35: Arcadia diagram for a train crossing system.
Figure B.36: A second Arcadia diagram from a train crossing system.
Figure B.37: Arcadia diagram for a security system.
Figure B.38: Arcadia diagram for an unknown system.
B.5 Feature Models

Figure B.39: Feature Model diagram for a bicycle configurator system (FeatureIDE).

Figure B.40: Feature Model diagram for a simple car Model (FeatureIDE).
Figure B.41: Feature Model diagram for a desktop searching system (FeatureIDE).

Figure B.42: Feature Model diagram for an elevator system (FeatureIDE).

Figure B.43: Feature Model diagram for programming expressions expressed as a product line (FeatureIDE).
Figure B.44: Feature Model diagram for a simple Hello World program (FeatureIDE).
Figure B.45: Feature Model diagram for the landing gear of a plane (FeatureIDE).

Figure B.46: Feature Model diagram for a game of Poker (FeatureIDE).
Figure B.47: Feature Model diagram for a game called TankWar (FeatureIDE).

Figure B.48: Feature Model diagram for fictional characters (FeatureIDE).