mel- Model Extractor Language for Extracting Facts from Models

Robert Hackman, Joanne M. Atlee, Alistair Finn Hackett, Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA
{r2hackma,jmatlee,afhacket,migod}@uwaterloo.ca

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 code base 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.

This paper introduces *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 *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 *mel* to create a fact extractor for a particular Model representation, than to develop a specialized fact extractor for the Model from scratch.

CCS CONCEPTS

• **Software and its engineering → System modeling languages**; *Unified Modeling Language (UML).*

KEYWORDS

Software models, Fact extraction, Domain-specific languages

ACM Reference Format:
Robert Hackman, Joanne M. Atlee, Alistair Finn Hackett, Michael W. Godfrey. 2020. *mel*- Model Extractor Language for Extracting Facts from Models. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410964

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 [26], Doop [4], Rigi [25], Frappé [7], eKNOWS Code Model Service [29], Rex [27]) that extract software models (e.g., call graphs, data-flow graphs, dependency graphs) from software artifacts. More general 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 [14], graph queries [7, 29], Datalog programs [4], 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[1]). Other components may be generated automatically from Models that are more descriptive and semantically informative than the generated code. In order to create a single coherent model of the entire system, we would need a means of extracting facts from those components that are expressed as 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) [5] and Model editors for the same Model types (e.g., UML Model editors) use the XML Metadata Interchange (XMI) [9] standard for expressing metamodels and metadata, the Models' XMI schema vary considerably. A specialized extractor would need to be created for every Model type supported by every Model editor. A secondary challenge is how to link together the facts from different fact extractors. Linking is less of a challenge among data extracted from code-based artifacts, where names to be linked are typically exact matches. References to the same Model elements are less likely to have the same name in all Models, especially when the Models are used as specifications (cf. are the input to code generation).

In this paper, we introduce 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) version 2.5.1 standard [9]. *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.

[1]Hereafter, we distinguish between *prescriptive and descriptive models* (denoted as capitalized *Models*) that are produced by engineers as part of the software-development process versus *extracted models* (denoted as lowercase *models*) that are automatically generated from software, Models, and other artifacts.

MoDELS '20, October 18–23, 2020, Montréal, QC, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410964

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.

Contributions: This paper makes the following contributions: (1) We present the DSL *mel* and interpreter `mint` for specifying and extracting model-based information from a Model represented in XML/XMI. (2) We evaluate the extent to which *mel* is expressive enough to apply to different Model types, including UML Class and state machine diagrams [10], Arcadia Logical Architecture diagrams [32], and Feature Models [20]. (3) We also assess 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 traditional parsing technologies.

The rest of this paper is organized as follows. Section 2 starts with a background on model extraction, facts, and XML representations for Models. In Section 3, we introduce *mel*, specify its input language, and describe its implementation. In Section 4, we report on our studies to evaluate the effectiveness of *mel* in easing the creation of new model extractors for Models and extent of *mel*'s ability to generalize to a variety of XML/XMI-based Models. We discuss evaluation results and threats to validity in Section 5, related works in Section 6, and conclusions in Section 7.

2 BACKGROUND

2.1 Extraction of Lightweight Models

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 [3, 24] and the validation of developers' mental models of a system's design against the code [26]. 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 artifacts, and there is a central model of the system into which this information is integrated. Adding this information is important, because it improves our existing knowledge about the system, and permits new kinds of analyses to be performed. In our work here, we propose a tool to aid in augmenting these system models with facts extracted from software Models, such as UML Class and StateMachine diagrams, Arcadia Logical Architecture diagrams, and Feature Models. To the best of our knowledge, this has not been done before.

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.

The factbases typically 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`.

2.2 XML/XMI

Many Modelling languages have a graphical representation, but Modelling tools represent this as a textual representation for storing and sharing Models. The most common textual representation for Models is a eXtensible Markup Language (XML) [5] schema, such as a variant of the OMG XML Metadata Interchange (XMI) standard [9]. The XMI standard is sufficiently expressive to represent Models belonging to any MOF-based metamodel.

An XML/XMI representation is a tree of nested tagged *elements*, with a single root element. Each element comprises a *start-tag*, *content*, and matching *end-tag*: `<tagname> content </tagname>`. An element's content can contain text, nested elements, or both. Attribute values can also be stored as attributes of the tag. When an element conveys all of its information as attributes, it is useful to express it as an *empty element* that is simply a self-closing tag with attributes and no content. Seen below is a tiny XMI Model that declares a UML Class named "C1", which has a feature attribute expressed as an empty element:

```
<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>
```

3 MODEL EXTRACTOR LANGUAGE (MEL)

In this section 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 XML as specified in the XML Metadata Interchange (XMI) version 2.5.1 standard [9]. 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.

A *mel* program is fed into the `mint` interpreter to create a specialized fact extractor for a particular Model representation. The extractor takes as input a Model expressed in XML/XMI and produces a result set comprising extracted *facts* about the Model: *entities* (which represent Model elements in the input Model), *relations* between entities, and *attributes* of entities and relations.

3.1 Research Methodology

The *mel* language and extractor have been developed using a design science methodology [13], whereby *mel* was initially designed to

```
<SWModel>
  <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 1: A simple Model expressed in XML.

```
component(C) |- compFeatNode{component:C};
feature(X) |- compFeatNode{id:X, userName:label};
featParent(C,F) |- compFeatNode{component:C, id:F};
$sibling(F,G) |- featParent(F,C), featParent(G,C);
```

Figure 2: A sample *mel* program.

generalize a special-purpose extractor for Acadia logical architecture Models to apply more generally to Models represented in XMI; *mel* has evolved as new language features were needed to extract facts from additional subject Models. This paper introduces *mel* after it has evolved to support extraction of Models represented in XMI and has been tested on UML Class diagrams, Rhapsody Statechart diagrams, and FeatureIDE Feature Models. In Section 4, we assess how well this version of *mel* generalizes to Models represented in XMI, and we discuss how the evaluation results suggest additional language features that would either ease the development of *mel* programs or would extend its expressivity to previously unsupported XMI formats. In Section 5, we discuss our initial ideas for extending *mel* to support the extraction of facts from Models represented more generally in XML.

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 followed by the production rules 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 2. A fact type with one parameter is an *entity* type; its parameter is the unique identifier for an extracted entity of that type. A fact type with two parameters is a *relationship* type; its parameters refer to the two entities that are being related. On the right-hand side, a production rule can refer to XML elements in the input Model (as in the first three rules) or to fact types that have already been declared (as in the fourth rule). If the former, the production rule starts with an XML tag followed by a list of bindings surrounded by curly braces. Each binding is of the form "X:Y" where X is the name of an XML Attribute[2] inside of the XML tag in the Model, and Y is the name of one of a fact type's parameters or is a declaration of an attribute of the fact type.

To understand how *mel* declarations extract facts from a Model, consider the small pedagogical Model given in Figure 1, in which component `CompA` contains features `F1` and `F3`, and component `CompB` contains feature `F2`. Suppose that the *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 *mel* program in Figure 2 comprises four definitions that correspond to the four types of facts to be extracted.

[2]As with Model vs. model, we distinguish between XML Attributes in the input Model from attributes in the extracted model by capitalizing all references to the former.

Table 1: Results of Sample *mel* Program on Simple Model

	Fact Type	Parm1	Parm2
Entities	component	CompA CompB CompC	
	feature	F1 F2 F3	
Relations	featParent	F1 F2 F3	CompA CompB CompA
	sibling	F1	F3
Attributes	F1 F2 F3	label label label	Feature 1 Feature 2 Feature 3

component(C) This declaration extracts `component` entities, whose parameter `C` denotes the extracted entity's identifier. The rule matches instances of the XML tag `compFeatNode` that have an XML Attribute named "component". The Attribute "component" must be present in the XML tag instance for an entity instance to be extracted; the Attribute's value is bound to the extracted `component`'s parameter `C`. The results of applying this declaration on the Model from Figure 1 are shown in the first three rows of Table 1.

feature(X) This declaration extracts `feature` entities, and may also extract associated attributes. The rule matches all instances of the XML tag `compFeatNode` that have an XML Attribute "id". If the matching XML tag instance also has an Attribute "userName", then an entity attribute is also extracted, which includes the entity's identifier, the attribute name `label`, and the value of the XML Attribute "userName". The results of this declaration on our Model are shown in the second three rows and last three rows of Table 1.

featParent(C,F) This declaration specifies a relation between `components` and their `features`. The rule matches instances of the XML tag `compFeatNode` that have an XML Attribute "component" and an XML Attribute "id". Each extracted `featParent` relates a (component) entity `C` (the value of XML Attribute "component") to a (feature) entity `F` (the value of XML Attribute "id"). The results of this declaration on our Model are shown in rows 7-9 of Table 1.

$sibling(F,G) This declaration extracts pairs of `feature` entities that have the same parent `component`. Unlike previous declarations, the rule does not refer to XML tags in the Model. Instead it refers to the previously defined `featParent` relationship, and it relates the `feature` entities of two facts if their `component` entities match. The prefix $ asserts that the `sibling` relation is anti-reflexive, meaning that the result set will not include any `sibling` instances that relate a `feature` to itself. The result of this declaration is row 10 of Table 1.

There are four additional features of *mel* that are worth noting that add to the user's ability to (1) filter the set of XML tags that match a *mel* declaration, (2) extract information from XML trees rather than tags, (3) elide the results of *intermediate rules* from the result set, and (4) extract a substring from an XML Attribute value.

3.2.1 Filter Matching XML Tags. A user may want to extract a subset of the matching XML tags in a Model. In the examples above, *attribute bindings* play the role of such a filter. However, a user may also want to constrain matching XML tag instances based on their Attribute values without necessarily binding those values. *mel* supports this concept via *attribute requirements* that specify constraints on an XML tag's Attribute values that must be met in order for an instance of the tag to match the fact type declaration. For example, the following declaration would extract only `feature` entities that belong to component "CompA":

```
feature(X) |- compFeatNode{component="CompA", id:X};
```

3.2.2 XML Trees of Elements. So far, we have only discussed extracting data from XML tags and their Attributes. However, other data of interest pertain to how XML tags are related to one another in a Model's XML tree. The most common is the *parent* relationship, in which a (child) XML tag is nested within a (parent) element:

```
<component id="CompA">
    <compFeatNode id="F1" userName="Feature 1"/>
    <compFeatNode id="F3" userName="Feature 3"/>
</component>
```

mel provides a function `mel.parent(P,C)` for extracting data from Model elements that are in a Parent/Child relationship. Given the above XML schema, the *mel* declaration for `featParent` would be:

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

This rule matches only instances of the XML tag `component` that have an XML Attribute "id" and have a child tag `compFeatNode` with its own XML Attribute "id".

3.2.3 Elide Results of Intermediate Rules. The *mel* user may want to declare and extract some 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 the same `component`, but not have facts about `featParent` relations appear in the final result set. If the declaration in Figure 2 were replaced by the following declaration

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

then the results of applying this modified *mel* program to the XML Model from Figure 1 would be as shown in Table 1, but with the contents of rows 7-9 omitted.

3.2.4 Extract a Substring from an XML Attribute Value. XML Attribute values sometimes contain extraneous characters that should not be included in facts. For example, elements in a Capella XML Model reference other elements by prefixing their identifiers with the symbol "#". To avoid exacting such symbols as part of an Attribute value, the user can append the pattern `[<prefix>%<suffix>]` to the Attribute name, where `<prefix>` and `<suffix>` are undesired substrings to be stripped away; the value extracted is the remaining substring that matches the wildcard character "%". For example in Figure 1, suppose that the `compFeatNode` `id` Attribute values have a prefix "F"; these prefixes could be removed by rewriting the `feature` rule from Figure 2 as follows:

```
feature(X) |- compFeatNode{id["F%"]:X, userName:label};
```

Such a definition, applied to the XML Model from Figure 1, would extract three `feature` entities with identifiers "1", "2", "3".

program	::=	*decl*+
decl	::=	[*mods*] *name* '(' *param* ')' '|-' *rules* ';'
mods	::=	('.' | '$' | '^' | '@')*
param	::=	*name* [',' *name*]
rules	::=	*rule* (',' *rule*)*
rule	::=	*xmlTag* | *fact* | *function*
xmlTag	::=	*id* '{' [*attreqs*] *bindings* '}'
attreqs	::=	(*attreq* ',')*
attreq	::=	*lvalue* [*xwordx*] *op* *literal* | *eReq* '(' *lvalue* ')'
eReq	::=	'mel.exists' | 'mel.nexists'
bindings	::=	*binding* (',' *binding*)*
binding	::=	*lvalue* [*xwordx*] ':' *id*
xwordx	::=	'"' CHAR* '%' CHAR* '"'
op	::=	'=' | '!='
fact	::=	*name* '(' *param* ')' '{' *bindings* '}'
function	::=	*melFunction* '(' *xmlTag* ',' *xmlTag* ')'
melFunction	::=	'mel.parent' | 'mel.ancestor'
lvalue	::=	*id* | 'mel.contents'
id	::=	*name* | *string*
name	::=	ALPHA (ALPHA | DIGIT | '_')*
literal	::=	*string* | *int*
string	::=	'"' CHAR* '"'
int	:=	DIGIT+

Figure 3: 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[3] for *mel* is provided in Figure 3. A *program* is a sequence of *decl*arations. Each *decl*aration defines either an entity type (with one *param*eter) or a relationship type (with two *param*eters). A *decl*aration prefaced with *mod*ifer '.' 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. *Decl*aration *mod*ifers "$" and "^" declare that the relationship type cannot be reflexive or commutative, respectively.

Declaration *rule*s can refer to *xmlTag*s in the Model, previously declared *fact* types, or the predefined *function* `mel.parent`. A *rule* that refers to an *xmlTag* may have requirements on the values of the xmlTag's attribute (*attreqs*); specifically, an XML tag matches a specified *xmlTag* with an attribute requirement *attreq* only if the name of the XML tag matches the *xmlTag* name, the *attreq* attribute exists within the tag, and its value is equal "=" (or not equal "!=", depending on the requirement) to the specified *literal* value. A *rule* that refers to an *xmlTag* will also have *binding*s that map some of

[3]Elements in the grammargray text were introduced to the mel language in response to evaluation findings; their descriptions are deferred to the Evaluation section.

```
Model : AbstractClass "My Class" {
    code : CClass "MyClass"
}
Model : AbstractComponent "Control" {
    code : * "control::*"
}
```

Figure 4: Sample linkage data file.

the *xmlTag*'s attribute values to *param*eters and attributes of the extracted fact. Sometimes, prefix or suffix characters may need to be stripped away from the extracted value of an xmlTag's attribute (non-terminal *xwordx*). A *decl*aration *rule* can refer to a previously declared *fact* type, thereby using extracted facts and their attribute values to find other related facts to include in the output. A *rule* that refers to the predefined *function* `mel.parent` defines a parent relationship between the first *xmlTag* parameter (the parent) and the second *xmlTag* parameter (the child); the *xmlTag*s might have *attreqs* if there are attribute requirements on the parent relationship, and they might have *binding*s if, for example, the goal is to extract an attribute value from a parent element in the Model. Towards the bottom of the grammar in Figure 3 are rules on what constitutes a valid identifier, literal value, and quoted string value.

3.4 Linking Facts from Multiple Sources

A large software system is likely to comprise a combination of source-code components, procured components (ideally specified by Models), and components generated or configured by Models. In order to create a single coherent model of the entire system, facts would need to be extracted from each component and combined in a shared repository. It is often inappropriate to simply merge the facts extracted from multiple sources, for two reasons:

(1) **Name Mismatch Problem** — Elements that are referenced in multiples sources may not have the same name in all sources. Models typically have more relaxed naming conventions than programming languages and often opt to express names in natural language, complicating the task of relating entities in Models to their counterparts in code artifacts.
(2) **Abstraction/Granularity Mismatch Problem** — Model elements are often at a higher level of abstraction than their analogous source-code elements, which means that their correspondence may not be one-to-one.

A separate tool is used to link the facts from two factbases: facts that have the same (expanded) identifier are linked automatically; and facts that are related but have mismatched names or levels of granularity are linked with the help of a user-provided *linkage data* file. A linkage data file explicates the correspondence between names of elements from different factbases and explicates relationships between elements from different factbases.

A simple linkage data file is shown in Figure 4. This file declares a one-to-one correspondence between fact "My Class" of type "AbstractClass" from factbase "Model" and fact "MyClass" of type "CClass" from factbase "code". The linkage data file also declares a one-to-many correspondence between fact "Control" of type "AbstractComponent" from factbase "Model" and (possibly many) "code" facts of any type with any identifier prefixed by "control::" (the character"*" is a wildcard that matches any string).

Asking the user to provide a linkage data file that maps each name in one artifact to each name in another artifact may seem like a major request. However, the size of this task depends on the degree of overlap between the factbases' software artifacts, and the degree of mismatch between names. If the factbases being linked are extracted from Models and source code for the same software component (e.g., for compliance analyses), then the overlap between factbases is considerable and the linkage data file may be lengthy. If the factbases being linked are extracted from artifacts for different components, then the linkage data file is limited to the names of elements that are common among the components' interfaces.

4 EVALUATION OF *MEL*

The evaluation of *mel* focuses on two criteria:

(1) The extent to which *mel* generalizes to XML/XMI-based representations of Models and to desired extractions
(2) The effectiveness of *mel* in easing the creation of new fact extractors for Models

The evaluation of *mel* was completed by the author of the tool, which is discussed under threats to validity in Section 5.

All data for the first study, including graphical representations of Models, XMI representations of Models, *mel* programs, and result sets from applying the *mel* programs to the Models' XMI representations, can be found at https://github.com/Roshack/MODELS2020. As the images of some Models are not included in this paper, the repository acts as an online appendix that includes those images.

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 that adhere to the OMG standard for XMI [9]. 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, 15, 18, 19]; and whose XMI representations varied from each other. Finding suitable Models for the study was a challenge because of the dearth of publicly available sophisticated Models. In the end, we often selected Models that were provided as tutorial examples in Tool downloads because they tended to exemplify multiple Modelling-language features.

The study applied *mel* to one UML StateMachine diagram (exported from IBM Rhapsody [16]), two UML Class diagrams exported from different tools (MagicDraw [17] and UMLDesigner [28]), one Arcadia Logical Architecture diagram (exported from Capella[11]), and one Feature Model (exported from FeatureIDE [20]). All of the relevant facts from each model were extracted, with the types of facts extracted shown in figure 1 Each of the Models is relatively small, ranging in numbers of entities, relationships, and attributes from the tens to the hundreds. The sizes of each Model in terms of numbers of facts extracted is provided in Figure 2. However, the size of the Models is inconsequential to our study as the evaluation of expressiveness requires Models with diverse XMI representation and Model elements, not size.

The correctness of the *mel* extractor was evaluated as follows. *Recall* was checked by comparing the elements in the Models against the data extracted by *mel* and confirming that all of the Model entities, relationships, and attributes targeted for extraction were

Table 2: Sizes of Models and types of facts extracted

Model Type	Entity Type (count)	Relationship Type (count)	Attribute Type (count)
Arcadia	Function (219), Component (18), Data (262)	Containment (30), DataFlow (221)	label (761), DataLabel (221)
MagicDraw Class Diagram	Class (150), Field (48), Method (163) Parameter (135), Interface (11)	Inheritance (87), FieldOf (104), MethodOf (154), ParameterOf (135), Composition (3)	Multiplicities (12), ReturnType (66), Static (8), Const (15), Visibility (211), Type (144)
UMLDesigner Class Diagram	Class (8), Field (11)	typeOf (10), FieldOf (11), Association (4), Composition (8)	Multiplicities (19), Type (11), RoleNames (16)
UML StateMachine Diagram	State Machine (4), Region (20), State (52)	Sibling (3), Contains (44), Transition (56)	Guard (12), Trigger (21), Effect (8), Type (52)
Feature Model Diagram	Feature (21)	FeatureOf (20), MutuallyExclusive (4), RequiredTogether (13)	isAbstract (12), isMandatory (4)

indeed extracted. *Precision* was checked by randomly selecting 30 facts (entities or relationships) from each extracted factbase and confirming that the extracted elements and their attributes were correct. Recall and precision were both 100%.

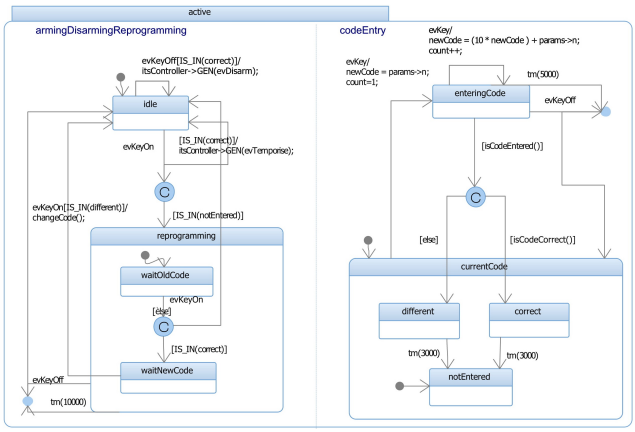

Figure 5: UML StateMachine diagram of keypad (Rhapsody).

4.1.1 UML StateMachine Diagrams. The UML StateMachine shown in Figure 5 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 download [16]. A Rhapsody Model comprises all of the diagrams in a UML project. Thus to restrict extraction to a particular diagram (e.g., the StateMachine depicted in Figure 5), the declaration for `StateMachine` includes an Attribute requirement that Attribute `name` have value "statechart_22". Subsequent definitions require that extracted facts be relevant to "statechart_22" or to facts extracted from "statechart_22".

In general, many elements in an XMI representation have the same XMI tag and use XMI Attribute values to specify the elements' types. For example, the XMI tag `packagedElement` may refer to several entity types in a UML StateMachine. In our Model, we are interested in `packagedElement` entities whose `xmi:type` Attribute has value "uml:SignalEvent" or "uml:Signal". Instances of these tags combined with related instances of the XML tag `trigger` (children of a `transition` tag) identify the trigger event of that transition.

An outcome of this study was the addition of three features to the *mel* language. The `Region` declaration uses a new function

```
<region xmi:id="_mapNhJ" name="armingDisarmingReprogramming">
  . . .
  <subvertex xmi:id="161" name="idle"/>
  . . .
  <transition xmi:id="170" target="161" guard="175" source="161">
    <ownedRule xmi:id="175" context="OLDID+1424988+170">
      <specification xmi:id="_mapNqp" value="IS_IN(correct)"/>
    </ownedRule>
    <effect xmi:id="_mapNq5">
      <body>&#xD;\nitsController->GEN(evDisarm);</body>
    </effect>
    <trigger xmi:id="174" event="1d0f6"/>
  </transition>
        . . .
</region>
. . .
<packagedElement xmi:type="uml:SignalEvent" xmi:id="1d0f6" signal="_marCdp"/>
<packagedElement xmi:type="uml:Signal" xmi:id="_marCdp" name="evKeyOff"/>
```

Figure 6: Snippet of the XMI representation of the StateMachine Model shown in Figure 5. Identifiers have been abbreviated and non-referenced Attributes have been omitted.

`mel.ancestor`, which is equivalent to repeated applications of the `mel.parent` function. The sixth rule in the `transition` declaration uses a new function `mel.contents` to extract the content of an XML element. In our StateMachine Model, a transition's actions reside within an XML element between start- and end-tags. The `mel.contents` function extracts the XML element's content as a string; the rules then bind this value to a transition's `effect`.

The third feature added to *mel* by this study is the concept of *optional rules*, designated with a prefix "@". If a declaration has two normal rules that have Attribute bindings to the same *mel* attribute, then the result set can only include pairs of entities (extracted by the two rules) that agree on the value of the shared attribute. In contrast, if one of the rules is an optional rule, then (1) for every entity extracted by the normal rule that can be paired with an entity extracted from the optional rule (where shared attributes have the same values), the combined information is included in the result set; and (2) for every remaining entity extracted by the normal rule that does not match the optional rule, the information extracted is added to the result set without considering the optional rule.

The declaration of `transition` includes several optional rules, to reflect that a transition's triggering event, guard, and effects are all optional attributes: the optional rules look for their values without insisting on their presence. Consider how the `transition` declaration operates on a snippet of the XMI representation of the StateMachine model shown in Figure 6 The first two optional

```
StateMachine(X) |- ownedBehavior{"xmi:type"="uml:StateMachine", name="statechart_22", "xmi:id":X};
Region(R) |- mel.ancestor(ownedBehavior{"xmi:id":SM}, region{"xmi:id":R, name:name}),
             StateMachine(SM);
$^Sibling(X,Y) |- mel.parent(subvertex{"xmi:id":parent}, region{"xmi:id":X}),
                  mel.parent(subvertex{"xmi:id":parent}, region{"xmi:id":Y});
State(X) |- mel.parent(region{"xmi:id":R}, subvertex{"xmi:id":X, name:label, "xmi:type"[":%"]:type}),
            Region(R);
contains(parent, child) |- mel.ancestor(subvertex{"xmi:id":parent}, subvertex{"xmi:id":child}),
                           State(child), State(parent);
transition(Source,Target) |- mel.parent(region{"xmi:id":Region},
                                        transition{"xmi:id":transID, source:Source, target:Target, guard:G}),
                             Region(Region),
                             @mel.parent(transition{"xmi:id":transID}, ownedRule{"xmi:id":G}),
                             @mel.parent(ownedRule{"xmi:id":G}, specification{value:guardVal}),
                             @mel.parent(transition{"xmi:id":transID}, effect{"xmi:id":eid}),
                             @mel.parent(effect{"xmi:id":eid}, body{mel.contents["\n%"]:effect}),
                             @mel.parent(transition { "xmi:id": transID }, trigger { event: eventId }),
                             @packagedElement {"xmi:type"="uml:SignalEvent", "xmi:id":eventId, signal:sigID},
                             @packagedElement {"xmi:type"="uml:Signal","xmi:id":sigID, name:trigger};
```

Figure 7: *mel* program for extracting facts from a UML StateMachine diagram (Rhapsody).

```
Class(X) |- packagedElement{"xmi:type"="uml:Class", name:label, "xmi:id":X};
InheritsFrom(Parent,Child) |- mel.parent(packagedElement{"xmi:id":Child}, generalization{general:Parent});
Attribute(X) |- ownedAttribute{"xmi:type"="uml:Property", "xmi:id":X, name:label, visibility:visiblity,
                               isReadOnly:const, isStatic:static, type:typeID, mel.exists(type), mel.nexists(association)},
                @packagedElement{"xmi:id":typeID, name:type};
Attribute(X) |- ownedAttribute{"xmi:type"="uml:Property", "xmi:id":X, name:label, visibility:visiblity,
                               isReadOnly:const, mel.nexists(type), mel.nexists(association)},
                mel.parent(ownedAttribute{"xmi:id":X}, type{href["#%"]:typeID}),
                packagedElement{"xsi:type"="uml:DataType", "xmi:id":typeID, name:type};
AttributeOf(C,A) |- mel.parent(packagedElement{"xmi:type"="uml:Class", "xmi:id":C}, ownedAttribute{"xmi:id"["_%"]:A});

Operation(X) |- ownedOperation{"xmi:id":X, name:name, visibility:visibility},
                @mel.parent(ownedOperation{"xmi:id":X},
                            ownedParameter{"xmi:type"="uml:Parameter", direction="return", name:returnName, type:TID}),
                @packagedElement{"xmi:type"="uml:Class", name:returnType, "xmi:id":TID};
OperationOf(Class, Op) |- mel.parent(packagedElement{"xmi:id":Class}, ownedOperation{"xmi:id":Op}),
                          Class(Class), Operation(Op);
```

Figure 8: Excerpt from the *mel* program for extracting facts from a UML Class diagram (MagicDraw).

```
^$Association(S,E) |- Class(S), Class(E),
                      mel.parent(packagedElement{"xmi:id"["_%"]:associationID, "xmi:type"="uml:Association", name:label},
                                 ownedEnd{type["_%"]:S, aggregation!="composite"}),
                      mel.parent(packagedElement{"xmi:id"["_%"]:associationID, "xmi:type"="uml:Association", name:label},
                                 ownedEnd{type["_%"]:E, aggregation!="composite"});
```

Figure 9: Excerpt from the *mel* program for extracting facts from a UML Class diagram (UMLDesigner).

```
.exchange(I,O) |- ownedFunctionalExchanges {source["#%"]:O, target["#%"]:I, name:label};
.output(Y,Z) |- mel.parent(ownedFunctions{id:Y}, outputs {id:Z});
.input(Y,Z) |- mel.parent(ownedFunctions{id:Y}, inputs {id:Z});
$^dataFlow(Y,Z) |- output(Y,O), input(Z, I), exchange(I, O) {label:data};
```

Figure 10: Excerpt from the *mel* program for extracting facts from an Arcadia Logical Architecture diagram (Capella).

```
Feature(X) |- and{abstract:isAbstract, mandatory:isMandatory, name:X};
FeatureOf(A,B) |- or{name:A}, mel.parent(or{name:A}, and{name:B});
$^MutuallyExclusive(A,B) |- Feature(A), Feature(B),
                            alt{name:parent}, FeatureOf(parent,A), FeatureOf(parent,B);
$^RequiredTogether(A,B) |- Feature(A), Feature(B),
                           and{name:parent}, FeatureOf(parent,A), FeatureOf(parent,B);
```

Figure 11: Excerpt from the *mel* program for extracting facts from a Feature Model (FeatureIDE).

rules extract the guard condition "`IS_IN(correct)`", using two `mel.parent` functions to relate the XMI tag `transition` to the corresponding XMI tag `specification` that holds the condition's value. The second two optional rules extract the transition's effect "`itsController->GEN(evDisarm)`", using a `mel.parent` function to relate the XML tag `transition` to the XMI tag `effect`, and a `mel.contents` function (with substring extraction) to extract the effect's value from the XML element's contents. The last three optional rules extract the transition's trigger, as described earlier.

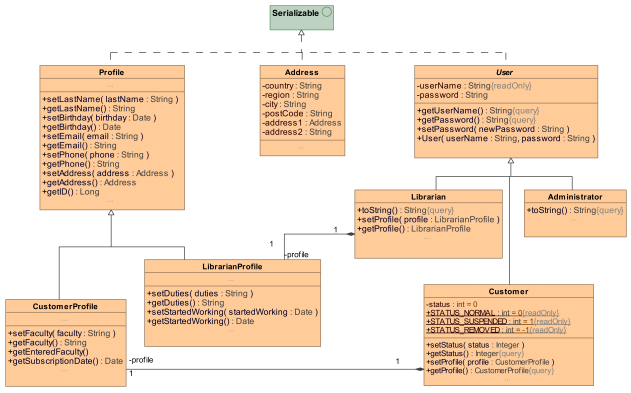

Figure 12: UML Class diagram of a library (MagicDraw).

4.1.2 UML Class Diagrams. This study considered two UML Class Models exported by different tools. The first Model is for a hypothetical Travel Agency system shown in the online appendix. It comes from a sample UML Class diagram project provided with the Sirius-based Modelling tool UMLDesigner version 9.0 [28]. The second Model (Figure 12) is for a hypothetical Library system. It comes from a sample UML Class diagram project provided with MagicDraw version 19.0 [17]. The two Models have different XMI representations. UMLDesigner serializes its Models to the OMG XMI standard version 2.1, and MagicDraw serializes its Models to the OMG XMI standard version 2.5.

Excerpts from our two *mel* programs are presented in Figures 8 and 9. The programs share some commonalities but they do differ because the Models have different XMI formats. Both programs have two definitions of the `Attribute` declaration, because both Models' XMI representations store the type information of a UML class attribute in two formats: either directly as an Attribute of an XML tag or as a nested `type` tag with a reference to the type. Two declarations of the same fact type specify alternative extraction rules; the extractor will collect both and union their results.

Two new attribute requirements were added to *mel* in the course of this study: `mel.exists` and `mel.nexists`. These allow the programmer to, specify that an Attribute in an XML tag must exist or must not exist. The use of these constraints within the `Attribute` declarations stops the rules from producing duplicate results.

4.1.3 Arcadia Logical Architecture Diagrams. The study included an Arcadia diagram, shown in the online appendix, for an in-flight entertainment system. The Model comes from a sample Capella project provided by Thales via the Capella website [11]. Capella's XMI representation adheres to the OMG XMI standard 2.0, but it is quite different from those of the previous Models in that references are more typically via identifiers than to nested information. A likely reason is that a Capella project consists of several Models that all reference a shared pool of (named) entities.

An excerpt from our *mel* program is presented in Figure 10; it shows several intermediate rules that lead up to the declaration of the `dataFlow` relationship. The full *mel* program is considerably smaller than many others in part because the tendency for Capella's XMI to have a "wide" structure with most relationships being resolved by identifier reference rather than nested tree structure. This reduces the need for `ancestor` and `parent` functions.

4.1.4 Feature Models. The study included a Feature Model, shown in the online appendix, of a hypothetical elevator system, part of the "Elevator-FeatureHouse-v1.1" example supplied with FeatureIDE version 3.6.2 [20]. FeatureIDE serializes Models according to its own XML format that does not adhere to any XMI standard, yet we successfully used *mel* to extract the relevant facts from the Model.

The resulting *mel* program (an excerpt of which is presented in Figure 11) is particularly verbose. Four declarations are needed to extract `Feature` entities, and twelve declarations are needed to extract the `FeatureOf` relationship. The reason for so many declarations is that internal `Feature` nodes of any type (`and`, `alt`, `or`) can be combined with child nodes of any type (`and`, `alt`, `or`, `leaf`) and each combination needs to be specified. One of the twelve `FeatureOf` declarations is shown in Figure 11. The `MutuallyExclusive` and `RequiredTogether` declarations illustrate the power of being able to refer to previously defined facts; both declarations avoid enumerating all possible combinations of feature-nodes by referring to existing `ParentOf` facts to find sibling relationships. Despite the verbosity in the number of declarations, each declaration is relatively short, comprising just one or two rules.

4.2 Ease of Using *mel*

The second part 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)[4]. 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 [30]; the comparison criteria are succinctness of the code and the development time (both as proxies for ease of programming). Execution times of the programs was not considered as a criterion because it is in the order of milliseconds on all examples. In each case the outputs of both programs were compared against each other to verify that they extracted the same facts.

4.2.1 Succinctness of mel *code.* 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

[4]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.

Table 3: Program Sizes of Extractors.

Model	mel Lines	Python Lines	XQuery Lines
Arcadia	15	382 (207†)	33
Class Diagram	31	N/A	47
StateChart	26	N/A	39

†The program size without the output formatting code.

Table 4: Time to Create Extractors using *mel* and XQuery.

Task	XML Deduction Time	*mel* Time	XQuery Time
1	26m	46m	75m
2	N/A	53m	43m
3	33m	30m	54m
Totals	N/A	129m	172m

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*.

Both extractors take as input an Arcadia Logical Architecture diagram generated by Capella [11] and output facts about

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

The sizes of the resulting programs are provided in Table 3. The size of the *mel* program is an order of magnitude smaller than the Python program, even when we remove the code responsible for formatting the output. This result is unsurprising: brevity is a natural side effect of most domain-specific languages.

4.2.2 Development time using mel. This study evaluated *mel* with respect to the time it takes to create a fact extractor. The user subject was the third author, who is not involved with the development of *mel* but who is experienced with declarative programming languages. The competing technology used in this study was XQuery [30] because of its facilities for parsing XML and manipulating extracted values and its ease of use.

The user was asked to create *mel* and XQuery fact extractors for three Model representations used in the evaluation study on *mel*'s expressiveness (Section 4.1). For each Model representation, the user was asked to extract either the same facts or a subset of the facts extracted in the expressiveness study:

Task1: Arcadia Logical Architecture diagram (Capella)
- **Entities:** functions, components, data entities
- **Relationships:** data flows between functions

Task2: UML Class diagram (UMLDesigner)
- **Entities:** classes, attributes
- **Relationships:** classes' attributes, associations, compositions
- **Attributes:** attributes' type, visibility, constness, multiplicity

Task3: UML StateMachine diagram (Rhapsody):
- **Entities:** state machines, regions, states
- **Relationships:** state hierarchy, transitions between states
- **Attributes:** transitions' trigger, guard condition, effects

For each Task, the user first analyzed the XML to determine which tags were relevant to the extraction. For Tasks 1 and 2, the user wrote the *mel* program first and the XQuery program second. For Task 3, the user wrote the XQuery program first and the *mel* program second. It is possible that, for any Task, creating and debugging the first program helped the user gain a better understanding of the XML, easing the development of the second program.

The study results are presented in Table 4. The user completed Tasks 1 and 3 much faster when using *mel* than when using XQuery. The user took longer to complete Task 2 when using *mel*, but they did not separate out the time spent analyzing the XML representation; thus, the time to complete the *mel* program includes time spent on analyzing the XML, whereas they may have spent less time analyzing the XML when writing the XQuery program.

5 DISCUSSION

Our evaluation of *mel*'s expressiveness produced both positive and negative results. On the positive side, we used *mel* to extract several fairly complex facts from Models with widely varying XML representations. However, our studies also revealed some deficiencies in *mel*'s language, which have led to minor extensions to *mel*; these extensions are denoted in gray in the *mel* grammar shown in Figure 3. Moreover, the study found that *mel* is very good at relating "wide" elements in the XML (i.e., spread across the XML, with no obvious structural hints) but is less powerful than XQuery for relating "deep" entities (i.e., deeply nested properties and tags). This is discussed in more detail in the Section 5.1.

The user subject in our efficiency studies noted that *mel* lacked error messages. If there was an error in a *mel* program (e.g., rules in a declaration with no matching bindings, or a misspelled name), the program would fail silently and simply return fewer results. Some error handling was subsequently added to `mint`, before the third user task was performed. We note that XQuery also fails silently under many error conditions. The user also found that optional rules were useful for debugging declarations comprising multiple rules; specifically, they would temporarily mark rules as "optional" to identify which rule was causing partial results to be omitted from the final result set, without having to otherwise refactor the rules for debugging.

More studies are needed to better assess whether *mel* improves a user's proficiency at creating fact extractors, but our preliminary results are promising. The user subject completed two Tasks significantly faster using *mel*; unfortunately, their time to complete the third Task was conflated with the time to analyze the Model's XMI.

5.1 Extending *mel* to Generalized XML

Going forward, we plan to extend *mel*'s scope of applicability to more general XML representations. In particular, we will explore extracting facts from XML representations that encode information in the XML tree structure. For example, in Simulink [22] XML, `Line` entities are stored as collections of tags that gather relevant information within a parent tag with no attributes:

```
<Line>
  <P Name="Name">qGust</P>
  <P Name="Src">48#out:2</P>
  <P Name="Points">[35, 0; 0, 40]</P>
</Line>
```

Since the `Line` tag has no attributes, *mel* is unable to relate the child tags with each other. Similarly, *mel* is unable to relate child tags that are nested under parent tags that do not have unique identifiers:

```
<ParentTag name="Foo">
  <ChildTag name="A"/>
</ParentTag>
<ParentTag name="Foo">
  <ChildTag name="B"/>
</ParentTag>
```

If a *mel* rule extracted sibling information from the above XML, it would collect all of the `ChildTags` as siblings, which is undesired.

We propose to extend *mel* with operators that allow users to express purely structural queries about the XML tree, aiming to emulate XPath's document navigation capabilities in a relational context. The operators could be composed in arbitrary sequences to reach elements deep in the XML tree.

5.1.1 Eclipse Modeling Framework Ecore. The Eclipse Modeling Framework (EMF) [31] is a widespread standard in the modelling community. EMF uses an XMI representation called Ecore for its Models. However, rather than referring to a Model element by an identifier, an Ecore representation refers to a Model element according to its position in the XMI tree structure. For example, in an Ecore representation of a Statechart diagram, transitions are encoded as child nodes of their source-state nodes; and a reference to a transition is to a specific child of its source state. For example, a reference to the third transition of a state titled "DoorBehaviour" may look like `DoorBehaviour/@transition.2`. Referring to a child node within an XMI tree structure is similar to the *mel* extensions needed to support structural references in Simulink diagrams, described above — except that to support EMF, *mel* would need features to refer to an arbitrary child of an XMI tree node, based on positional information encoded in the XMI document. We are currently exploring how best to add such features to *mel*.

5.2 Threats to Validity

There are several threats to the validity of our evaluation results. The first is that our studies of *mel*'s scope of applicability were performed entirely by the author of the language. This introduces the threat that the author would choose Model types that they believe *mel* would perform well on. We tried to mitigate against this threat by focusing on Models generated by industrial-strength tools and Models that exercise advanced features of the Modelling language. A secondary threat is that the author could choose to extract only fact types that *mel* could process. We tried to mitigate against this threat by extracting what we believe are all of the Models' entities, relationships, and attributes. However, there may be some niche relationships or attributes in Models that our evaluations do not consider.

The tool author was also involved in the design of the studies that evaluated whether *mel* eases the creation of fact extractors for Models, and they may have selected a biased corpus of Models for those studies. Furthermore, these evaluations were performed by only one user subject who is an experienced programmer in declarative languages, on which *mel* is loosely based.

6 RELATED WORK

There is a large body of literature on technologies for reverse engineering models from source-code artifacts (e.g., LSME [26], Rigi [25], MoDisco [6], JaMoPP [12], Frappé [7], eKNOWS Code Model Service [29], Rex [27]). In contrast, *mel* supports the extraction of user-specified lightweight models (i.e., facts) from XMI Models. The latter is useful when a software component's source code is not available (e.g., libraries, third-party software).

mel sits at the intersection of declarative languages and XML querying/parsing/translation. Li, Liu, Zhu, and Ghafoor present XTQ [21], which is a declarative XML query language whose syntax is heavily influenced by both XQuery and SQL. *mel* and XTQ differ in that XTQ extracts XML snippets the user is querying so that they can look at that specific portion of the XML. Effectively XTQ can extract only excerpts and display them with all the other extracted excerpts; it is not focused on the production of user-defined relationships as *mel* is. XPathLog presented by May [23] is a declarative language whose syntax is influenced by the W3C's XML Path query language XPath. While superficially similar to *mel*, XPathLog focuses on the querying and modification of existing XML, while *mel* focuses on reasoning about XML in a relational context, operating on and outputting sets of tuples (that is, factbases) rather than hierarchically structured XML. XQuery [30] is a mature general-purpose XML query and transformation language that is designed to process XML into either other hierarchical documents or user-formatted reports. While we have shown that XQuery does well in expressing the same queries for which *mel* was designed, it is not intended for logic programming. For this reason, writing logic code in XQuery can be unwieldy as the user must manually handle issues like result multiplicity and sequencing, concepts that remain entirely implicit in a specialized logic language like *mel*.

Some work has been done on transforming XML representations of Models. Bhaati and Malik [2] propose a tool for bi-directional transformation between the XML for one Model and the XML for another Model. The proposed bi-directional framework is preliminary, applied to only small examples, and limited to cases where the tool is provided with metamodels for both XML representations.

Graaumans [8] has studied the usability of three XML query languages SQL/XML, XQuery and XSLT. *mel* effectively functions as a domain-specific XML query language, and we are interested in its usability. Graaumans' work goes beyond the evaluations conducted for this paper, in that it conducts a full user study; it provides a good template for future user studies involving *mel*.

7 CONCLUSIONS

This paper presents *mel* and its interpreter `mint` for rapid development of fact extractors for Models whose textual representation adheres to the OMG XMI standard. *mel* has been used to extract facts from UML Class and StateMachine diagrams, Arcadia Logical Architecture diagrams, and Feature Models. Evaluations show promising results with respect to the applicability of *mel* to Models represented in XMI; and to *mel*'s ability to ease the development of fact extractors. Future plans are to extend *mel* to Models with more general XML textual representations and Models with Ecore representation, and to evaluate *mel* on a larger collection of Models.

REFERENCES

[1] Luciane T. W. Agner and Timothy C. Lethbridge. 2017. A Survey of Tool Use in Modeling Education. In *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems* (Austin, Texas) *(MODELS '17)*. IEEE Press, 303–311. https://doi.org/10.1109/MODELS.2017.1

[2] Shahid Nazir Bhatti and Asif Muhammad Malik. 2009. An XML-Based Framework for Bidirectional Transformation in Model-Driven Architecture (MDA). *SIGSOFT Softw. Eng. Notes* 34, 3 (May 2009), 1–5. https://doi.org/10.1145/1527202.1527206

[3] I. T. Bowman, R. C. Holt, and N. V. Brewster. 1999. Linux as a case study: its extracted software architecture. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 555–563.

[4] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. *SIGPLAN Notices* 44, 10 (October 2009), 243–262.

[5] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau eds. 2008. *Extensible Markup Language (XML) 1.0 (Fifth Edition) W3C Recommendation*. Technical Report. Retrieved May 25th, 2020 from https://www.w3.org/TR/2008/REC-xml-20081126/

[6] Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. MoDisco: A model driven reverse engineering framework. *Information and software technology* 56, 8 (2014), 1012–1032.

[7] O. Goonetilleke, D. Meibusch, and B. Barham. 2017. Graph Data Management of Evolving Dependency Graphs for Multi-versioned Codebases. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 574–583.

[8] Joris Graaumans. 2004. A Qualitative Study to the Usability of Three XML Query Languages. In *Proceedings of the Conference on Dutch Directions in HCI* (Amsterdam, The Netherlands) *(Dutch HCI '04)*. Association for Computing Machinery, New York, NY, USA, 6. https://doi.org/10.1145/1005220.1005228

[9] Object Management Group. 2015. *XML Metadata Interchange (XMI) Specification, Version 2.5.1*. Technical Report formal/2015-06-07. Object Management Group. Retrieved May 25th, 2020 from http://www.omg.org/spec/XMI/2.5.1

[10] Object Management Group. 2020. About the Unified Modeling Language Specification Version 2.5.1. Retrieved May 25th, 2020 from https://www.omg.org/spec/UML/About-UML/

[11] Thales Group. 2020. Model Based Software Engineering | Capella MBSE Tool. Retrieved May 25th, 2020 from https://www.eclipse.org/capella/

[12] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. 2009. Closing the Gap between Modelling and Java. In *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers (Lecture Notes in Computer Science)*, Mark van den Brand, Dragan Gasevic, and Jeff Gray (Eds.), Vol. 5969. Springer, 374–383. https://doi.org/10.1007/978-3-642-12107-4_25

[13] Hevner, March, Park, and Ram. 2004. Design Science in Information Systems Research. *MIS quarterly* 28, 1 (2004), 75–105.

[14] R. C. Holt. 1998. Structural Manipulations of Software Architecture using Tarski Relational Algebra. In *Proceedings Fifth Working Conference on Reverse Engineering (WCRE'98)*. 210–219.

[15] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 471–480. https://doi.org/10.1145/1985793.1985858

[16] IBM. 2020. IBM Engineering Systems Design Rhapsody. Retrieved May 25th, 2020 from https://www.ibm.com/ca-en/marketplace/systems-design-rhapsody

[17] No Magic Inc. 2000. MagicDraw. Retrieved May 25th, 2020 from https://www.nomagic.com/products/magicdraw

[18] Eirini Kalliamvakou, Marc Palyart, Gail C. Murphy, and Daniela Damian. 2015. A Field Study of Modellers at Work. In *Proceedings of the Seventh International Workshop on Modeling in Software Engineering* (Florence, Italy) *(MiSE '15)*. IEEE Press, 25–29.

[19] Amal Khalil and Juergen Dingel. 2015. Incremental Symbolic Execution of Evolving State Machines. In *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems* (Ottawa, Ontario, Canada) *(MODELS '15)*. IEEE Press, 14–23.

[20] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 611–614. https://doi.org/10.1109/ICSE.2009.5070568

[21] Xuhui Li, Mengchi Liu, Shanfeng Zhu, and Arif Ghafoor. 20140604. XTQ: A Declarative Functional XML Query Language. (20140604).

[22] MathWorks. 1194. Simulink - Simulation and Model-Based Design. Retrieved May 25th, 2020 from https://www.mathworks.com/products/simulink.html

[23] W. May. 2001. XPathLog: a declarative, native XML data manipulation language. In *Proceedings 2001 International Database Engineering and Applications Symposium*. 123–128.

[24] H. A. Müller and K. Klashinsky. 1988. Rigi-A System for Programming-in-the-Large. In *Proceedings of the 10th International Conference on Software Engineering*. 80–86.

[25] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. 1993. Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*. IBM Press, 217–226.

[26] Gail C. Murphy and David Notkin. 1996. Lightweight Lexical Source Model Extraction. *ACM Trans. Softw. Eng. Methodol.* 5, 3 (July 1996), 262–292.

[27] B. J. Muscedere, R. Hackman, D. Anbarnam, J. M. Atlee, I. J. Davis, and M. W. Godfrey. 2019. Detecting Feature-Interaction Symptoms in Automotive Software using Lightweight Analysis. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 175–185.

[28] Obeo. 2020. UML Designer Documentation. Retrieved May 25th, 2020 from http://www.umldesigner.org/

[29] Rudolf Ramler, Georg Buchgeher, Claus Klammer, Michael Pfeiffer, Christian Salomon, Hannes Thaller, and Lukas Linsbauer. 2019. Benefits and Drawbacks of Representing and Analyzing Source Code and Software Engineering Artifacts with Graph Databases. In *Software Quality: The Complexity and Challenges of Software Engineering and Software Quality in the Cloud*, Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann (Eds.). 125–148.

[30] Jonathan Robie, Michael Dyck, and Josh Spiegel. 2017. *XQuery 3.1: An XML Query Language*. Technical Report. Retrieved May 25th, 2020 from https://www.w3.org/TR/xquery-31/

[31] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. [n.d.]. *EMF : Eclipse Modeling Framework* (2nd ed. ed.). Addison Wesley.

[32] Jean-Luc Voirin. 2017. *Model-based System and Architecture Engineering with the Arcadia Method* (1 ed.). Elsevier.