Towards Partial Loading of XMI Models

Ran Wei
University of York
Heslington
York, United Kingdom
ran.wei@york.ac.uk

Dimitrios S. Kolovos
University of York
Heslington
York, United Kingdom
dimitris.kolovos@york.ac.uk

Antonio Garcia-Dominguez
University of York
Heslington
York, United Kingdom
antonio.garcia-dominguez@york.ac.uk

Konstantinos Barmpis
University of York
Heslington
York, United Kingdom
kb634@york.ac.uk

Richard F. Paige
University of York
Heslington
York, United Kingdom
richard.paige@york.ac.uk

ABSTRACT

XML Metadata Interchange (XMI) is an OMG-standardised model exchange format, which is natively supported by the Eclipse Modeling Framework (EMF) and the majority of the modelling and model management languages and tools. Whilst XMI is widely supported, the XMI parser provided by EMF is inefficient in some cases where models are read-only (such as input models for model query, model-to-model transformation, etc) as it always requires loading the entire model into memory. In this paper we present a novel algorithm, and a prototype implementation (SmartSAX), which is capable of partially loading models persisted in XMI. SmartSAX offers improved performance, in terms of loading time and memory footprint, over the default EMF XMI parser. We describe the algorithm in detail, and present benchmarking results that demonstrate the substantial improvements of the prototype implementation over the XMI parser provided by EMF.

CCS Concepts

•Software and its engineering → Software development methods;

Keywords

EMF; XMI; Partial Model Loading;

1. INTRODUCTION

Model Driven Engineering (MDE) is a contemporary software development paradigm in which *models* are first class artefacts. On *models*, a series of *model management operations* can be performed to automatically produce software artefacts such as software source code and documentation. An MDE-based development processes typically involves modelling and model management, supported by a large number of modelling and model management tools.

Tool interoperability among MDE tools is achieved by the use of common persistence format(s) for models. XML Metadata Interchange (XMI) is an XML-based model persistence format standardised by the Object Management Group (OMG)[1]. XMI is supported natively by the widely used Eclipse Modeling Framework (EMF) [1], and as an import/export format by the majority of software modelling tools, including Enterprise Architect, Modelio, MagicDraw, Microsoft Visio, Visual Paradigm, and Archi[2]. XMI is also supported by all major model management (e.g. model querying, model-to-model transformation, model validation) languages and tools through their implementation based on EMF, including OCL [2], Acceleo [3], Xpand [4], ATL [5] and Epsilon [6].

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

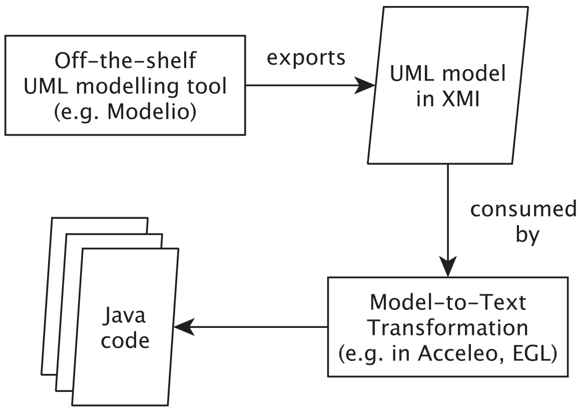

Off-the-shelf UML modelling tool (e.g. Modelio)
exports
UML model in XMI
consumed by
Model-to-Text Transformation (e.g. in Acceleo, EGL)
Java code

Figure 1: A common use-case that would benefit from partial XMI loading capabilities

[1]http://www.omg.org/
[2]http://www.archimatetool.com

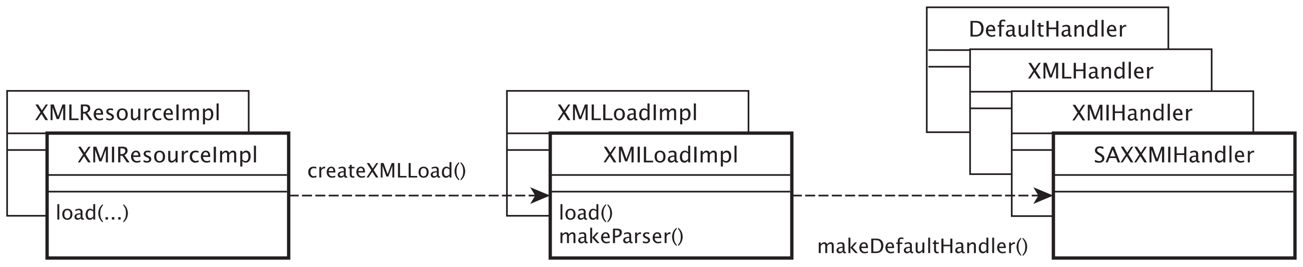


Figure 2: EMF SAX Parser Structure.

While some model management programs (e.g. model-to-model, model-to-text transformations) only need to access a subset of the elements in a model, existing XMI parsers do not support partial loading and as such they need to parse the entire model into memory before any model elements can be accessed – which is clearly wasteful both in terms of loading time and memory footprint. As a motivating example, consider the scenario illustrated in Figure 1, where a developer is using an off-the-shelf UML modelling tool (e.g. Modelio) to construct UML models. Models are then exported into XMI so that they can be consumed by an XMI-compatible model-to-text transformation (e.g. written in Acceleo [3] or EGL [7]) which generates Java source code from the classes of the model and their structural features and associations.

While the transformation in question is only interested in a subset of its input model and is oblivious to parts such as sequence diagrams, state machines, use cases etc. in the absence of partial loading capabilities, all of the latter will need to be unnecessarily loaded into memory from the XMI-based model representation before the transformation can be executed. While the cost of fully loading a small XMI model is practically negligible, this becomes problematic when models grow in size as observed in a number of studies [8–11].

It is worth noting that in this (frequently encountered) case, a database-backed model persistence framework like CDO[3] would not be of much assistance. In fact, injecting the exported XMI model into a CDO repository would first require the XMI model to be fully-loaded, which would defeat the purpose of introducing this facility in the first place.

To address this limitation, this work contributes a novel parsing algorithm that can be used to partially load an XMI-based EMF model into memory, given in-advance knowledge of the part of the model that will be exercised by the model management program that consumes it (such knowledge can be obtained through static analysis of the program itself). This novel XMI parsing algorithm is presented in the form of a prototype, SmartSAX, built atop the Eclipse Modelling Framework (EMF). The rest of the paper is organised as follows.

Section 2 reviews the EMF's default XMI parser and explains in detail the parsing algorithm. Section 3 discusses the proposed partial loading algorithm with a detailed example. Section 4 discusses the related work in this line of research, and Section 5 reports on the results of empirical evaluations which demonstrate that the proposed partial XMI loading algorithm delivers significant benefits both in terms of loading and memory footprint. Section 6 concludes the paper and provides directions for future work.

[3]http://www.eclipse.org/cdo/

2. BACKGROUND

2.1 XMI and MDE tools

Model Driven Engineering (MDE) enables software developers to operate at the *model* level, for capturing relevant details of a system under development, and facilitates *model management*, which comprises operations performed on models to reason about the system and automatically generate software artefacts. MDE has been shown to increase productivity by as much as a factor of 10 from empirical studies [12, 13] and thus has received a considerable amount of attention in recent years resulting a large number of modelling and model management tools being developed.

The Eclipse Modeling Framework (EMF) is a modelling framework that enables MDE by providing support for modelling and code generation. To enable interoperability of models, EMF supports serialising models in the XMI format. Based on EMF, a large number of different MDE tools have been developed for different model management operations, such as OCL for model validation [14, 15], XText and MoDisco for text-to-model transformation [16, 17], ATL and ETL for model-to-model transformation [18, 19], EGL and Acceleo for model-to-text transformation [7, 20], model merging [21], etc. EMF has become a *de facto* standard for building MDE tools [22] provided that the majority of model management tools in MDE are implemented atop EMF. As such, the XMI format is inherently supported by the majority of model management tools.

2.2 Java SAX Parser

The Java SAX (Simple API for XML) parser[4] is an event-based XML parser that operates by going through an XML file/stream and invoking callback methods on a *listener/handler* object (which subclasses SAX's internal *DefaultHandler* class) when it encounters certain structural elements of the XML file. For example, the parser invokes the handler's *startDocument()* method when the start of an XML document is encountered, its *startElement()* method when the start of an element is encountered, etc. It is the responsibility of the handler to extract the information it needs from these elements (by extending the callbacks methods where appropriate) and perform corresponding actions (e.g. to create *EObject*s for EMF's SAX parser).

[4]https://docs.oracle.com/javase/tutorial/jaxp/sax/

2.3 EMF's SAX-based XMI Parser

Figure 2 illustrates the main components of EMF's built-in XMI SAX parser. To parse an XMI-based model an *XMIResourceImpl* object needs to be created, which points at the URI (location) of the (XMI) model. To load the contents of the model, the *XMIResourceImpl* object creates an *XMILoadImpl* object which is in turn responsible for creating a *SAXXMIHandler*. The *SAXXMIHandler* is responsible for handling XML events and for creating model elements (*EObjects*), populating their attributes and references (*EStructuralFeatures*), and putting them in the *XMIResourceImpl*.

2.4 Default XMI Parsing Algorithm

To illustrate how EMF's existing SAX-based XMI parser works, we introduce a contrived running example involving the *University* metamodel shown in Figure 4, and a sample model that conforms to it. The XMI representation of our sample model appears on the left part of Figure 3. This model contains a *University* element, which in turn contains a *Department* (*Computer Science*) element. Under *Computer Science*, there are two members: a *Lecturer* (Tom Brown) and a *Student* (Cathy Smith). Tom has a web page while Cathy does not. Under *Computer Science*, there is also a *Module* element: *MODE*, which also has a web page. Finally, both Tom and Cathy are involved in MODE (see *modules="e6"* in lines 6 and 10).

We will now describe how EMF's XMI parser works with reference to the XMI model in Figure 3. The parser (and in particular its *SAXXMIHandler* component) maintains a stack of model elements (*EObject*s in EMF's terminology) to keep track of its current position in the XMI document. This is needed in order to determine what *EObject*s to create next, as illustrated in the right part of Figure 3. When line 1 of the XMI file is read, the callback method *startElement()* is triggered, the <*university*> element is handled and a new instance of *University* (with its *name* attribute set to UoY) is created. The new *EObject* is pushed into the object stack. When line 4 is read, the parser processes the top of the stack (*peekObject* in EMF's terminology) together with the <*departments*> element and decides that an instance of *Department* should be created and added to the *departments* reference of the *University* model element. The created instance of *Department* is also pushed into the object stack. The same principle is applied when line 5 is read: the element <*members*> is handled and an instance of *Staff* is created, added to the *members* reference of the *Department* and pushed into the stack. When an element tag ends (e.g. in line 8), the top element of the object stack is popped. Once all XML elements have been processed, a tree structure has been constructed in memory and resolution of non-containment references (e.g. *Member.modules*) takes place to transform the tree into the graph of Figure 5.

3. PARTIAL XMI LOADING

While parsing the entire contents of an XMI-based model into an in-memory object graph is often necessary (e.g. when it is not known in advance which elements of the model will need to be accessed by a program/user, or when there is a need to modify the model and persist changes), there are also cases where only parts of the model need to be loaded, and precise information about which parts are relevant can be provided in advance. For example, when a model is loaded in order to be queried by a program (e.g. a set of OCL constraints or an M2M/M2T transformation), it is possible to detect through static analysis which parts of the model the program is likely to exercise. In such cases, loading parts of the model that the program is guaranteed not to access is inefficient both in terms of loading time and in terms of memory footprint. To improve support for working with XMI-based models in such scenarios, in this section we demonstrate an algorithm that is used to load only *EObject*s of interest into memory and ignore other remaining XML elements.

3.1 Effective Metamodel Structure

To achieve partial loading, our XMI parser needs to be provided, in advance, with information about the parts of the model that will be subsequently needed by programs that consume the model (i.e. the model's *effective metamodel*, similar to the concept described in [23]). Figure 6 illustrates how effective metamodels are represented in our prototype implementation. For clarification, assume a program P manages a model M which conforms to its metamodel MM. The base construct is *EffectiveType*, which represents a meta-class in MM. *EffectiveType* contains a *name*, a collection of *attributes* and a collection of *references* of interest. The effective metamodel of M is represented by *EffectiveMetamodel*, which has a *name*, an *nsURI* (MM's globally unique identifier), and three collections of *EffectiveType* (*types*, *allOfKind* and *allOfType*).

3.1.1 types, allOfKind and allOfType

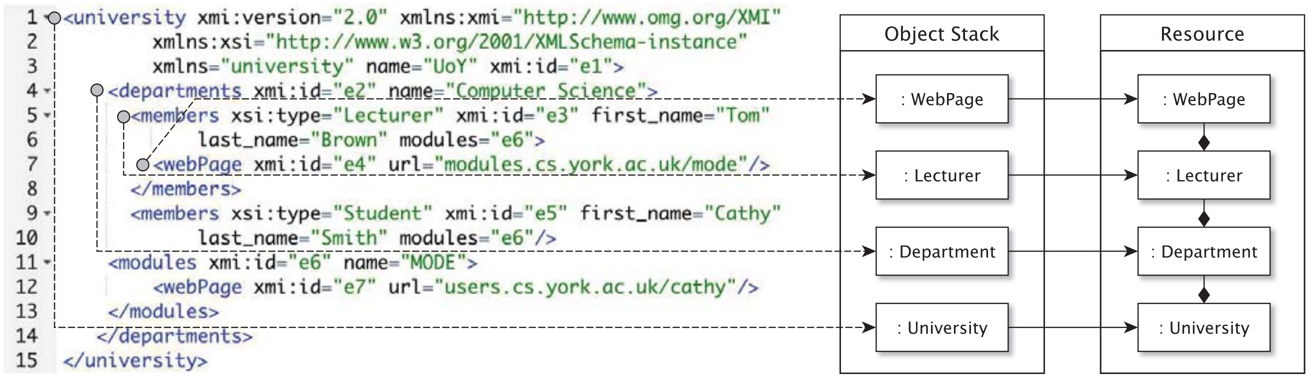
The *allOfKind* and *allOfType* references are used to specify the types in MM, instances of which need to be loaded by the parser. As an example, if we declare in our effective metamodel that we need *allOfKind* of *Person*, this implies that instances of both *Lecturer* and *Student* should be loaded (as they are sub-types of *Person*). If we declare that we need *allOfType* of *Person*, only instances of *Person* should be loaded (but not of its subtypes). The *types* reference specifies types, instances of which should be loaded only when they appear under containment references of interest. For example, in the effective metamodel of Figure 7, we specify that we need to load all instances of *Lecturer* anywhere in the model, but only instances of *WebPage* that are contained in containment references of interest (i.e. *Lecturer.webPage*).

3.1.2 Attributes and References

In each *EffectiveType* we can declare the names of the attributes and references that need to be populated for loaded instances of that type. For example, we can declare that we are only interested in the *first_name* attribute and *modules* reference of *Person* model elements, which will cause the partial loading parser not to populate the value of the *last_name* attributes of loaded *Person* elements.

3.2 Automated Effective Metamodel Extraction

Extracting an effective metamodel from a query/transformation expressed in a typed model management language (such as OCL, QVTr/o, Acceleo or Epsilon's model management languages) for which static type-inference facilities are in place is a relatively straightforward and computationally inexpensive process. For example, analysing the program



```
<university xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="university" name="UoY" xmi:id="e1">
   <departments xmi:id="e2" name="Computer Science">
     <members xsi:type="Lecturer" xmi:id="e3" first_name="Tom"
           last_name="Brown" modules="e6">
       <webPage xmi:id="e4" url="modules.cs.york.ac.uk/mode"/>
     </members>
     <members xsi:type="Student" xmi:id="e5" first_name="Cathy"
           last_name="Smith" modules="e6"/>
    <modules xmi:id="e6" name="MODE">
       <webPage xmi:id="e7" url="users.cs.york.ac.uk/cathy"/>
    </modules>
   </departments>
</university>
```

Figure 3: Parsing a University XMI model using EMF's built-in XMI parser.

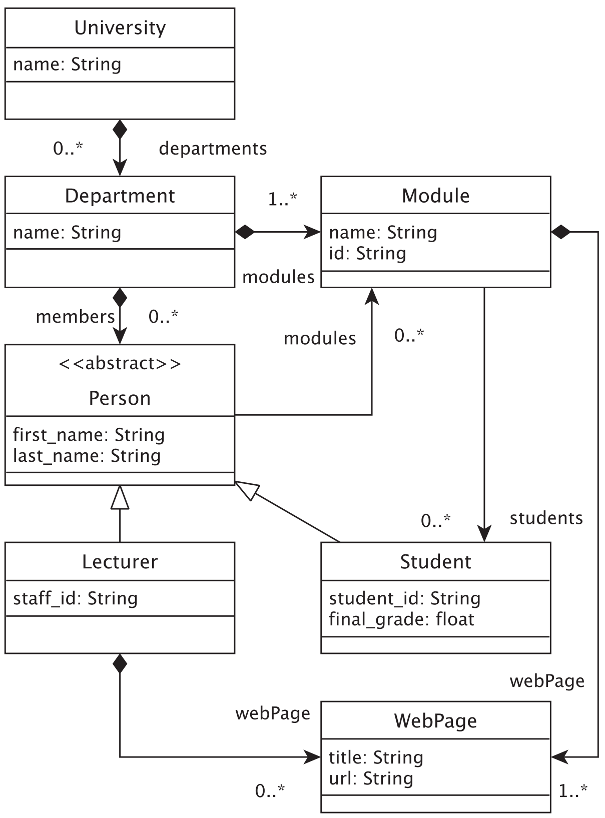


Figure 4: The University Metamodel

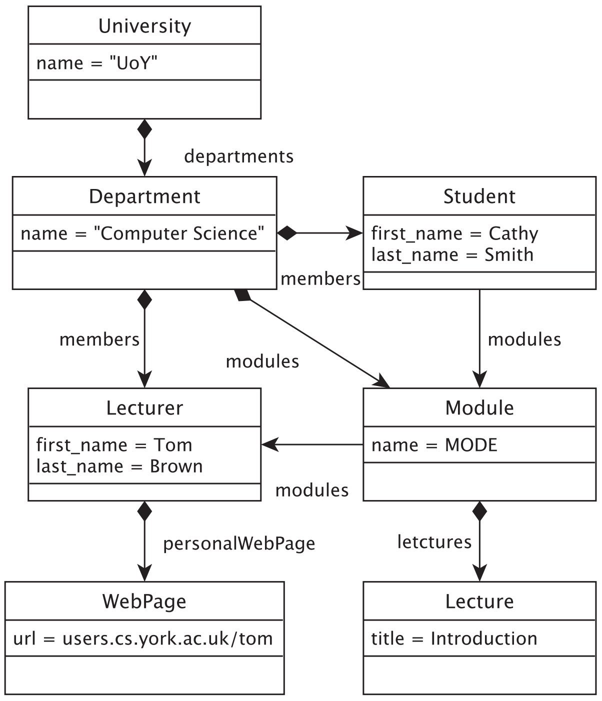


Figure 5: Graphical representation of the loaded XMI model of Figure 3

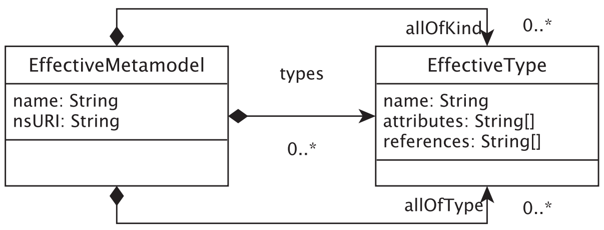


Figure 6: Effective Metamodel Representation

written in Epsilon Object Language (EOL) [24] in Listing 1 using Epsilon's static analyser [25] produces the effective metamodel shown in Figure 7.

```
var lecturers = Lecturer.allOfKind();
for (l in lecturers) {
  ("First name:" + l.first_name).println();
  ("Last name:" + l.last_name).println();
  ("Web page:" + l.webPage.url).println();
  ("Number of modules taught:" + l.modules.size
      ()).println();
}
```

Listing 1: An example EOL Program

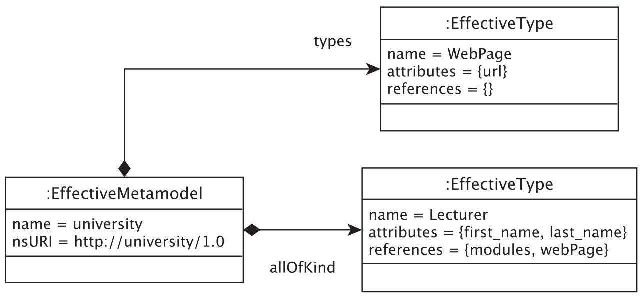


Figure 7: Automatically-extracted Effective Metamodel from Listing 1

3.3 Effective Metamodel Reconciliation

Effective metamodels (either specified manually or extracted through static analysis of model management programs) can be incomplete. For example, the effective metamodel illustrated in Figure 7 specifies that we are interested in loading only instances of *Lecturer*, and populating their *first_name* and *last_name* attributes and their *webPage* and *modules* references. It also specifies that for loaded instances of *WebPage*, their *url* attribute should be populated.

As the effective metamodel does not include the *Module* type, the parser will not load any instances of *Module*, and as such the *modules* reference of all loaded *Lecturer* elements will be empty. To handle such cases, in this step we automatically reconcile a provided (potentially incomplete) effective metamodel by adding *allOfKind* relationships to the types of declared non-containment references. If an *allOfType* relationship already exists for that type, it is converted to an *allOfKind* relationship. The reconciled effective metamodel for our example, where the missing *allOfKind* relationship has been added for the *Module* type, appears in Figure 8.

3.4 Partial XMI Loading Algorithm

As discussed above, EMF's built-in XMI parser maintains a stack of non-null *EObjects* to determine what type of *EObject* it needs to create when it encounters a new XML element, and where[5] it should place the new element in the containment hierarchy. Whilst we wish to reuse as much of the (by far non-trivial) functionality of the existing XMI parser as possible, creating all *EObject*s in order to keep

[5]i.e. under which containment reference

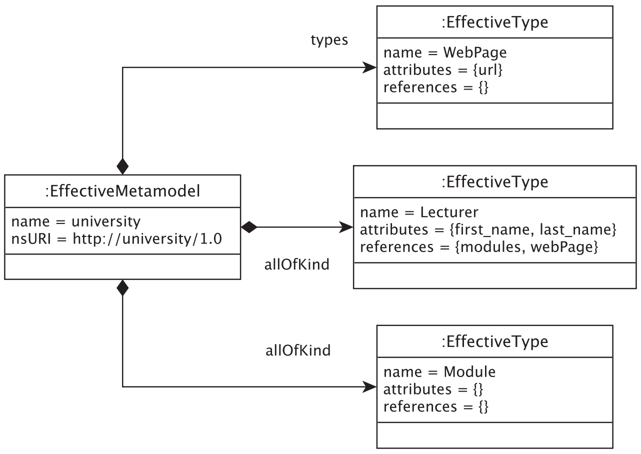


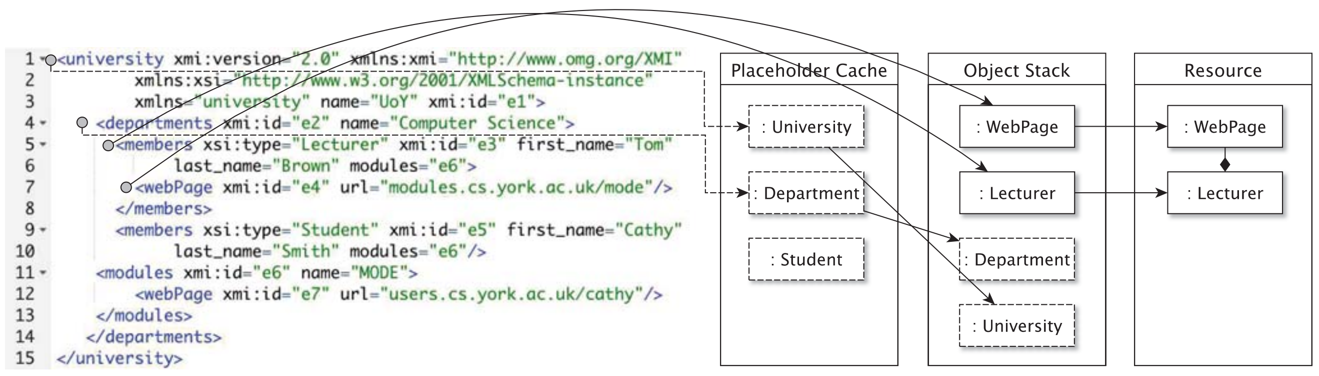
Figure 8: Reconciled Version of the Effective Metamodel of Figure 7

the stack null-free is not efficient in terms of time and memory consumption, and defeats the purpose of partial loading. Therefore, we have introduced a *placeholder cache*, which contains one empty/placeholder *EObject* for each type that is not declared in the effective metamodel. As such, when the parser encounters an XML element that it wishes to *skip* it can fetch the respective placeholder *EObject* for that type from the cache and put it in the object stack (note that this process does not involve processing the actual XML element, therefore reducing loading time and memory consumption).

On the other hand, when it encounters an XML element of a type that is included in the effective metamodel under an appropriate *allOfType* or *allOfKind* reference, or an element that belongs to a containment reference of interest and is included in the effective metamodel under a *types* reference, it creates a new *EObject*. If the top element of the stack is not an placeholder *EObject* and the containment reference is included in the effective metamodel, it puts the new object under the containment reference; otherwise it adds it as a top level element in the resource (model).[6] Finally, the parser adds the new *EObject* to the top of the stack.

Figure 9 illustrates a snapshot of the state of our parser at the point where it has parsed the University XMI model up to line 7, with reference to the reconciled effective metamodel of Figure 8. When parsing starts, our parser populates the *Placeholder Cache* with one placeholder *EObject* for each type of the full metamodel that is not included in the effective metamodel as illustrated on the bottom-left corner of Figure 9. Another approach would be to populate the cache in a lazy manner - i.e. to only create placeholder *EObjects* the first time they are needed. Given the relatively small size of metamodels – e.g. the UML metamodel has fewer than 300 types – the performance benefits of a lazy cache population strategy are negligible. Having populated the *Placeholder Cache*, the parser then handles the XML elements it encounters as follows:

[6]The rearranged order of the *EObject*s in the resource does not affect how existing model management tools interact with the models – tools that interact with EMF based XMI models do not have to make any changes to their implementation.



```
 1 <university xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
 2         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 3         xmlns="university" name="UoY" xmi:id="e1">
 4   <departments xmi:id="e2" name="Computer Science">
 5     <members xsi:type="Lecturer" xmi:id="e3" first_name="Tom"
 6           last_name="Brown" modules="e6">
 7       <webPage xmi:id="e4" url="modules.cs.york.ac.uk/mode"/>
 8     </members>
 9     <members xsi:type="Student" xmi:id="e5" first_name="Cathy"
10           last_name="Smith" modules="e6"/>
11   <modules xmi:id="e6" name="MODE">
12       <webPage xmi:id="e7" url="users.cs.york.ac.uk/cathy"/>
13   </modules>
14   </departments>
15 </university>
```

Figure 9: Parsing the University model with SmartSAX.

- When the <*university*> element in line 1 is encountered, the parser checks the effective metamodel, determines that instances of *University* do not need to be loaded, and therefore fetches the placeholder instance of *University* from the placeholder cache and pushes it to the object stack.

- When the <*departments*> element in line 4 is encountered, the parser determines that the type of the object to be instantiated is *Department*. However, according to the effective metamodel, instances of *Department* do not need to be loaded either so the parser fetches the placeholder instance of *Department* from the placeholder cache and pushes it to the object stack.

- When the parser encounters the <*members*> element in line 5 it determines that it needs to create a new instance of *Lecturer* (as *Lecturer* is part of the effective metamodel). After creating the new instance, it consults the effective metamodel and populates the values of its *first_name* and *last_name* attributes. Then it looks at the element on the top of the stack (currently the placeholder instance of *Department*), detects that it is a placeholder, and as such adds the populated instance of *Lecturer* to the resource as a top-level element.

- When the <*webPage*> element in line 7 is encountered, the parser determines that it needs to create an instance of *WebPage* and place it in the *webPage* containment reference of the top element of the stack. It also populates the *url* attribute of the new instance with the value of the respective attribute of the XML element.

- When </*webPage*> is encountered in line 7, the top object of the stack is popped (the current top element is now *Tom*)

- When </*members*> is encountered in line 8, the top object of the stack is popped (the current top element is now *Computer Science*)

- When the <*members*> element is encountered in line 9, the parser determines that it needs to create an instance of *Student*. Since the *Student* type is not part of the effective metamodel, it fetches the *Student* placeholder object and puts it at the top of the stack.

- When </*members*> is encountered in line 10, the top object of the stack is popped (the current top element is again *Computer Science*)

- When the <*modules*> element is encountered in line 11, the parser determines that it needs to create an instance of *Module* and since the effective metamodel declares that all instances of *Module* need to be loaded, it creates a fresh object (but does not populate any of its attributes/references as none of these need to be loaded according to the effective metamodel). Since the top element in the stack is a placeholder, it adds the new *Module* instance to the resource as a top-level element and also pushes it to the stack.

- When the <*webPage*> element is encountered in line 12, the parser determines that it maps to an instance of *WebPage* that should be placed under the *webPage* containment reference of the top element of the stack (which is currently the *MODE* module). Since the *WebPage* type is part of the *types* reference of the effective metamodel, and its containment reference (*Student.webPage*) is not of interest, the parser fetches the placeholder *WebPage* object and pushes it to the stack.

- Each of the last three lines (13-15) cause the parser to pop the top element of its stack – thus ending up with an empty stack.

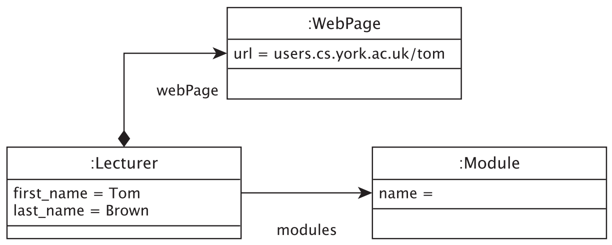


Figure 10: The Partially Loaded Model

Since all required objects have been loaded, the last step of the algorithm involves resolving non-containment references (in this case, link *Tom* to the *MODE* module, through its *modules* reference). The obtained partially-loaded model is shown in Figure 10. Note how the *name* attribute of the loaded *Module* is empty, as the value of this attribute is not of interest according to the effective metamodel of Figure 8.

The partial loading algorithm illustrated above is also presented in an example-independent manner in Algorithm 1, 2 and 3.

```
let stack = new stack of model elements;
let cache = new set of model elements;
let model = new model;
let elements = new stack of xml elements;
let EM = the defined/extracted effective metamodel;
let referencesToHandle = non-containment references
to resolve after file is fully read;
Procedure startElement(xmlElement)
    push xmlElement to elements;
    let peekModelElement = peek top model element of
    stack;
    if peekModelElement is nil then
        // We are at a root element
        let type = find a model element type for the tag
        name of the xmlElement;
        let modelElement = createModelElement(type);
        push modelElement to the stack;
    end
    else
        let peekModelElementType = the type of
        peekModelElement;
        if a feature needs to be created based on
        peekModelElementType and xmlElement then
            handleFeature(xmlElement);
        end
        else if a (top level) model element to be created
        then
            let type = find a model element type for the
            tag name of the xmlElement;
            let modelElement =
            createModelElement(type);
            push modelElement to the stack;
        end
    end
Procedure endElement(element)
    pop the top model element from the stack;
    pop the top model element from the elements;
```
Algorithm 1: Partial Loading Algorithm 1 of 3

3.5 Integration to EMF

With the partial loading algorithm in mind, we implement our prototype, SmartSAX[7], which extends EMF. The structure of SmartSAX is shown in Figure 11. The partial loading algorithm is encompassed in *SmartSAXXMIHandler*, which extends EMF's SAXXMIHandler. For this purposes, extensions of *XMILoadImpl* and *XMIResourceImpl* are created, which require passing of effective metamodels in their *load(...)* options. The partially loaded in-memory models (Resources) can still be accessed in the same way by model management tools as for default loaded EMF models.

3.6 Limitations

There are three noteworthy limitations in the presented partial loading approach. First, it requires elements referenced from non-containment references to have IDs that

[7]https://github.com/wrwei/SmartSAX

```
let stack = new stack of model elements;
let cache = new set of model elements;
let model = new model;
let elements = new stack of xml elements;
let EM = the defined/extracted effective metamodel;
let referencesToHandle = non-containment references
to resolve after file is fully read;
Procedure createModelElement(type)
    let modelElement = instance to be
    created/retrieved;
    if EM contains type under allOfKind/allOfType
    then
        modelElement = create an instance of the type;
        add modelElement to the resource;
    end
    else
        modelElement = create/retrieve cache object
        from the cache by type;
    end
    return modelElement;
Procedure handleObjectAttributes(eObject)
    foreach attribute in the current xmlElement do
        let name = name of the attribute;
        let value = the value of the attribute;
        if shouldHandleFeature(eObject, attribute)
        then
            setFeatureValue(eObject, name, value);
        end
    end
Procedure setFeatureValue(eObject, name, value)
    let feature = identify feature based on eObject and
    name;
    if feature is single valued then
        set value to feature;
    end
    else
        add value to feature;
    end
```
Algorithm 2: Partial Loading Algorithm 2 of 3

do not depend on their positions in the containment hierarchy (i.e. *intrinsic IDs* or *extrinsic IDs* instead of *fragment path* IDs [1] such as *//@departments.0/@modules.0*) as partial loading can affect the internal structure of the loaded model. Also, it currently does not support propagating changes made to the partially-loaded model back to its XMI source (i.e. it is only useful for read-only operations on models). Finally, SmartSAX currently does not support loading models that are persisted in multiple XMI files, which will be addressed in the future work.

4. RELATED WORK

To the best of our knowledge, there is no previous work on partial loading of XMI-based models. However, to address the limitations of XMI when working with large models, several (non-standard) alternatives have been proposed. In [26], the Binary Model Syntax (BMS) is introduced as a high performance binary alternative to XMI. However, there have not been any updates or releases of BMS in the public domain. A number of database-backed persistence technologies have also been proposed to achieve partial loading. The Connected Data Objects framework (CDO)[8] is a framework

[8]http://www.eclipse.org/cdo/

```
Procedure handleFeature(xmlElement)
    let peekModelElement = peek top model element of stack;
    let peekModelElementType = the type of peekModelElement;
    let feature = the feature that is to be created based on peekModelElementType and xmlElement;
    if feature is an attribute then
        handleObjectAttributes(peekModelElement);
    end
    else
        //feature is a reference
        if feature is a containment reference then
            let eType = the type of the reference;
            let modelElement = createModelElement(eType);
            if modelElement is null then
                if em contains eType under types then
                    if shouldHandleFeature(peekModelElement, feature) then
                        let modelElement = create an instance of the eType;
                        add modelElement to the resource;
                        setFeatureValue(peekModelElement, feature name, modelElement);
                    end
                end
                else
                    let modelElement = create/retrieve cache object from the cache for typeToCreate;
                end
            end
            else
                add modelElement to the resource;
                setFeatureValue(peekModelElement, feature name, modelElement);
            end
            push modelElement to the stack;
        end
        else
            if shouldHandleFeature(peekModelElement, feature) then
                add feature to referencesToHandle;
            end
        end
    end
Procedure shouldHandleFeature(eObject, feature)
    let effectiveType = retrieve effective type from EM based on eObject;
    if effectiveType is not nil then
        if effectiveType contains the name of feature then
            return true;
        end
    end
    return false;
```

Algorithm 3: Partial Loading Algorithm 3 of 3

built on top of EMF and supports persistence of large models in contemporary databases. Morsa [27] was one of the first approaches to provide persistence of large scale EMF models using NoSQL databases. Morsa is backed by MongoDB and is built atop EMF so that standard EMF interfaces can be used for persisting the models. MongoEMF[9] and Neo4EMF [22] are two additional alternatives for storing EMF models in MongoDB and Neo4J[10] (graph-based) databases respectively.

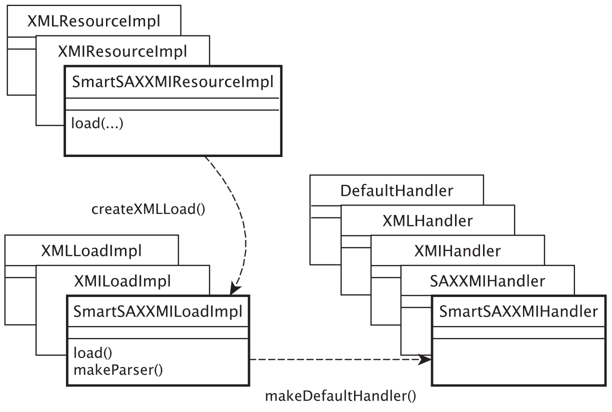


Figure 11: SmartSAX as an extension to EMF

EMF fragments [28] is another persistence layer for EMF that uses NoSQL databases to achieve fast storage of new data and fast navigation of persisted models. In EMF fragments, models are automatically partitioned in fragments so that all data from a single fragment is loaded at a time, and links to other fragments are loaded on demand. However, EMF fragments requires that metamodels to be modified to indicate where the partitions should be made to get the partitioning capabilities. In addition, EMF fragments are beneficial to a particular set of queries that the fragmentation strategies are designed specifically for. Therefore, EMF requires the re-fragmentation of models for it to be applicable for other types of queries.

As models grow in size, database-based persistence or a combination of model fragmentation and indexing [29] are clearly preferable compared to persisting models in large monolithic XMI files. However, when working with off-the-shelf modelling tools, large XMI files are often a “given”. Even if engineers choose to transform such models into a database-backed representation before they process them further (e.g. to validate them or to generate code from them), they need to parse the provided XMI representation, in which cases the proposed partial loading approach can also be useful.

Overall, we do not view the proposed partial XMI loading approach as a competitor to database-based model persistence approaches, but rather as a complementary facility when large XMI models are a given due to factors beyond the control of the engineers.

5. EVALUATION

In this section we report the results of benchmarks performed on a prototype implementation (SmartSAX) of the algorithm described in Section 3 to evaluate the scalability and practicality of the proposed approach. Benchmarks were performed on a computer with Intel(R) Core(TM) i7

[9]http://github.com/BryanHunt/mongo-emf/wiki
[10]Neo4J Graph Database: http://neo4j.com

CPU @ 2.3GHz, with 8GB of physical memory, running OS X Yosemite. The version of the Java Virtual Machine used was 1.8.0_31-b13.

For our benchmarks, models of varying sizes obtained from reverse engineered Java code in the 2009 GraBaTs contest[11] are used. These models, named set0 – set4 (9.2MB, 27.9MB, 283.2MB, 626.7MB, 676.9MB respectively) are stored in XMI 2.0 format and have been used for benchmarks in various tools [22, 29, 30].

5.1 Loading Unit Coverage

To quantify partial loading in our benchmarks we use the concept of *loading units*. We identify three types of such units: objects (model elements), attribute values, and reference values. For example, the partially-loaded model of Figure 10 consists of 8 loading units (3 objects, 3 attribute values (we exclude *Module.name*) and 2 reference values), while the fully-loaded model of Figure 5 consists of 24 loading units (7 objects, 9 attribute values and 8 reference values).

With regard to our experiment, our first step was to count the number of loading units in each of the models (from set0 to set4) using an EOL query[12]. Then, EOL programs which exercise 20%, 40%, 60%, 80% and 100% of the loading units for models from set0 to set4 were generated. An example of such programs is illustrated in Listing 2. These programs were analysed by the Epsilon static analysis framework for effective metamodel extraction. Finally, the effective metamodels extracted from the EOL programs were used to load all models using SmartSAX. The performance in terms of loading time and memory consumption were recorded and compared with the performance of the default EMF XMI parser on the same models.

```
var size = 0;
var markerAnnotation = MarkerAnnotation.all.
    first();
size = size + MarkerAnnotation.all.
    resolveBoxing.size();
size = size + MarkerAnnotation.all.
    resolveUnboxing.size();
size = size + MarkerAnnotation.all.typeBinding.
    size();
...
size.println();
```

Listing 2: An example of generated EOL program for loading unit coverage

5.2 Results

The obtained results for data sets set0 – set4 are presented in Figure 5.2 – 5.2. From the benchmark results we observe that the partial loading algorithm demonstrates a linear behaviour with respect to loading unit coverage, both in terms of loading time and memory consumption. We also observe that to load 100% of the model, SmartSAX requires marginally more time and memory. This is due to the (small) overhead incurred by the additional checks it involves when creating objects (i.e. the time to consult the effective metamodel).

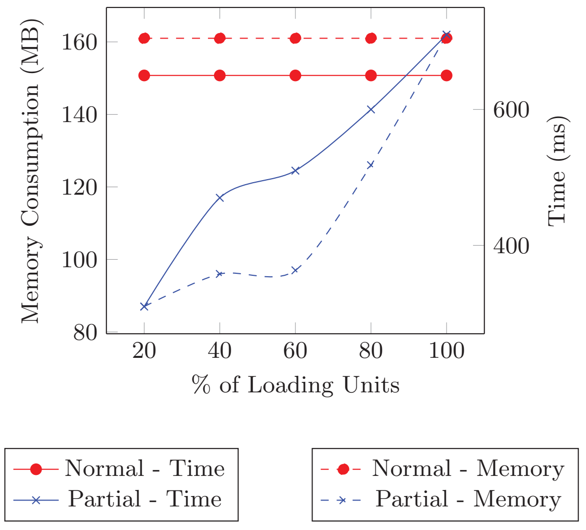


Figure 12: Benchmark results for Set 0

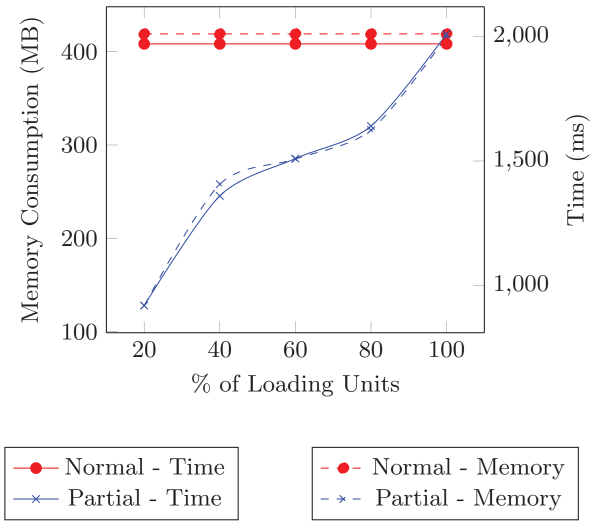


Figure 13: Benchmark results for Set 1

5.3 Threats to Validity

For the experiments reported in this paper we have only used XMI-based models conforming to the JDTAST mestamodel used in the GraBaTs 2009 contest. The performance and memory footprint of the partial loading algorithm may be sensitive to particular features of the metamodel – although, analytically, we have not identified any reasons why this would be the case.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel approach that enables partial loading of XMI-based models. The proposed approach tackles scenarios where read-only access to the models is sufficient and where the parts of the model (typically a small subset of the entire model) that are of interest are

[11]GraBaTs2009: 5th Int. Workshop on Graph-Based Tools, http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/

[12]https://github.com/wrwei/SmartSAX/blob/master/src/org/eclipse/epsilon/labs/smartsax/coverage_generator.eol

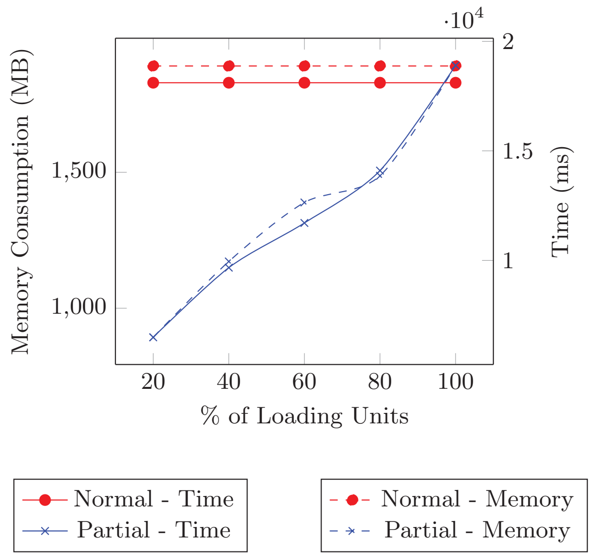


Figure 14: Benchmark results for Set 2

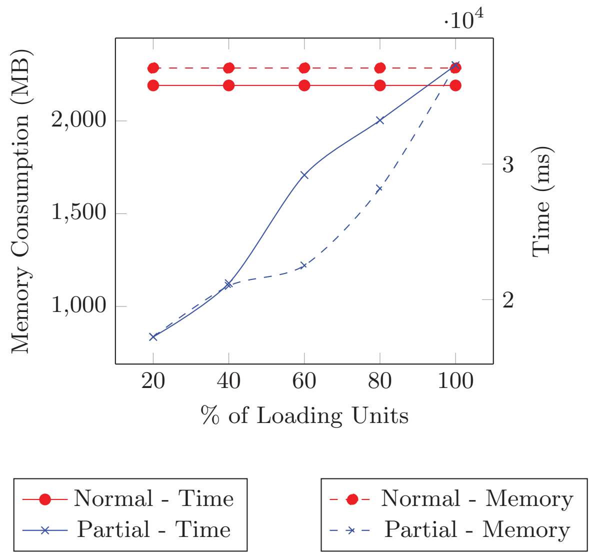


Figure 15: Benchmark results for Set 3

known in advance. Extensive benchmarking has illustrated that the proposed algorithm scales linearly with respect to the size of the part of the model that is of interest, both in terms of loading time and in terms of memory footprint.

We are currently working on extending the effective metamodel extraction algorithm to support additional languages of the Epsilon platform (i.e. Epsilon's M2M and M2T transformation, and model validation languages) and on supporting *fragment paths* so that our partial loading algorithm can also accommodate XMI-based models that do not make use of XMI ids. In addition, we are working on a facility which compares the effective metamodel with the actual metamodel, in the sense that if the effective metamodel is "identical" to the actual metamodel (through comparison), SmartSAX switches to normal loading, which out-performs partial loading for 100% model coverage.

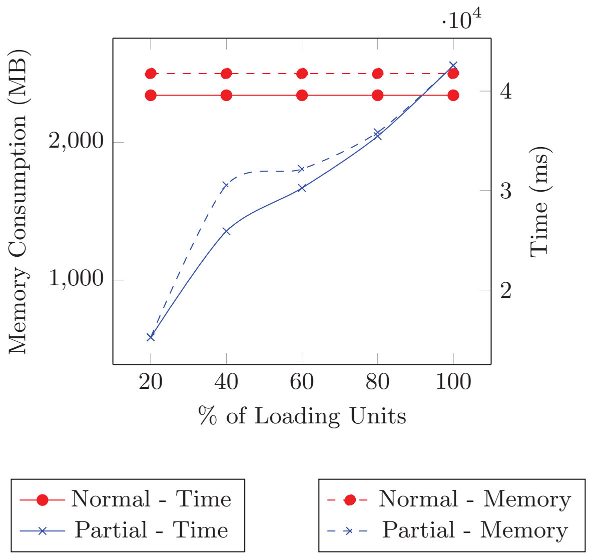


Figure 16: Benchmark results for Set 4

Acknowledgments

This research was part supported by the EU, through the MONDO FP7 STREP project (#611125).

7. REFERENCES

[1] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[2] Object Management Group. Object Constraint Language. http://www.omg.org/spec/OCL/. Accessed: 01-01-2016.

[3] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud, and Freddy Allilaire. Acceleo User Guide. http://www.acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf. Accessed 01-01-2016.

[4] S. Mitra and Kee Sup Kim. XPAND: an efficient test stimulus compression technique. *Computers, IEEE Transactions on*, 55(2):163–173, Feb 2006.

[5] Jean Bézivin, Frédéric Jouault, and David Touzet. An Introduction to the Atlas Model Management Architecture. *Rapport de recherche*, (05.01):10–49, 2005.

[6] Dimitrios Kolovos, Louis Rose, Richard Paige, and A Garcia-Dominguez. *The Epsilon Book*. Eclipse, 2010.

[7] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. The Epsilon Generation Language. In *Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2008.

[8] Parastoo Mohagheghi, MiguelA. Fernandez, JuanA. Martell, Mathias Fritzsche, and Wasif Gilani. MDE Adoption in Industry: Challenges and Success Criteria. In Michel R.V. Chaudron, editor, *Models in Software Engineering*, volume 5421 of *Lecture Notes in Computer Science*, pages 54–59. Springer Berlin Heidelberg, 2009.

[9] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context âĂŤ Motorola Case Study. In Lionel Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 476–491. Springer Berlin Heidelberg, 2005.

[10] Dimitrios S. Kolovos, Richard F. Paige, and Fiona AC Polack. Scalability: The holy grail of model driven engineering. In *ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering*, pages 10–14, 2008.

[11] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot. A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 2:1–2:10, New York, NY, USA, 2013. ACM.

[12] Ari Jaaksi. Developing Mobile Browsers in a Product Line. *IEEE software*, 19(4):73–80, 2002.

[13] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. Evaluating the Use of Domain-Specific Modeling in Practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*, 2009.

[14] Jordi Cabot and Martin Gogolla. Object Constraint Language (OCL): A Definitive Guide. In *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science*, pages 58–90. Springer Berlin Heidelberg, 2012.

[15] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *Lecture Notes in Computer Science*, pages 204–218. Springer Berlin Heidelberg, 2009.

[16] Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM.

[17] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM.

[18] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin Heidelberg, 2006.

[19] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2008.

[20] beo group. Acceleo. http://www.eclipse.org/acceleo/. Accessed: 01-01-2016.

[21] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. Merging Models with the Epsilon Merging Language (EML). In *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin Heidelberg, 2006.

[22] Amine Benelallam, Abel GÃşmez, Gerson SunyÃľ, Massimo Tisi, and David Launay. Neo4EMF, A Scalable Persistence Layer for EMF Models. In *Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer International Publishing, 2014.

[23] J McQuillan and J Power. White-box Coverage Criteria for Model Transformations. In *Proceedings of the First International Workshop on Model Transformation with ATL*, pages 63–77, 2009.

[24] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture – Foundations and Applications: Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006. Proceedings*, pages 128–142. Springer Berlin Heidelberg, 2006.

[25] Ran Wei and Dimitris S Kolovos. Automated analysis, validation and suboptimal code detection in model management programs. In *Proc. 2nd International Workshop on Scalable Model Driven Engineering (BigMDE)*, pages 48–57, 2014.

[26] Frédéric Jouault, Jean Bézivin, and Mikaël" Barbero. Towards An Advanced Model-Driven Engineering Toolbox. *Innovations in Systems and Software Engineering*, 5(1):5–12, 2009.

[27] Javier Espinazo Pagán, Jesúss Sánchez Cuadrado, and Jesús García Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 77–92. Springer Berlin Heidelberg, 2011.

[28] Markus Scheidgen. Reference Representation Techniques for Large Models. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 5.1–5.9, New York, NY, USA, 2013. ACM.

[29] Konstantinos Barmpis and Dimitris Kolovos. Hawk: Towards a Scalable Model Indexing Architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 6:1–6:9, New York, NY, USA, 2013. ACM.

[30] Seyyed M. Shah, Ran Wei, Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Konstantinos Barmpis. *Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, Valencia, Spain, September 28 – October 3, 2014. Proceedings*, chapter A Framework to Benchmark NoSQL Data Stores for Large-Scale Model Persistence, pages 586–601. Springer International Publishing, 2014.