Towards Scalable Model Indexing

Konstantinos Barmpis

Engineering Doctorate

University of York
Computer Science

March 2016

Abstract

Model-Driven Engineering (MDE) is a software engineering discipline promoting *models* as first-class artefacts of the software lifecycle. It offers increased productivity, consistency, maintainability and reuse by using these models to generate other necessary products, such as program code or documentation. As such, persisting, accessing, manipulating, transforming and querying such models needs to be efficient, for maintaining the various benefits MDE can offer. Scalability is often identified to be a bottleneck for potential adapters of MDE, as large-scale models need to be handled seamlessly, without causing disproportionate losses in performance or limiting the ability of multiple stakeholders to work simultaneously on the same collection of large models. This work identifies the primary scalability concerns of MDE and tackles those related to the querying of large collections of models in collaborative modeling environments; it presents a novel approach whereby information contained in such models can be efficiently retrieved, orthogonally to the formats in which models are persisted. This approach, coined *model indexing* leverages the use of file-based version control systems for storing models, while allowing developers to efficiently query models without needing to retrieve them from remote locations or load them into memory beforehand. Empirical evidence gathered during the course of the research project is then detailed, which provides confidence that such novel tools and technologies can mitigate these specific scalability concerns; the results obtained are promising, offering large improvements in the execution time of certain classes of queries, which can be further optimized by use of caching and database indexing techniques. The architecture of the approach is also empirically validated, by virtue of integration with various state-of-the-art modeling and model management tools, and so is the correctness of the various algorithms used in this approach.

For my grandfather Costas

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acknowledgments

I would like to thank my supervisor Dr. Dimitrios Kolovos for his continuous guidance and support throughout my degree. I am grateful to Prof. Richard Paige and Dr. Radu Calinescu for their encouragement as well as to Dr. Louis Rose for his feedback on my work.

I thank my colleagues and friends in the Enterprise Systems group and the department, especially Ran Wei, Dr. James Williams, Adolfo Sanchez-Barbudo Herrera, Athanasios Zolotas, Septavera Sharvia, Frank Burton and Dr. Antonio Garcia Dominguez for engaging in interesting discussions, for providing valuable insights and for their endless support. I thank the various MONDO project partners, particularly Gábor Szárnyas, for being exceptional collaborators and for helping integrate various technologies with my work.

Finally, to my parents and grandparents, thank you for always being there and for tirelessly encouraging me to achieve my goals.

Author Declaration

Except where stated, all of the work contained in this thesis represents the original contribution of the author. This work has not been submitted for any other award at this or any other institution.

This work was partially funded by the MONDO EU project and contributed to its various written deliverables of workpackage 5, specifically to all of the deliverables D5.1 through D5.6. The core ideas of this work, including the architecture and design of the system presented here, have not been a product of any collaborations within the MONDO project. Sections 4.3.4.2 and 5.4 detail collaborative work performed for integrating this work with other MDE tools and technologies in the form of additional drivers that can be used alongside the core system presented in this work.

Parts of the work described in this thesis have been previously published by the author:

- **Hawk: towards a scalable model indexing architecture.** Konstantinos Barmpis and Dimitrios S. Kolovos. In Proceedings of the Workshop on Scalability in Model Driven Engineering, BigMDE '13, pages 6:10–6:9, New York, NY, USA, June 2013. ACM.

- **Evaluation of contemporary graph databases for efficient persistence of large-scale models.** Konstantinos Barmpis and Dimitrios S. Kolovos. Journal of Object Technology, 13-3:3:1–26, July 2014.

- **Towards scalable querying of large-scale models.** Konstantinos Barmpis and Dimitrios S. Kolovos. In Proceedings of the 10th European Conference on Modelling Foundations and Applications. ECMFA'14, July 2014.

- **Towards incremental updates in large-scale model indexes.** Konstantinos Barmpis, Seyyed Shah, and Dimitrios S. Kolovos. In Proceedings of the 11th European Conference on Modelling Foundations and Applications. ECMFA'15, July 2015.

1. Introduction

In today's fast-paced competitive world, software engineers are pushed to create larger and more complex distributed systems every day, often with a very small time to market. This introduces the requirement for rapid iterations over quickly developed prototypes which creates the need for more efficient tools and techniques for developing, testing and deploying such systems in a systematic manner.

For achieving this goal, the discipline of Model-Driven Engineering offers increased productivity by automating or semi-automating effort-intensive and error-prone steps of the software engineering process by the use of *models* as first-class citizens.

1.1. Overview of Model-Driven Engineering

MDE allows for the rapid creation and management of software systems by using models that can be transformed to generate lower level necessary artefacts such as code and documentation. As such systems and their models grow in size, concerns regarding scalability and collaborative development emerge. Many widely-used modeling tools and technologies tend to rapidly reach their limits as they exhaust their resources when trying to manage very large models, either taking extensive periods of time to perform even the most basic tasks, or not being able to perform them at all. Furthermore, as multiple developers need to collaborate on the creation and use of such models, reliable forms of distribution and versioning of these models need to be available and integrated into the development process. This thesis focuses on identifying these limitations and investigating solutions for tackling such very large models, with Section 2.1 providing an initial review of the field.

1.2. Overview of Versioning Systems

In software engineering, it is common practice to maintain a record of previous revisions of the software; this facilitates the collaboration on any artefacts as well as providing a reliable form of recovery and change management. When versioning models, there are two primary approaches that can be taken:

The first is to use one of the most popular forms of version control, which is based around versioning text files (commonly used for files comprising program source code). Such systems offer the ability to efficiently store large collections of rapidly evolving files by only storing changes (deltas) between revisions, which is effective for assisting in the collaborative development of large numbers of text files, managed by multiple developers. This approach works well with text-based model persistence formats like XML as they lend themselves to this paradigm; on the other hand, many other model persistence formats, such as any database-based persistence, which use binary files, do not. When binary files are used, the version control system will likely need to store the entire contents of each file even for minor changes.

The second approach is to use dedicated model-specific version control systems. This approach has the advantage that it is built with a semantic understanding of MDE (knows about the concepts of metamodels and models, as opposed to files and lines) and can hence store model-based deltas that will usually be more fine-grained than text-based file deltas, which allows for the appropriate model-level fragmentation of the commits. The downside is that such systems require a tight integration between themselves and the modeling technology used (in order to be provided with the necessary semantics), which is in contrast to the file-based paradigm, where the concerns are orthogonal. Section 2.2.2 reviews such systems in the context of MDE.

1.3. Motivation and Research Hypothesis

The need for tackling scalability in a collaborative MDE context introduces the need for researching ways to improve this area. From a theoretical standpoint, there is the need for offering a way to handle very large models (with respect to the various model management operations commonly required in MDE) without running into resource starvation issues, as well as being able to facilitate the collaborative development of such models from multiple developers. From a practical standpoint, various industrial collaborators of the MONDO project[1] used to host this work (Ikerlan[2], Softeam[3], Soft-Maint (subsidiary of Sodifrance[4]), Uninova[5]) have expressed the need for their models to be handled in a more scalable way in their respective domains.

Today, if models are versioned using classical file-based version control systems they will either have to be stored as a single monolithic file (which may often cause issues with loading such a large file into memory) or as a collection of interconnected smaller model fragment files. Each developer will commonly only store and manipulate their own subset of the model fragments and in order to query information about the entire model would need to fetch all other model fragments (if they are not stored locally) and then load all of the fragments into memory in order to ensure that a complete result is returned. This returns to the original problem of having to load a large model in its entirety, or risks causing a loss in the big picture when only some of the fragments are loaded (as global information regarding other model fragments cannot be obtained). On the other hand if model-specific version control is used, the development process will likely have to be altered in order to be aligned with the procedures and techniques used by the specific versioning tool (as it will have to be tightly integrated with the modeling technology in order to function).

The focus of this work is on providing a solution which does not attempt to radically alter the techniques used in managing large models in MDE, but attempts at offering a medium between currently used state-of-the-art technologies and novel ones which may take little account of current practice and norms in the field. The proposed approach introduces the concept of *model indexing*, whereby a separate system (a *model index*) is introduced that holds a read-only representation of the models stored in file-based version control systems. The model index can be used as an efficient way to query large (collections of) models without incurring the cost of having to load them locally or losing the big picture if only some of them are loaded. As such, the research hypothesis of this thesis is as follows:

[1] This project is partially funded by the MONDO FP7 STREP European Union research project (#611125) which brings together various universities and businesses for tackling scalability in MDE: `http://www.mondo-project.org/`
[2] `http://www.ikerlan.es/en/`
[3] `http://rd.softeam.com/`
[4] `http://www.sodifrance.fr/`
[5] `http://www.uninova.pt/`

The overhead of computing model-element-level queries over large (collections of) models stored in a file-based VCS can be significantly reduced using a non-invasive model-indexing system orthogonal to the specific VCS or model representation format.

1.4. Research Results

This research proposes a novel approach for tackling scalability in MDE; a model indexing framework is presented alongside a prototype implementation that offers an extensible way to manage large collections of models developed by multiple stakeholders. The work is validated through extensive empirical evaluation with the results supporting the research hypothesis with models of the order of millions of model elements being managed, providing up to 95.1% decrease in execution time for executing certain queries, offering incrementality for keeping the index up to date (which is shown to provide an average of 70.7% decrease in execution time when compared to a non-incremental approach) and suggesting that such a model indexing approach can be likely generalized for other modeling technologies as well as other types of persistence back-ends and version control systems. Finally the work provides a comprehensive review and categorization of current state-of-the-art tools and technologies used in MDE, presenting their key benefits and limitations, as the proposed solution does not attempt to replace such technologies but instead offers an alternative approach suitable for some scenarios (identified in Section 3.4).

1.5. Thesis Structure

Chapter 2 provides an overview of MDE with Section 2.1 presenting the discipline and introducing the reader to the observed scalability concerns; Section 2.2 reviews various widely used state-of-the-art tools and technologies available today and describes their functionalities and unique characteristics.

Chapter 3 presents the analysis of the problem at hand and states the research hypothesis this research project aims at proving, it defines the research objectives and the scope the work is limited to, identifying areas to be investigated.

Chapter 4 details the architecture, design and implementation of a prototype tool developed as a proof-of-concept for managing scalability in MDE. Section 4.1 presents the capabilities such a system is envisioned to have; Section 4.2 presents the system architecture including its various components; Section 4.3 details the design, including the various APIs the tool offers and delves into the specifics of each component individually, as well as into the various key procedures and algorithms used by the system; lastly Section 4.4 briefly presents some implementation details and a user guide for the tool.

Chapter 5 analyzes empirical data gathered as part of evaluating the aforementioned prototype and discusses its significance with respect to various functional and non-functional properties of the system. Section 5.1 presents the strategy used for evaluating the system; Section 5.2 presents the benchmarks used; Section 5.3 details the results obtained and discusses their significance; Section 5.4 talks about the various integration efforts made to allow the system to work with other widely used tools in the field; and finally Section 5.5 mentions the various alternative drivers developed for the system that can be used instead of (or alongside) the primary ones developed as part of this work, as a way to evaluate the architecture of the system.

Finally Chapter 6 synthesizes a summary of all the knowledge gained in this work and concludes with directions this work can take in the future as well as identifying the contributions it has offered to the field of model-driven software engineering.

2. Background

This chapter provides an introduction to MDE (Section 2.1) and then focuses on existing work related to model persistence and versioning. Section 2.2.1 provides a discussion on data persistence back-ends/formats and how they are leveraged to store models, Section 2.2.2 focuses on model versioning, with Section 2.2.2.1 providing a discussion on file-based version control systems followed by a discussion of model-based version control (Section 2.2.2.2). Finally, Section 2.2.3 introduces the reader to model querying with respect to various state-of-the-art technologies used today.

A background in MDE is needed in order to understand the origin of the scalability problems, as well as for being able to gain insight for coming up with a solution that addresses some of the challenges in this field of study. Information about various forms of version control used in MDE is useful for understanding their limitations, when used in collaborative work on model development, as well as for insight on building a tool suited for scalable multi-user model management. Finally, detailed knowledge of the various forms of model persistence, such as file or database serializations, is paramount for an effective implementation of a scalable solution, as well as for discovering, designing, implementing, testing and comparing the most suitable back-end store itself.

2.1. Model-Driven Engineering

MDE is an approach to software development, where models are first class artefacts of the software engineering process. A model, in this context, as described by [1], is “a description of phenomena of interest”, represented in textual, graphical or other form (such as tabular or tree-based). In MDE, models are used to describe a system and (partly) automate its implementation through automated transformation to lower-level artefacts. In order for models to be amenable to automated processing, they must be defined in terms of rigorously specified modeling languages (metamodels). This section will present several key aspects of MDE, give examples of different widely used model management approaches taken by the community, and discuss their importance. Furthermore it will present the challenges the current state-of-the-art MDE tools face when dealing with large (collections of) models in a collaborative environment.

2.1.1. Modeling and Automated Model Management

MDE has several standard processes and activities, the most prominent of which are presented below.

2.1.1.1. Modeling Languages

Models used in MDE typically adhere to a rigid set of rules that are encapsulated in a *metamodel*. This "blueprint" of the model contains all of the allowed syntax, restrictions and features that models conforming to it can have. Modeling languages (metamodels) are often separated into two categories (domain-specific and general-purpose), but can also be seen as a continuum from minimal (domain-focused) to maximal expressiveness.

General-purpose languages include languages such as the Unified Modeling Language (UML) [2], the Business Process Model and Notation (BPMN) [3], the Systems Modeling Language (SysML) [4] and the Archimate enterprise architecture modeling language [5] and allow for a wide variety of models to be created, capturing a broad spectrum of concerns. A general-purpose language tries to support a wide variety of concepts and favors portability and maintainability [6] as it is intended to be used and understood by a wide audience.

Domain-specific languages include WebML [7] or the Systems Biology Markup Language (SBML) [8] and are tailored to represent a specific area (domain) of interest. While limited in their scope, they provide the ability for solutions to be expressed at the same level of abstraction as the problem domain as well as more concisely, hence allowing domain experts to work with their area of expertise. They are often optimized for productivity and are less prone to portability [6].

2.1.1.2. Metamodeling Architectures

Various architectures are available for defining metamodels and constructing models that conform to them, with different numbers of layers of abstraction used to represent their artefacts. Their main commonality is that they all define at least two layers, one for the model used to describe a system/process and one for the metamodel used to define such models.

The MetaObject Facility (MOF[1]) The Object Management Group (OMG) [9] introduced MDA, in 2001 together with an architecture for defining metamodels (Figure 2.1) and constraints on models conforming to these metamodels [10].

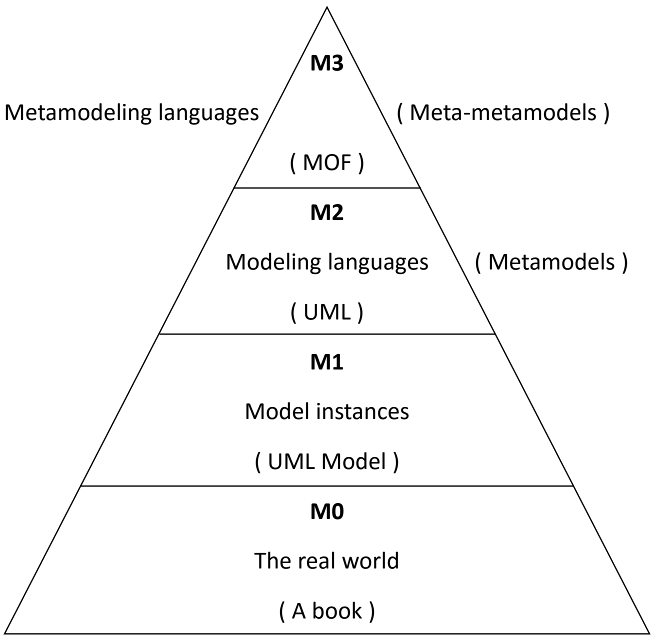


Figure 2.1.: The four layers of model abstraction, based on [10]

In the pyramid seen in Figure 2.1, M0 represents the real-world such as the physical elements which are to be modeled (a book or a customer) or the software system, if software is being modeled. M1 then represents the model of the real-world (found in M0) such as a book being represented by a class with attributes such as ISBN and title. M2 represents the modeling language (metamodel) used to define elements in M1 (such as UML [2]). Finally, M3 represents the meta-modeling language (meta-metamodel) used to define elements in M2; for the OMG's MDA the only M3 element is MOF [11].

It is worth noting that this type of layering (MOF – UML – UML model – real-world) can be paralleled to how computer software and programing languages are layered

[1] `http://www.omg.org/mof/`

(Backus-Naur Form (BNF) [12] – Java Language – Java Program – real-world). In the same way that BNF defines the abstract syntax of Java, MOF defines the abstract syntax of UML.

The MOF architecture is not discussed in more detail as it is very similar in structure and capabilities to the Eclipse Modeling Framework discussed in detail below.

MetaCase – The MetaEdit+ Tool MetaCase[2] offers a graphical metamodeling architecture whereby custom domain-specific metamodels can be created using an editor (The MetaEdit+ Workbench Tool[3]) that uses visual notation (shapes, colors etc.) to define the various concepts and their features/rules. This metamodel (which would come under the M3 layer in MDA) is seen in Figure 2.2 from [13].

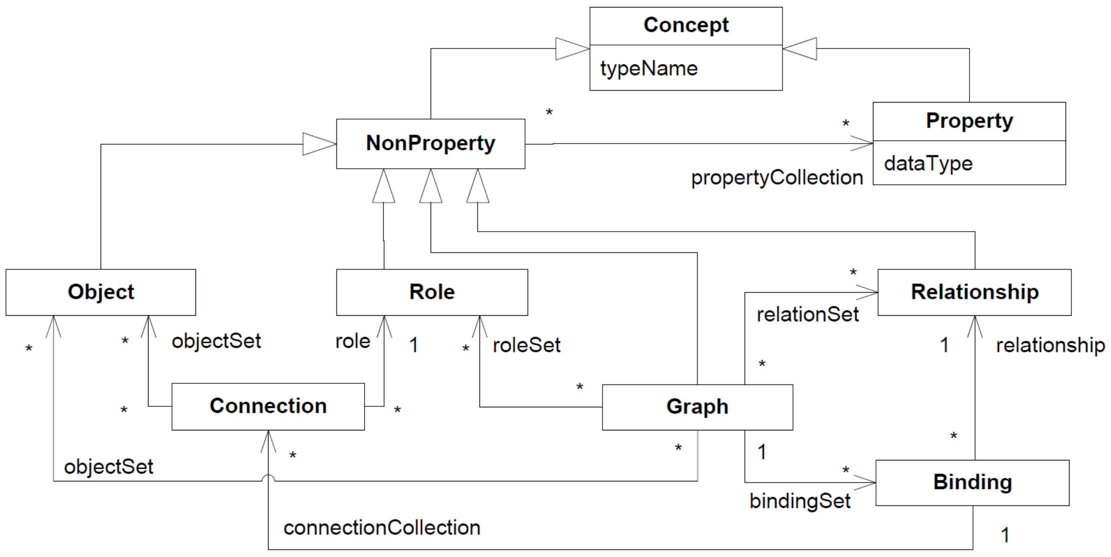


Figure 2.2.: The GOPRR metamodel, from [13]

In this architecture, a metamodel is defined as a *Graph*, containing *Object*s, *Relationship*s and *Role*s. An *Object* describes a class of model element, which can be connected by *Relationship*s to others; the *Role* specifies how such *Objects* participate in a *Relationship*. *Binding*s connect a *Relationship* with two or more *Roles* and one or more *Objects* for each *Role* in a *Graph*. All these concepts (barring *Binding*s) can have *Properties* which can be of various data types such as Strings, Booleans, numbers etc. [13].

[2] http://www.metacase.com/
[3] http://www.metacase.com/mwb/

Using these concepts, and with the help of the MetaEdit+ Workbench editor, users are able to define their domain-specific concepts from scratch or use one of the pre-defined metamodels offered; this Workbench editor allows for the creation of symbols for the various *Object*s, *Relationship*s and *Role*s. The metamodel can be used to define instance models that use these concepts and follow any defined rules, and these models can then be used to generate further artefacts such as documentation or code.

The Eclipse Modeling Framework (EMF) EMF [14] is a framework that facilitates the definition and instantiation of metamodels. In EMF, metamodels are defined using the Ecore metamodeling language, a high level overview of which is illustrated in Figure 2.3[4]. As EMF is the most widely-used metamodeling framework today, we delve into more detail on its structure and capabilities.

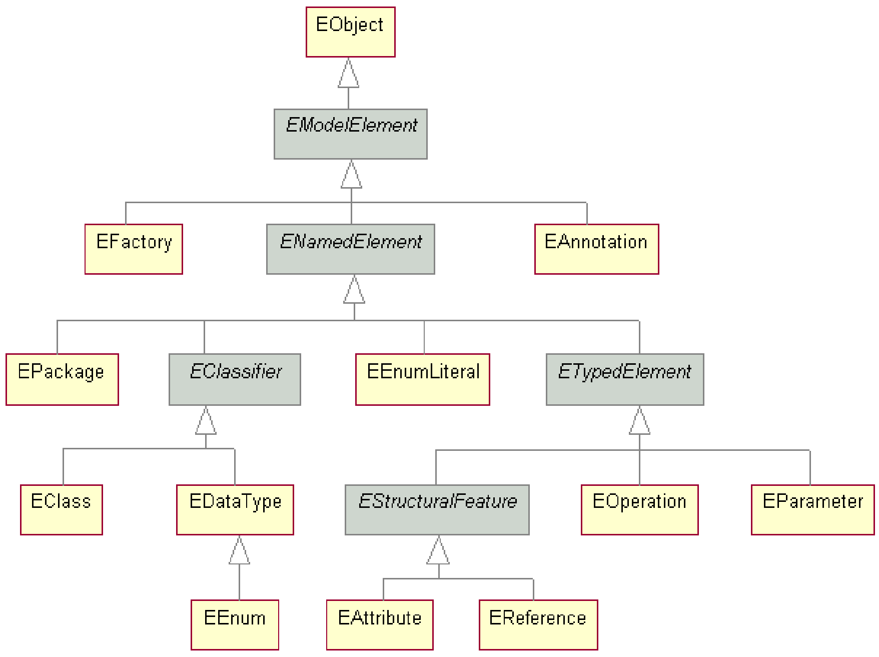


Figure 2.3.: The Ecore Metamodeling Language

Metamodels are expressed in the form of *EPackage*s; these define the unique global identifier of the metamodel (its *nsURI* – used as a unique key for retrieving metamodels from a registry (*EPackageRegistry*) that stores a map of identifiers to metamodels) and contain the types found in the metamodel, in the form of *EClassifier*s. Types can be

[4] `http://download.eclipse.org/modeling/emf/emf/javadoc/2.8.0/org/eclipse/emf/ecore/package-summary.html`

either *EClass*es or *EDataType*s (both of which are *ENamedElement*s); *EClass*es contain their features, in the form of *EAttribute*s (that have an *EDataType* as their value type), and *EReference*s (that have an *EClass* as their value type). Model elements instantiated from these *EClass*es contain values for the *EAttribute*s and *EReference*s defined by their type (and its supertypes).

EMF uses the concept of a *Resource* for maintaining an in-memory representation of metamodels and models. Such *Resource*s can be serialized to disk (in various forms such as textual or database, discussed in the sequel) and read into memory from their respective serializations. Contents of a *Resource* can be iterated through and modified, with EMF automating the majority of both inter-resource and intra-resource consistency operations. For example EMF will automatically keep any loaded resources synchronized with model element deletion (with respect to any other elements referencing them) or modification.

By default, EMF serializes models in a standard XML[5]-based representation called the XML Metadata Interchange (XMI[6]), an OMG-standardized format that was designed to enhance tool-interoperability. This form of serialization allows for several optimizations such as the ability to save a model in multiple physical files and to use lazy reference resolution to link them together. Due to the nature of the format, models serialized in XMI are quite verbose and need to be de-serialized in a sequential manner. For example the default EMF XMI parser (SAX[7]-based) needs to read the entire model file before it can create a usable in-memory resource.

2.1.1.3. Model Querying and Modification

Retrieving information from stored models, as well as being able to evolve them by altering their contents is of paramount importance in MDE. Queries are of great use for answering complex questions about large models, which may be required by the stakeholders; furthermore they are required in order to perform model transformations, described below. Since model queries are used for most other model management operations, tackling issues such as scalability and collaborative development for querying can be beneficial for all other operations that require querying capabilities. As such, this thesis focuses on solving querying concerns, in anticipation that this work will facilitate

[5] `http://www.omg.org/spec/XML/`
[6] `http://www.omg.org/spec/XMI/`
[7] `http://sax.sourceforge.net/`

more efficient model transformations etc. Model querying is commonly performed using third generation languages such as Java or model query languages like OCL [15] or EOL [16]. Modification of stored models should provide a consistent way to add or remove model elements or to modify their properties, without compromising their integrity and (if the tool supports it) validity.

2.1.1.4. Model Transformation

Models need to be transformed to other representations, in order to generate useful artefacts such as runnable code, documentation or models conforming to a different metamodel. There are two distinguishable types of transformations that can be performed on models – model to model and model to text – a high-level overview of which is seen in Figure 2.4, and described below.

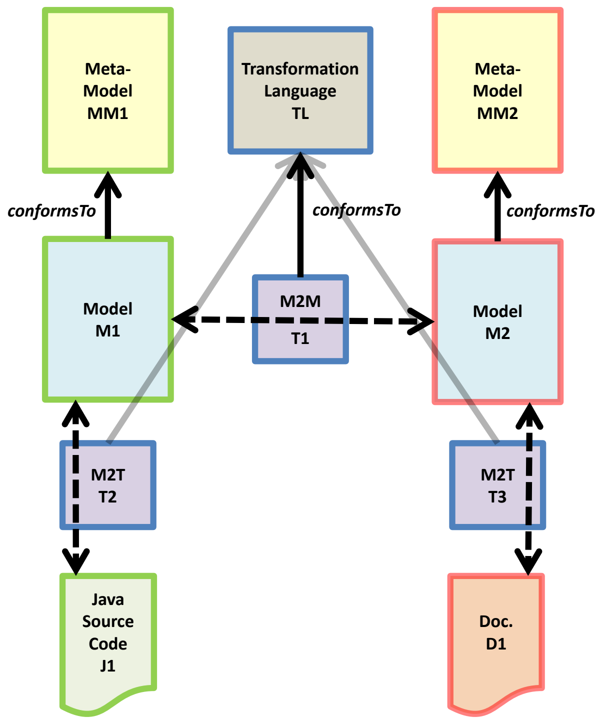


Figure 2.4.: High-level overview of various types of model transformation

Model to Model The first form of model transformation is Model to Model (M2M). In this paradigm, input model *M1* adhering to metamodel *MM1* is transformed using a transformation (or a set of transformations) *T1* adhering to transformation language *TL* to output model *M2* adhering to metamodel *MM2*. Such transformations allow for the (partial) mapping of a model onto one of a potentially different metamodel (such as mapping a model representing UML Classes to one representing a Relational Database, for example).

There are three types of M2M transformation languages, as presented by [17]:

- Declarative Languages. Declarative approaches focus on what needs to be transformed, by defining a relation between the source and target models. Such approaches usually automate features like source model traversal, traceability and bidirectionality. On the other hand such approaches often do not allow for fine-grained control of the transformation execution (such as rule scheduling), which may be limiting in some cases. An example of a language using this approach is QVT-Relations [18] by the Object Management Group (OMG).

- Imperative Languages. Imperative approaches focus on how the transformation needs to be performed by specifying the steps required to get to the target models from the source models. Such approaches are beneficial when declarative approaches do not provide a sufficient level of control; for example when the application order of a set of transformations needs to be controlled explicitly (as such languages understand notions like sequence or selection). Nevertheless issues such as traceability and bidirectionality have to be manually resolved. An example of a language using this approach is QVT-Operational [19] by the OMG.

- Hybrid Languages. As both of the above paradigms have benefits and drawbacks, "hybrid languages provide a declarative rule-based execution scheme as well as imperative features for handling complex transformation scenarios" [20]. Nevertheless, such languages usually have limited efficiency and lack of optimization, which has to be manually performed [21]. Examples of languages using this approach are the ATLAS Transformation Language [21, 22] and the Epsilon Transformation Language [20].

Model to Text The second form is Model to Text (M2T); Here, input model *M1* adhering to metamodel *MM1* is transformed using transformation *T2* adhering to transformation language *TL* to produce one or more text files as output. Such output can have two main forms: the first is a plain text output such as a report document file

(*D1* – in the example of Figure 2.4) and the second is in the form of a grammatically structured output such as a Java source code file (*J1* – in the example of Figure 2.4 – structured in the Java programming language grammar). Such transformations allow for the automated (or partially automated) generation of runnable code from a model or for human-readable documentation to be created. Examples of languages used to perform such transformations are the Epsilon Generation Language (EGL) [23], Xpand [24], Acceleo [25] and Jet [26].

As the aim of this work is not to contribute to the area of modeling and automated model management, this section presented only a brief overview of some of the topics in this area; others such as model validation, simulation, comparison, merging and refactoring are out of the scope of this work.

2.1.2. MDE with Large-Scale Models

The popularity and adoption of MDE in industry has dramatically increased in the past decade as it provides several benefits compared to traditional software engineering practices, such as improved productivity and reuse [27], which allow for systems to be built faster and cheaper. Nevertheless, certain limitations of the current state of the art tools used in MDE such as scalability concerns are seen to be preventing its wider use in industry [28, 29] and need to be overcome. Scalability issues arise when large models (or collections of models – of the order of millions of model elements) are used in MDE processes. As summarized by [30], "Loading and storing big models is a resource and time-consuming activity".

2.1.2.1. Scalability

When referring to scalability issues in MDE, they can be split into the following categories [31], seen in figure 2.5:

Model persistence Storage of large models and the ability to access and update such models with low memory footprint and fast execution time can be seen as a bottleneck in MDE-based tools, when large models, in the order of millions of elements, are used. Especially in industry, several non-functional properties (performance, availability, security, etc.) often need to be met in order for adoption of a tool. A detailed description

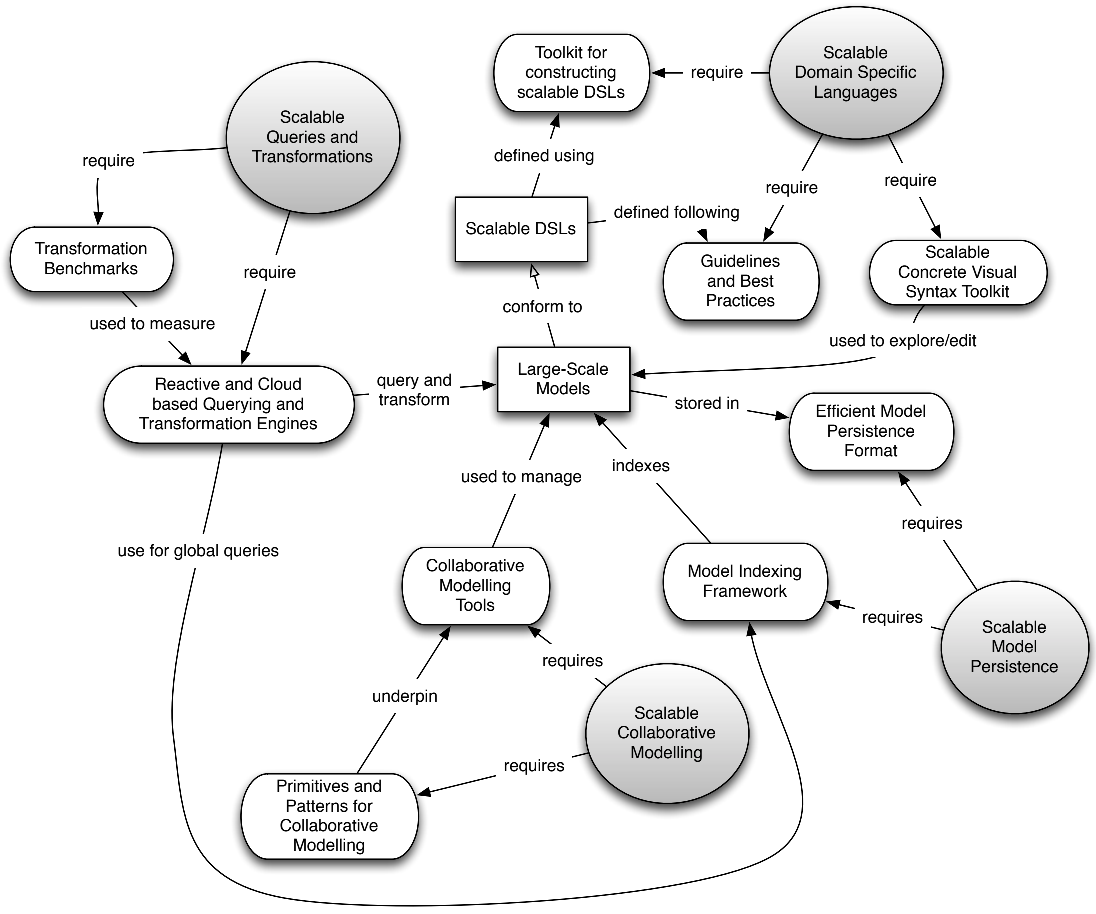

Toolkit for constructing scalable DSLs
require
Scalable Domain Specific Languages
Scalable Queries and Transformations
require
defined using
require
require
Scalable DSLs
defined following
Transformation Benchmarks
require
Guidelines and Best Practices
Scalable Concrete Visual Syntax Toolkit
conform to
used to measure
used to explore/edit
Large-Scale Models
Reactive and Cloud based Querying and Transformation Engines
query and transform
stored in
Efficient Model Persistence Format
indexes
used to manage
requires
use for global queries
Collaborative Modelling Tools
Model Indexing Framework
requires
Scalable Model Persistence
requires
underpin
Scalable Collaborative Modelling
requires
Primitives and Patterns for Collaborative Modelling

Figure 2.5.: Scalability issues in MDE, from [31]

of this issue can be found in Section 2.2.1.

Model querying and transformation The ability to perform intensive and complex queries and transformations on large models with fast execution time is the second issue MDE faces. Efficient execution of *global queries* can be of importance when dealing with large collections of (possibly interconnected[8]) models. A global query on a collection of model files is one which requires multiple (commonly all of the) model files to be loaded in order to be computed. For example a query asking whether a particular model is referenced by other models needs all other model files to be loaded as well as that of the

[8] some modeling technologies such as EMF support the concept of persisting interconnected model fragments, whereby a model file can reference elements in other model files instead of just elements found in itself

model itself, in order to be calculated. A detailed description of this issue can be found alongside the various technologies reviewed in Section 2.2.

Collaborative work Multiple developers collaborating (either online or offline), each with their own part of the model, by querying or editing it in a consistent and synchronized manner, is another aspect of scalability on large projects. A detailed description of this issue (including the use of model comparison) can be found in Section 2.2.2.

Creation/exploration/visualization of large models Managing large models starts from model creation; scalable modeling languages need to be developed that allow building and exploring large models incrementally. Furthermore the ability to visualize such large models efficiently becomes important when various stakeholders, who may not be capable of using other techniques such as querying, need to work with the models.

This research primarily focuses on the first three issues mentioned here: model persistence, model querying and collaborative work; as well as touching upon efficient model comparison, in order to tackle its research objectives presented in Section 3.3.

2.1.3. Running Example

In order to provide context for presenting this work, we will use the metamodel seen in Figure 2.6 as a running example. This metamodel is a simplified version of BPMN[9]. BPMN is a modeling language used to define business processes in a standardized manner, and is widely adopted.

This metamodel was chosen for the following reasons:

- To avoid name clashes between concepts in different meta-levels (in contrast to UML, for example).
- BPMN provides a high-level set of concepts (Event, Task, Flow, etc.), understandable by a wide audience, while lending itself nicely to the field of MDE as it can be used to realistically model a software system.
- A simplified version allows for readability while still using all of the interesting concepts found in Ecore, such as inheritance, containment, opposite references etc.,

[9] this is a simplified version of the OMG-standardized business process model and notation (BPMN) specification: `http://www.bpmn.org/`

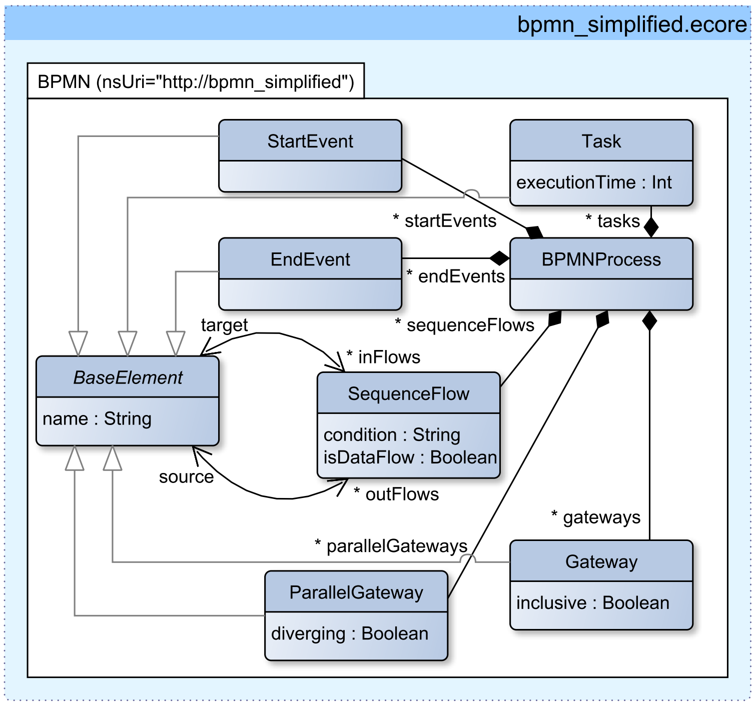


Figure 2.6.: Running example – (simplified) BPMN metamodel

so any discussion on this metamodel is also applicable to any other EMF-based metamodel.

This metamodel (from now on referred to as the "BPMN metamodel"), defines that models that conform to it can contain the following elements:

- *Task*s. Each *Task* represents a timed activity the system has to perform and has an attribute denoting this execution time needed to complete. This simplified metamodel assumes all other transitions, decisions and states are resolved instantaneously (a time of 0 is associated with visiting any of them).
- *StartEvent*s. A *StartEvent* denotes a possible starting point for the business process. There can be more than one such elements in any process, for example one for an administrator starting the process and one for an unregistered user starting it.
- *EndEvent*s. An *EndEvent* denotes a possible termination point for the process. There can be more than one such elements in any process, for example one for

normal and one for abnormal termination.

- *SequenceFlow*s. A *SequenceFlow* is a link between two *BaseElement*s (ie: any concrete subclass of *BaseElement*). It has references to its source element and its target element (as well as their opposite references); it has a flag denoting whether it is a data-flow element (data is expected to travel between the elements it is connecting); finally it may have a *condition* attribute denoting which conditions need to be satisfied for this flow to be followed (in this simplified example, such conditions are written in plain English).

- *Gateway*s. A *Gateway* breaks the process flow into one or more alternative *SequenceFlow*s, depending on whether the *condition* attribute of the relevant flow is satisfied, as well as whether the gateway is inclusive or exclusive (based on the value of the *inclusive* attribute flag).

- *ParallelGateway*s. A *ParallelGateway* breaks the process flow into all of its possible outcomes, regardless of conditions. If its *diverging* attribute is *False* then it is expected to collect multiple parallel flows into a single one instead of breaking up a single flow into multiple.

A small model conforming to this metamodel is seen in Figure 2.7, containing a BPMN process for processing a loan application (from now on referred to as the "BPMN Loan").

This process starts by recording the relevant application information and checking it. If the check fails the application is immediately rejected and the process ends, otherwise it continues to a loan study. If the loan study approves the loan, then the applicant is given the agreed upon loan, otherwise they are informed about the rejection details; in both cases the process then ends. Figure 2.8 shows the same model in its full structural representation (even for such a small example, we can see that it starts getting cluttered and barely readable).

In the interest of clarity/readability this example has been kept small (using a very small modified subset of BPMN), but a more complex example using large models can be found in Chapter 5.2.1.

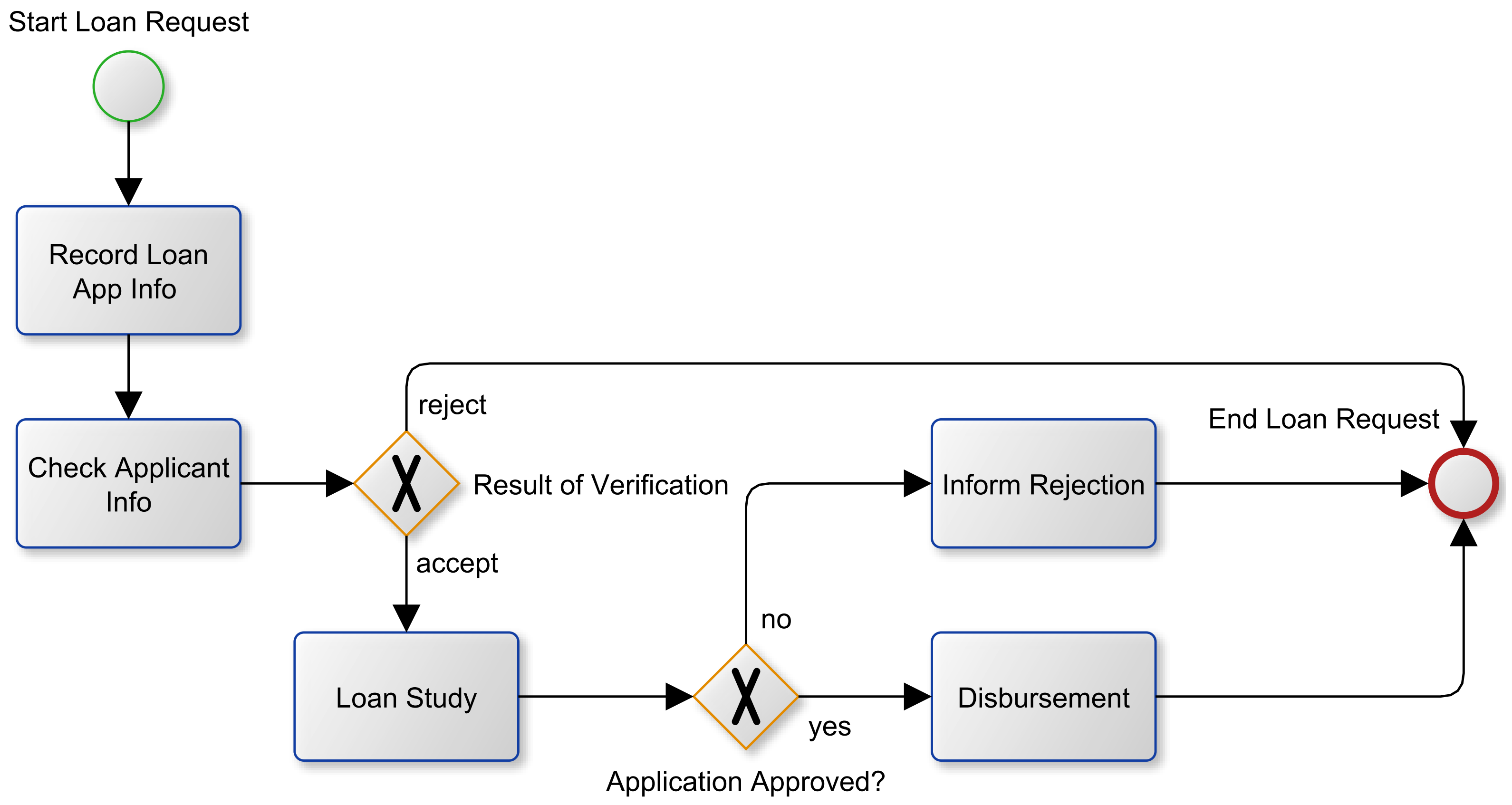


Figure 2.7.: Running example – BPMN model diagram

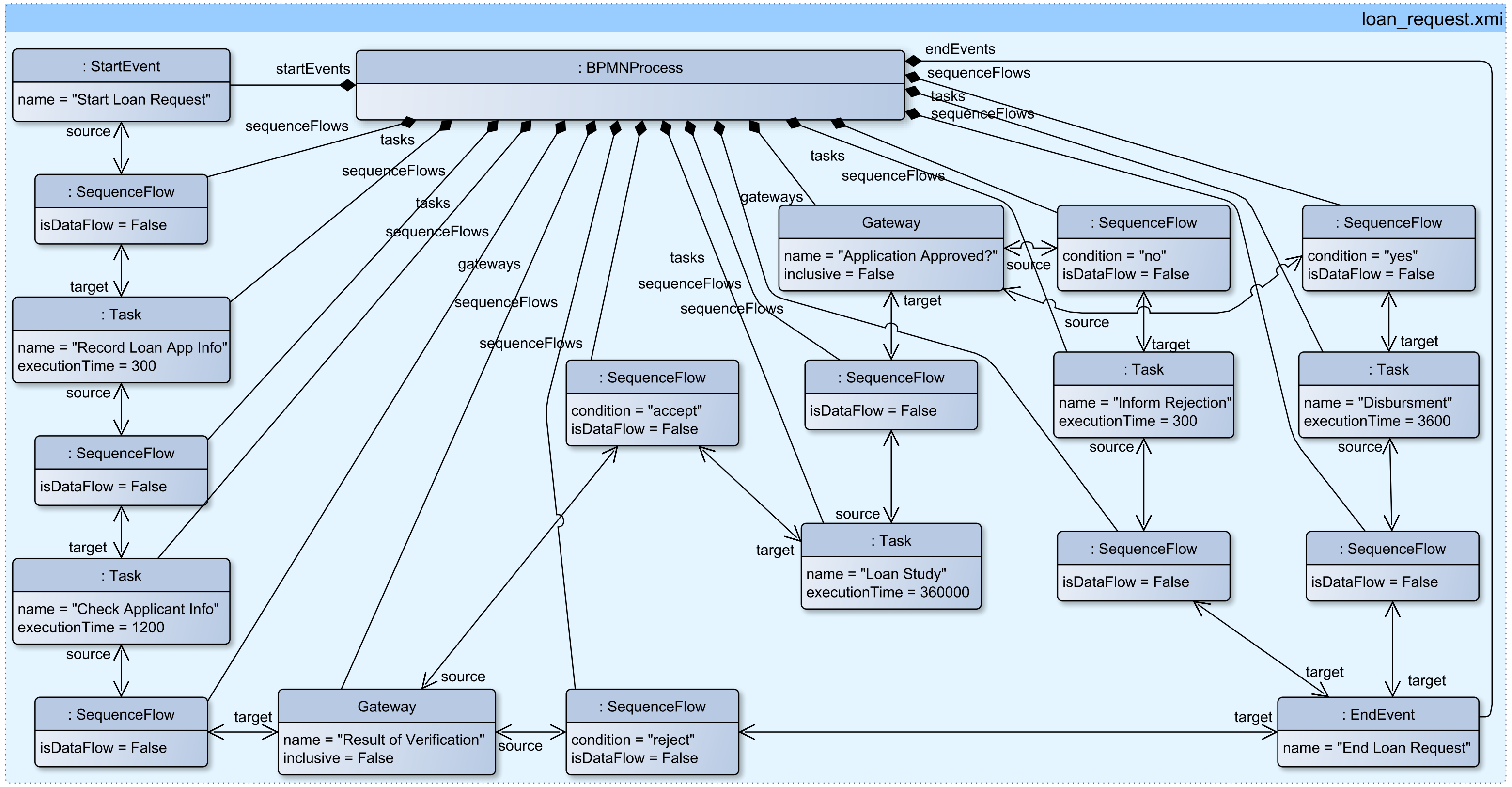


Figure 2.8.: Running example – BPMN model

2.2. Model Persistence and Versioning

This section reviews some of the most prominent tools and techniques used today for persisting and versioning large models. Whilst each tool offers different functionality, the commonality found in all is that they either use XMI as a means of storing models, or they replace it for their own proprietary technology (such as using custom database or textual persistence). This paradigm of either using XMI or offering a wholesale replacement brings up the question of whether an approach can lie somewhere in the middle. Chapter 3 presents such an approach, aimed at leveraging the current use of XMI whilst still offering a scalable way to query such models.

2.2.1. Model Persistence Formats

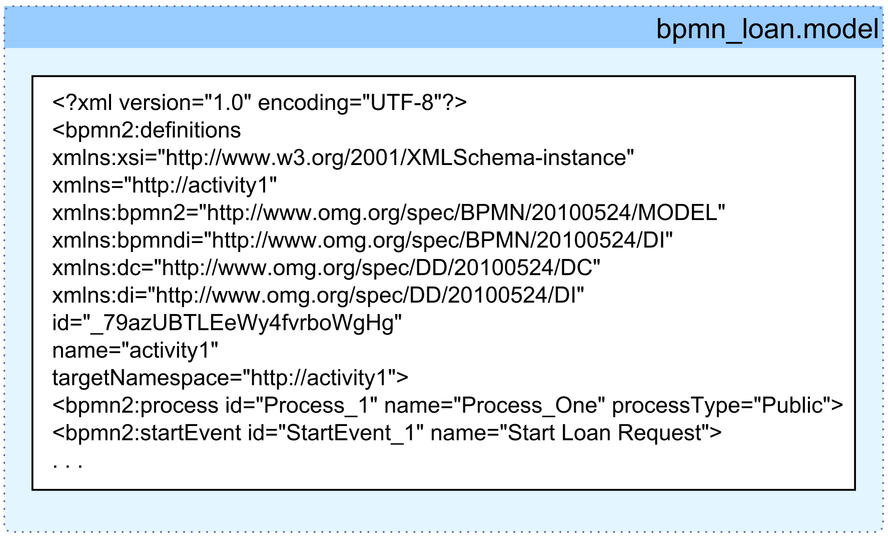
Several widely-used modeling frameworks such as EMF serialize their models in XML form, often using the XMI standard. This format will be used as a baseline for comparison with various other alternatives used today.

2.2.1.1. XML Metadata Interchange (XMI)

XMI is an extension of XML-based documents with each model element being mapped to an XML element and having a unique identifier. Model element containment is represented as XML element containment and model references refer to the unique model element identifiers in the document. Many of today's MDE tools either directly support loading models form XMI or at least offer an import/export functionality from XMI to their own persistence format. This allows for the development of artefacts with a certain degree of confidence that vendor lock-in is avoided and that porting the model between various tools is feasible. While this format does partially aid in collaborative work (as models can be fragmented into multiple interconnected files), several major issues remain such as the need to load multiple (or all of) the XMI files into memory, should a global query on the model be needed (one which requires multiple – commonly all – of the model fragment files to be loaded into memory in order to be calculated), or that any change on a model element may affect elements in other XMI files (for example the deletion of an element which is a container of one in another XMI file). Furthermore, as XMI is an XML-based format, models stored in single XMI files cannot be partially loaded and as such, loading an XMI-based model requires reading this entire document using a SAX parser, and converting it into an in-memory object graph that conforms to the respective metamodel. As such, XMI scales poorly for large models both in terms of the time needed for upfront parsing and in terms of the resources needed to maintain the entire object graph in memory.

Finally, as XML is a text-based file format, it lends itself nicely to classical file-based VCS; many such systems will only have to store the changed part of the file each time it is updated and can hence perform efficient (albeit non-semantic) comparison between the versions, or will offer the means to quickly compute it. Many other persistence formats are binary in nature and cannot be directly compared by the VCS (Section 2.2.2.1).

In the BPMN Loan example, the BPMN metamodel would be commonly stored in an XMI file (such as one named bpmn.ecore – due to the fact that in EMF metamodels are models too, this file will have the same structure as a model instance file) and the model would be stored in another XMI file (such as one named bpmn_loan.model, as shown in Figure 2.9).


bpmn_loan.model

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn2:definitions
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://activity1"
xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
id="_79azUBTLEeWy4fvrboWgHg"
name="activity1"
targetNamespace="http://activity1">
<bpmn2:process id="Process_1" name="Process_One" processType="Public">
<bpmn2:startEvent id="StartEvent_1" name="Start Loan Request">
. . .
```

Figure 2.9.: XMI persistence of the BPMN Loan model (partial)

One approach in attempting to improve the handling the persistence of large models is to offer high-performance alternatives to XMI, for storing these models. Such technologies use a variety of back-end persistence mechanisms to achieve this:

2.2.1.2. Relational Database Persistence – Teneo/Hibernate

Teneo-Hibernate [32] stores EMF models in a relational database. Databases such as MySQL and HSQLDB are supported. In this approach, an Ecore metamodel is used to derive a relational schema as well as an object-oriented API that hides the underlying database and enables developers to interact with models that conform to the Ecore metamodel at a high level of abstraction, using Hibernate queries (Figure 2.10).

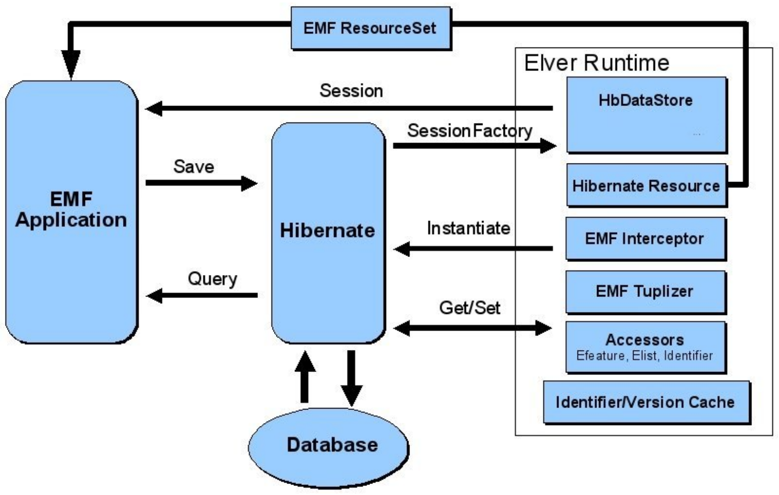


Figure 2.10.: The Teneo Runtime Layer, from [32]

This process can be automated or customized by the use of Java Persistence API (JPA) *Annotations*[10] that allow for custom schemas to be produced (which can produce custom naming of tables or specify the inheritance mapping, for example).

Model-Relational Mapping [32] The default mapping persists model elements by creating one table for each *EClass* hierarchy (vertical mapping strategy). Hence instances of that *EClass* as well as instances of all of its subclasses are stored in that table; this can be changed to having each class in its own table (using JPA) if deemed appropriate (horizontal mapping strategy). Multiple inheritance is handled seamlessly, by automatically choosing one superclass as the one used for insertion (this can be customized through JPA if necessary). *EAttribute*s are stored as columns in the table and *EReference*s as foreign keys. Ordered 1:n relations are handled by Hibernate lists (which require an additional column with the list order to be kept as well). Teneo-Hibernate also supports

[10] `http://www.oracle.com/technetwork/middleware/ias/toplink-jpa-annotations-096251.html`

unordered 1:n relations, which are specified by JPA and eliminate the need for the extra column, but will also not guarantee consistency in the return order of reference values, as expected. Ordered n:m relations are handled by producing two table joins, one for each side of the relation (as there is no way to guarantee the ordering of such relations with a single table join). Unordered n:m relations are persisted using a single table join by using the appropriate JPA annotation. "Contains" relations are supported as columns in the "contained" class table and store the relevant information about its container; These relationships are stored in parallel to 1:n relations due to lack of knowledge of containment state upon generation of the database schema (as one class can be contained by multiple classes in the metamodel but in the instance level you can only have a single container for an object). Finally Teneo supports an alternative Entity Attribute Value Mapping (EVM) that uses only two tables (object / values), which alleviates the need for changing the database schema every time the metamodel changes. Furthermore it supports dynamic-EMF but is unsuitable for large models as querying (if EVM is used) and maintenance[11] quickly becomes very inefficient.

Model-relational mapping approaches eliminate the initial overhead of loading the entire model in memory by providing support for partial and on-demand loading of subsets of model elements as demonstrated by experiments performed and described in the sequel. However, due to the nature of relational databases (object-relational impedance mismatch), such approaches, while better than XMI, are still largely ineffective. As models often have more references (to other model elements) than elements, complex queries can end up requiring multiple expensive table joins to be executed (every time such a reference to another element needs to be navigated) and hence do not scale well for large models, as demonstrated in Section 5.3.1.2. Even though Teneo-Hibernate (by default) tries to minimize the number of tables generated, by having all the subclasses of an *EClass* stored in the same table as the *EClass* itself (resulting in a fraction of the tables otherwise required if all *EClass*es made their own table of *EObject*s), the fact that the database itself is made up of sparsely populated data (as *EAttribute*s of all subclasses need to be stored in the schema, even if they are only effectively used by a small percentage of actual objects stored) results in increased insertion and query time, as demonstrated by various works such as [33, 34].

[11] `http://wiki.eclipse.org/Teneo/Hibernate/Dynamic_EMF_Tutorial`

As with most database persistence formats, relational databases persist data in the form of binary files. As discussed in Section 2.2.2.1, versioning binary files in file-based version control systems is discouraged and will create large overheads for storage and comparison of versions.

In the BPMN Loan example, the BPMN metamodel would be mapped to the back-end (database) store as a schema, and then the model would be inserted into the same back-end as fields in tables; a subset of the resulting database would look as in Figure 2.11.

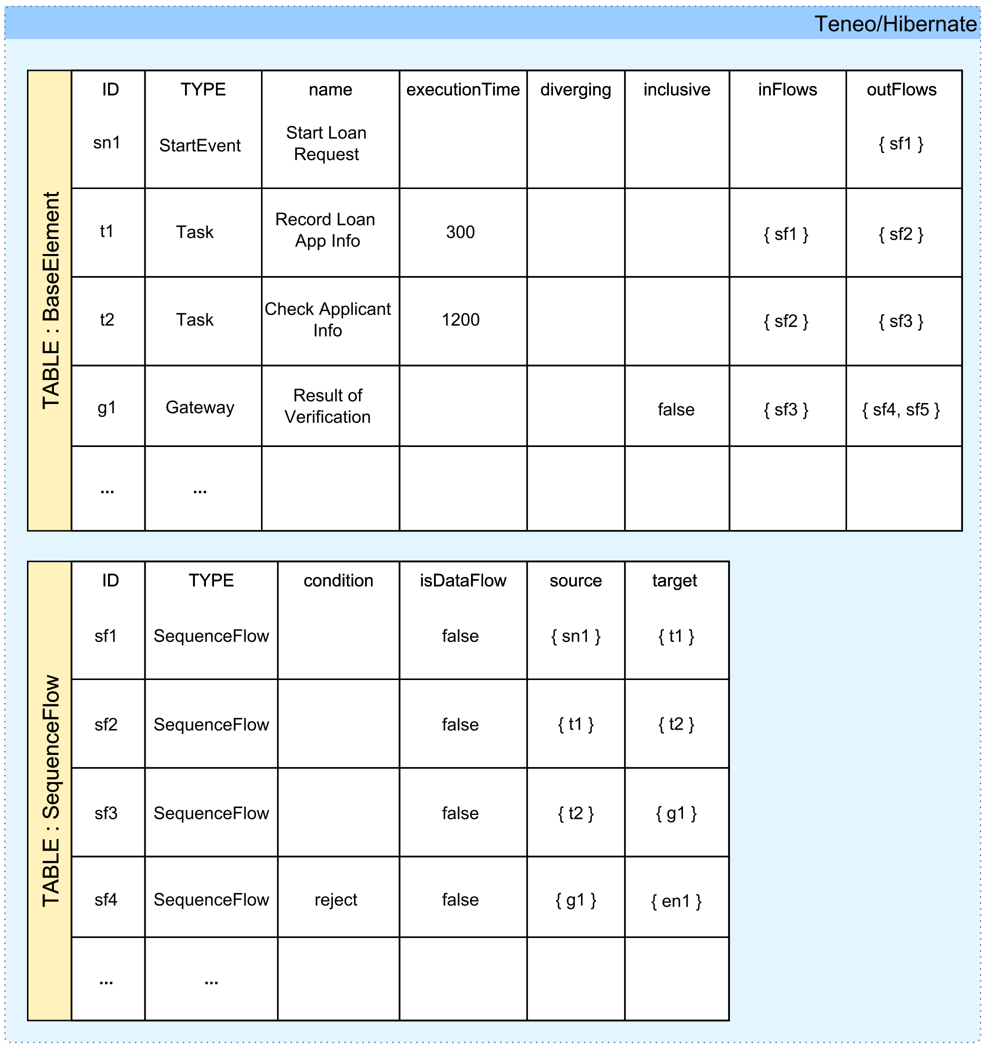

Teneo/Hibernate

TABLE : BaseElement

ID	TYPE	name	executionTime	diverging	inclusive	inFlows	outFlows
sn1	StartEvent	Start Loan Request					{ sf1 }
t1	Task	Record Loan App Info	300			{ sf1 }	{ sf2 }
t2	Task	Check Applicant Info	1200			{ sf2 }	{ sf3 }
g1	Gateway	Result of Verification			false	{ sf3 }	{ sf4, sf5 }
...	...						

TABLE : SequenceFlow

ID	TYPE	condition	isDataFlow	source	target
sf1	SequenceFlow		false	{ sn1 }	{ t1 }
sf2	SequenceFlow		false	{ t1 }	{ t2 }
sf3	SequenceFlow		false	{ t2 }	{ g1 }
sf4	SequenceFlow	reject	false	{ g1 }	{ en1 }
...	...				

Figure 2.11.: Relational database persistence of the BPMN Loan model (partial)

2.2.1.3. Document-based Persistence – Morsa

Morsa [35] is a prototype that attempts to address the issue of scalable model persistence using a document store NoSQL database (MongoDB) to store EMF models as collections of documents.

NoSQL Document-based stores As the reader cannot be assumed to be familiar with document stores (in contrast to relational databases for example), a brief background discussion is provided here. Document databases (seen in Figure 2.12) consist of a set of documents (possibly nested), each of which contains fields of data serialized in a standard format like XML or JSON[12]. They allow for data to be structured in a schema-less way as heterogeneous collections of such documents. Popular examples are MongoDB [36], CouchDB [37] and OrientDB [38].

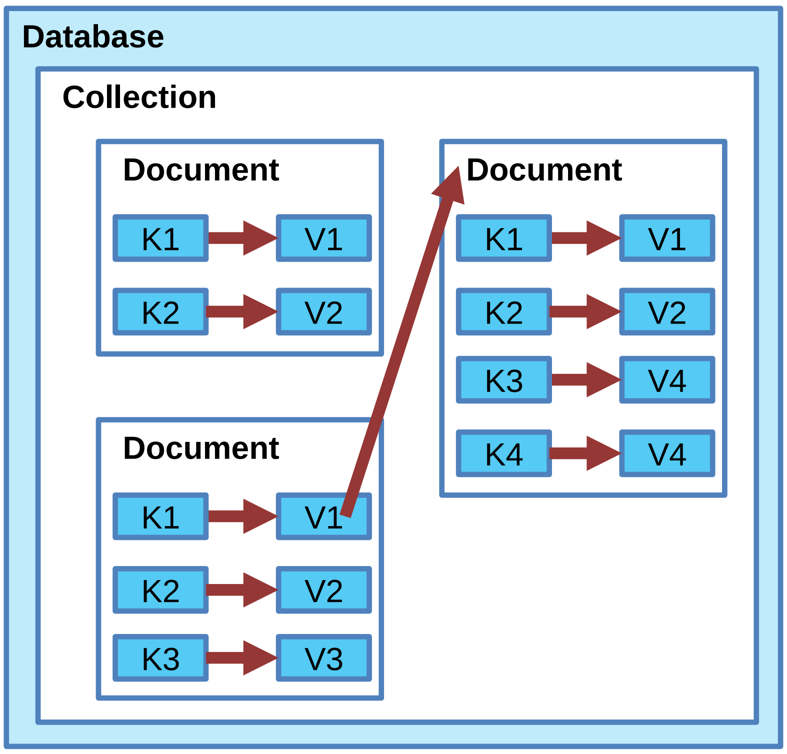


Figure 2.12.: Conceptual organization of data stored in a document store

[12] `http://json.org/`

Model-MongoDB Mapping Morsa stores one model element per document, whereby their attributes are stored as a key-value pair alongside persistence metadata of the model element (such as reference to its metaclass). Metamodel elements are stored in a similar fashion to model elements and are also represented as entries in an index document that maps each model or metamodel URI (the unique identifier of a model or metamodel element) in the store to an array of references to the documents that represent its root objects. An example of this architecture is displayed in Figure 2.13.

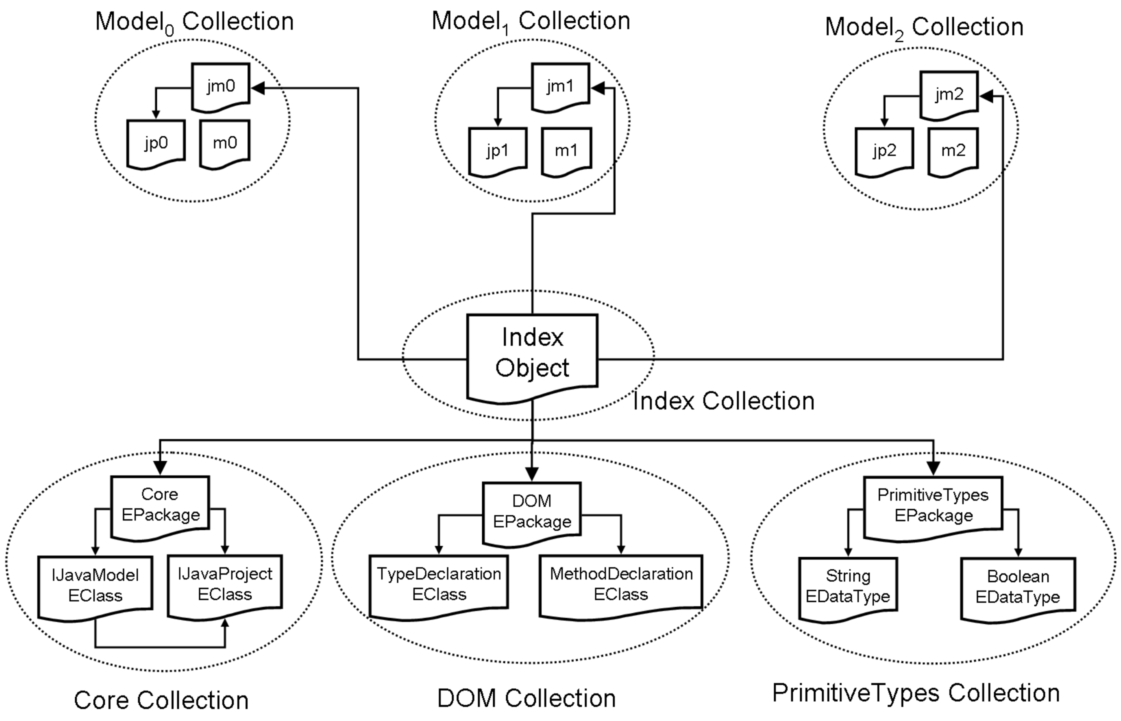


Figure 2.13.: Persistence back-end structure excerpt for Morsa, from [35]

Morsa uses a load on demand mechanism that relies on an object cache that holds loaded model objects. This cache is managed by a configurable cache replacement policy that chooses which objects must be unloaded from the client memory (should the cache be deemed full by the current configuration).

Similar to storing models in a relational database, storing them in a document store will create the same problems when using a file-based version control system to version them as the resulting binary database files will require inefficient comparison and storage in the VCS.

In the BPMN Loan example, the BPMN metamodel would be inserted into the MongoDB store, and then the model would be also inserted. The resulting database would look as in Figure 2.14.

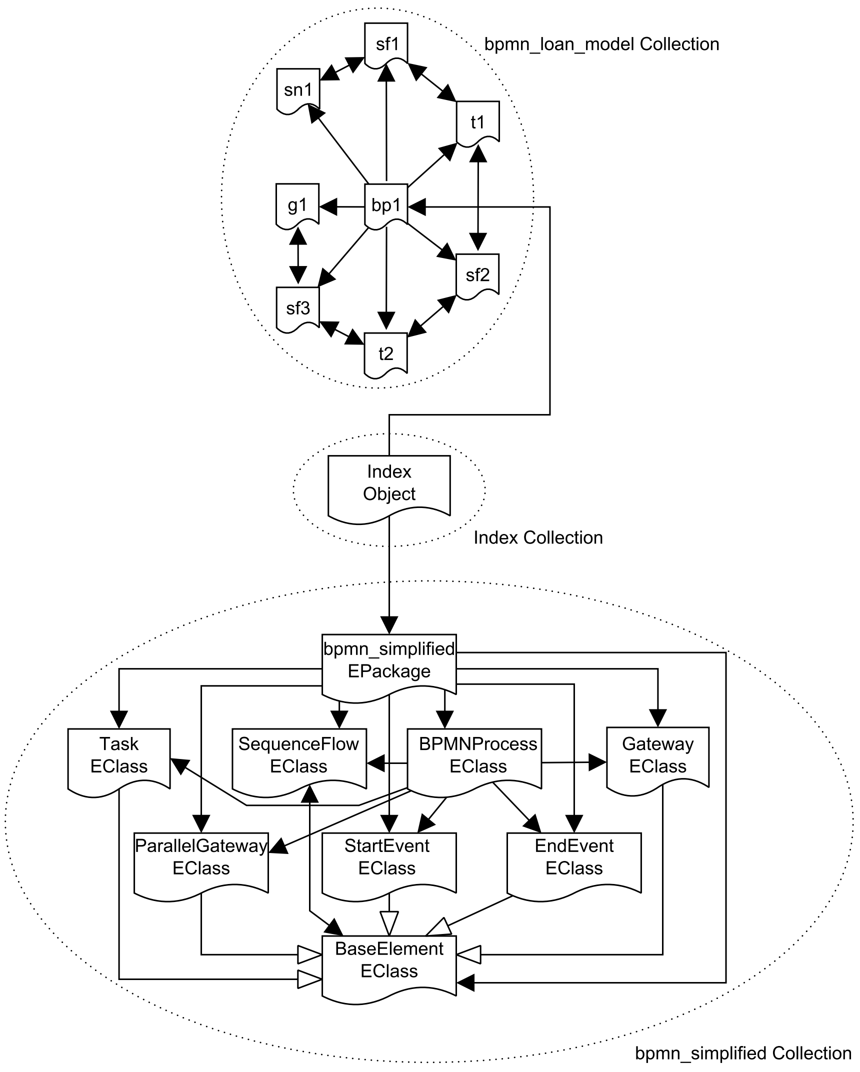


Figure 2.14.: NoSQL database persistence of the BPMN Loan model in Morsa (partial)

2.2.1.4. Document-based Persistence – MongoEMF

MongoEMF[13] also allows EMF models to be stored in MongoDB. This allows the use of EMF's API while offering possible scalability benefits by storing the models in a NoSQL store.

It extends the EMF *Resource* interface by using custom URIs to identify them. Such URIs are of the form: "`mongodb://host[:port]/database/collection/{id}`". Individual element ids can be either generated by the tool (as a default) or can be manually specified when a new object is created. If generated ids are used, the bulk insertion of objects can be performed (by using a single save call on the resource). Creating, removing or updating *EObjects* is done by saving the relevant resource with the objects in question. Retrieving objects can either be done by identifier or by querying the store. Querying can either be a native MongoDB query or a query of the proprietary format provided by MongoEMF (named simple query format).

Model-MongoDB Mapping MongoEMF offers two options for using MongoDB collections of objects; a collection of objects can either be stored within a parent object or as top level documents in a MongoDB collection. When an object contained by a parent is operated upon (modified), the entire collection of objects contained by the parent is also consequently operated upon. On the other hand if objects are stored separately in a MongoDB collection, the client will have to handle them individually.

Furthermore, EMF references are treated depending on their type. Non-containment references are always persisted as proxies to the object in question. For references to objects in other resources, the object in question will have to be created before the object referencing it can be created (as the id of the referenced object will have to exist for the proxy to be valid). Containment references create nested objects in MongoDB unless the target object is in its own resource (different to the one of the container object) and also containment proxies is set to true in the Generator model.

The default handling of *EDataType*s is to convert the ones which cannot be directly mapped to MongoDB's types (such as Boolean, Integer, String etc.) into Strings (based on the relevant conversion functions in the model). If one wishes to handle such data types in a different way, for example saving one as an Integer (with some custom meaning), an instance of *IValueConverter* can define this mapping for MongoEMF.

[13] `https://github.com/BryanHunt/mongo-emf/wiki`

Finally, various options are available for loading a MongoEMF resource, the most important ones summarized below:

- OPTION_PROXY_ATTRIBUTES – This Boolean option allows for cross-document proxy references to have their attributes populated automatically (without resolving the actual proxy), when set to true.

- OPTION_QUERY_CURSOR – This Boolean option allows for a *MongoCursor* to be returned by a query, instead of a default *Result* with proxies to its contents, when set to true. This permits the subsequent iteration with creation on demand only of the necessary objects.

- OPTION_SERIALIZE_DEFAULT_ATTRIBUTE_VALUES – This Boolean option allows for the persistence of default values of attributes, when set to true. This aids in querying by attribute value as the default EMF behavior is to not store default values.

- OPTION_USE_ID_ATTRIBUTE_AS_PRIMARY_KEY – This Boolean option allows for the automatic use of the *ID* attribute as the MongoDB id of the object, when set to true. This only actually sets the ids of new objects which have no id in their URI and have an *ID* attribute.

In the BPMN Loan example, the Loan model would be inserted into the MongoDB store when the "save" method is called on its resource, and any relevant metamodels the resource used would be inserted if they did not exist already in the store (as these metamodels are already loaded by EMF in order for the resource to be loaded). While the exact structure of the resulting store has not been published, it will be similar to the one presented in Figure 2.14.

2.2.1.5. Graph-based Persistence – NeoEMF (Graph)

NeoEMF[14] (formerly known as Neo4EMF and Kyanos) comprises two tools developed by Inria[15]; it uses graph databases as its back end.

[14] `https://raweb.inria.fr/rapportsactivite/RA2014/atlanmod/uid49.html`
[15] `http://www.inria.fr/`

NoSQL Graph-based stores Graph Databases (seen in Figure 2.15) consist of a set of graph nodes linked together by edges (hence providing index-free adjacency of nodes). Each node contains fields of data and querying the store commonly uses efficient graph-traversal algorithms to achieve performance. As such, these databases are optimized for traversal of highly interconnected data. Examples of such stores are Neo4J [39], InfiniteGraph [40] and the graph layer of OrientDB [38].

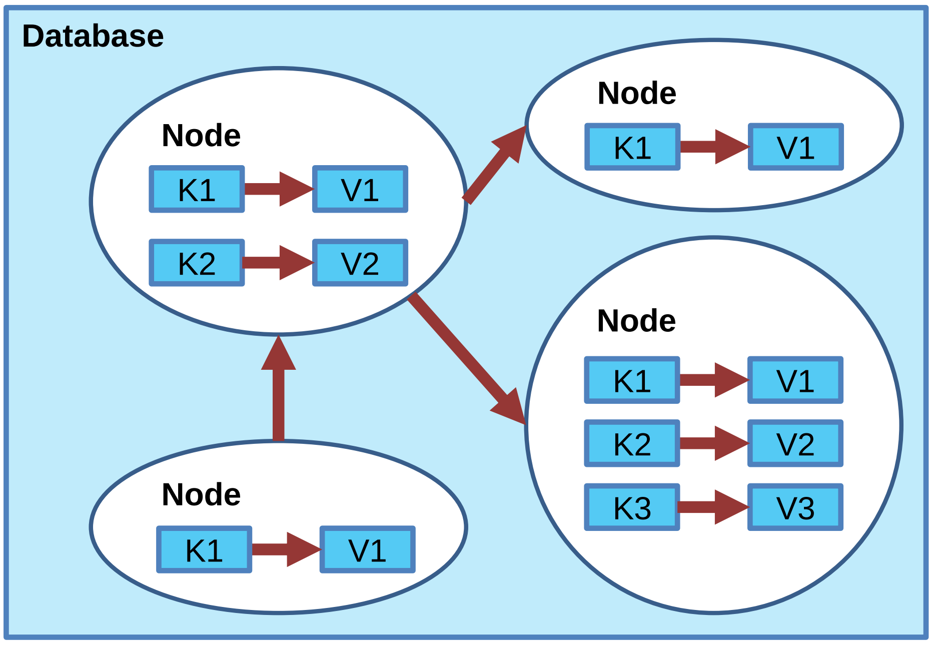


Figure 2.15.: Conceptual organization of data stored in a graph store

NeoEMF/Graph offers "a backend-agnostic persistence solution for big, complex and highly interconnected EMF models"[14]. It uses the Blueprints API[16] to connect to a variety of NoSQL graph databases and hence leverages lazy loading and database caching. Whilst blueprints offers the convenience of abstracting from the actual graph-based backend, it has been shown to cause a loss in performance in some cases [41], when compared to using the raw database API. NeoEMF also allows for custom caches to be added according to specific strategies defined in a decorator pattern and also offers a "dirty save" mechanism to handle the safe splitting of large transactions into smaller ones (while retaining the ability to revert to a consistent state).

[16] `https://github.com/tinkerpop/blueprints/`

Model-Neo4J Mapping The following strategy is used to persist an in-memory EMF model resource in Neo4J [42]:

- Model elements are stored as Neo4J nodes with a special node denoting "root" elements which reference all other elements in the model.
- Element attributes are stored as node properties in the corresponding model element node.
- Metamodel elements are stored as nodes. Such nodes only contain two properties, one for their name and one for the unique identifier (nsURI) of their metamodel.
- Conformance relationships are stored as outgoing Neo4J relationships with type *INSTANCE_OF* pointing to the node representing the relevant metamodel type of the model element.
- References between model elements are stored as relationships between the elements, using a specific naming convention to avoid possible conflicts with other relationships (such as the conformance relationships described above).

An example of this architecture can be seen in Figure 2.16.

In the BPMN Loan example, the Loan model would be inserted into the Neo4J store when the "save" method is called on its resource, and any relevant metamodels the resource used would be inserted if they did not exist already in the store (as these metamodels are already loaded by EMF in order for the resource to be loaded). The resulting database would look as in Figure 2.17.

2.2.1.6. Key-value-based Persistence – NeoEMF (Map)

NeoEMF/Map[17] is an alternative back-end layer for NeoEMF which uses a MapDB[18] NoSQL key-value store. The aim is to work with limited memory and allow for the common model management operations to be performed on large models stored in a NeoEMF/Map store. Similarly to NeoEMF/Graph it allows for custom caching to be added on demand and is extensible so other similar back-ends can be used instead.

[17] `http://www.emn.fr/z-info/atlanmod/index.php/NeoEMF/Map`
[18] `https://github.com/jankotek/MapDB`

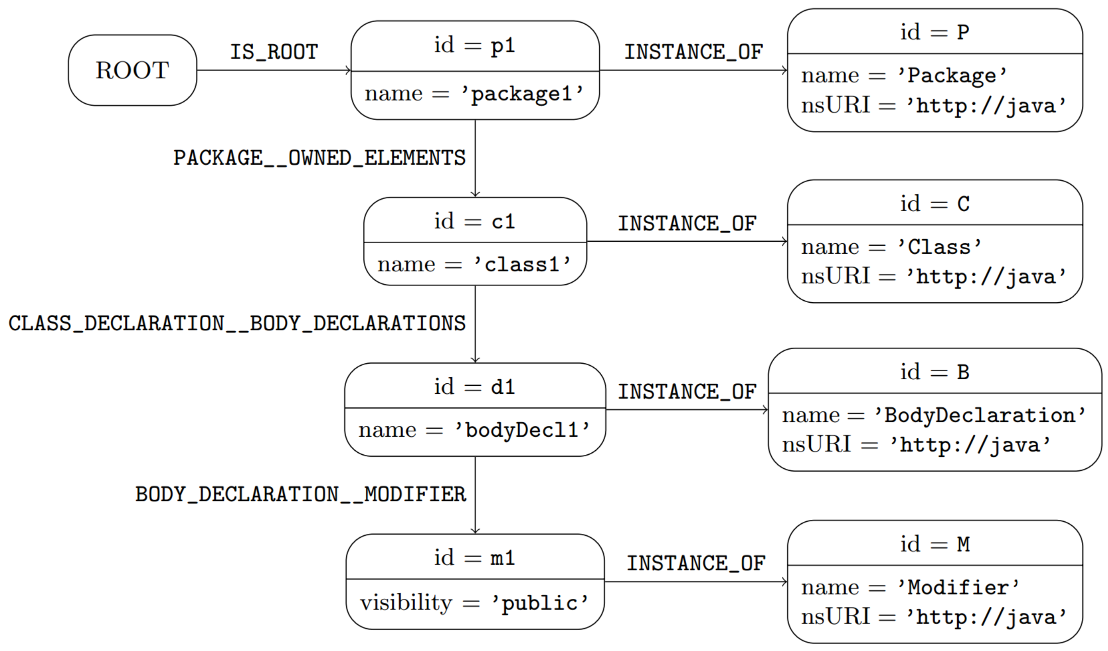


Figure 2.16.: Persistence back-end structure excerpt for NeoEMF Graph, from [42]

NoSQL Key-value Stores Key-value Stores (seen in Figure 2.18) consist of keys and their corresponding values, which allows for data to be stored in a schema-less way. Such stores claim that this allows for search of millions of values in a fraction of the time needed by conventional storage. Inspired by databases such as Amazon's Dynamo [43], they are tailored for handling terabytes of distributed key-value data.

Model-MapDB Mapping The following strategy [44] is used to persist an in-memory EMF model resource in MapDB. Firstly a unique identifier is assigned to every model element which allows for the subsequent decomposition of the entire model into a collection of key-value pairs. NeoEMF/Map uses three maps to store this information:

- Property map: This is a map containing the slots (values) for each feature of the model elements. Each row in this map has the following syntax:

  $< (id, name), v >$

  Where the key is made of the pair comprising the unique identifier of the model element and the name of the feature, and the value is made up of either the literal containing the value of the feature (either the identifier of another element for a

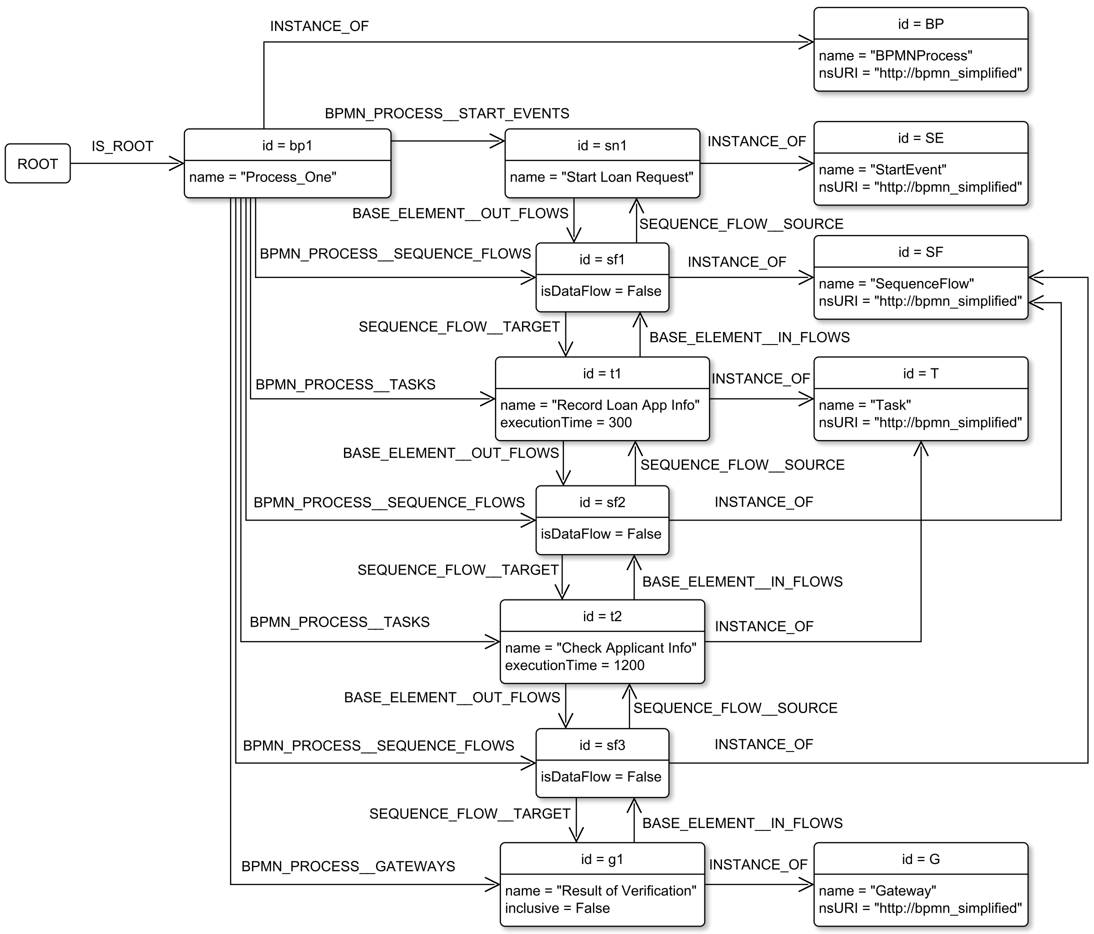


Figure 2.17.: NoSQL database persistence of the BPMN Loan model in NeoEMF Graph (partial)

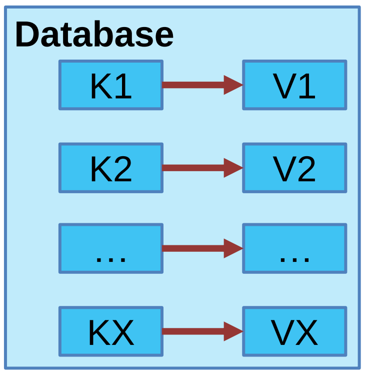


Figure 2.18.: Conceptual organization of data stored in a key-value store

reference or the primitive value for attributes) for single-valued or of an array of such literals for multi-valued features.

- Type map: This is a map containing information about metamodel types. Each row in this map has the following syntax:

  $< id, (nsURI, name) >$

  Where the key is the unique identifier of the model element in the store and the value is made up of named tuples containing various meta-information about its type, such as their name and the nsURI of the package they are contained in.

- Containment map: This is a map containing information about containments. Each row in this map has the following syntax:

  $< id, (cont\_id, name) >$

  Where the key is the unique identifier of the model element in the store and the value is made up of named tuples containing the unique identifier of the container object and the name of the feature (reference) that relates the container object with the child object.

An example of this architecture can be seen in Figure 2.19.

In the BPMN Loan example, the Loan model would be inserted into the MapDB store when the “save” method is called on its resource, and any relevant metamodels the resource used would be inserted if they did not exist already in the store (as these metamodels are already loaded by EMF in order for the resource to be loaded). The resulting database would look as in Figure 2.20.

2.2.2. Model Versioning

Models, like all other artefacts involved in the software development process need to be versioned in a systematic and disciplined manner. This section presents two popular approaches for versioning in MDE: using file-based version control tools, and using model-specific version control tools.

Table 1: Property map

KEY	VALUE
⟨'ROOT', 'eContents' ⟩	{ 'p1' }
⟨'p1', 'name' ⟩	'package1'
⟨'p1', 'ownedElement' ⟩	{ 'c1' }
⟨'c1', 'name' ⟩	'class1'
⟨'c1', 'bodyDeclarations' ⟩	{ 'b1', 'b2' }
⟨'b1', 'name' ⟩	'bodyDecl1'
⟨'b1', 'modifier' ⟩	'm1'
⟨'b2', 'name' ⟩	'bodyDecl2'
⟨'b2', 'modifier' ⟩	'm2'
⟨'m1', 'visibility' ⟩	'public'
⟨'m2', 'visibility' ⟩	'public'

Table 2: Type map

KEY	VALUE
'ROOT'	⟨nsUri='http://java', class='RootEObject'⟩
'p1'	⟨nsUri='http://java', class='Package'⟩
'c1'	⟨nsUri='http://java', class='ClassDeclaration'⟩
'b1'	⟨nsUri='http://java', class='BodyDeclaration'⟩
'b2'	⟨nsUri='http://java', class='BodyDeclaration'⟩
'm1'	⟨nsUri='http://java', class='Modifier'⟩
'm2'	⟨nsUri='http://java', class='Modifier'⟩

Table 3: Containment map

KEY	VALUE
'p1'	⟨container='ROOT', featureName='eContents'⟩
'c1'	⟨container='p1', featureName='ownedElements'⟩
'b1'	⟨container='c1', featureName='bodyDeclarations'⟩
'b2'	⟨container='c1', featureName='bodyDeclarations'⟩
'm1'	⟨container='b1', featureName='modifiers'⟩
'm2'	⟨container='b2', featureName='Mmodifiers'⟩

Figure 2.19.: Persistence back-end structure excerpt for NeoEMF Graph, from [44]

2.2.2.1. File-based Versioning

Version Control (also referred to as Revision Control) is the management of changes made to a set of files. These files can be textual or binary in nature, and is organized as a set of ordered (usually numbered) revisions. Such systems perform a line-by-line comparison of files to discover differences (deltas) between versions; as such, text-based files lend themselves nicely as small changes will be understood by the VCS and propagated accordingly (with only the small delta being stored); on the other hand binary files will most likely change in structure throughout, even for small changes, and will hence have to be fully compared and stored each time, a time and resource consuming operation that should be avoided if possible (it is widely understood that storing binary files in file-based VCS is discouraged). Applications which offer this service can be both stand-alone in nature (such as Subversion [45]) or embedded in programs like as word-

Property Map	
Key	**Value**
⟨‘ROOT’, ‘eContents’⟩	{ ‘bp1’ }
⟨‘bp1’, ‘name’⟩	‘Process_One’
⟨‘bp1’, ‘startEvents’⟩	{ ‘sn1’ }
⟨‘bp1’, ‘sequenceFlows’⟩	{ ‘sf1’, ‘sf2’, ‘sf3’ }
⟨‘bp1’, ‘tasks’⟩	{ ‘t1’, ‘t2’ }
⟨‘bp1’, ‘gateways’⟩	{ ‘g1’ }
⟨‘sn1’, ‘name’⟩	‘Start Loan Request’
⟨‘sn1’, ‘outFlows’⟩	{ ‘sf1’ }
⟨‘sf1’, ‘isDataFlow’⟩	False
⟨‘sf1’, ‘source’⟩	{ ‘sn1’ }
⟨‘sf1’, ‘target’⟩	{ ‘t1’ }
⟨‘t1’, ‘name’⟩	‘Record Loan App Info’
⟨‘t1’, ‘executionTime’⟩	300
⟨‘t1’, ‘inFlows’⟩	{ ‘sf1’ }
⟨‘t1’, ‘outFlows’⟩	{ ‘sf2’ }
⟨‘sf2’, ‘isDataFlow’⟩	False
⟨‘sf2’, ‘source’⟩	{ ‘t1’ }
⟨‘sf2’, ‘target’⟩	{ ‘t2’ }
...	...

Type Map	
Key	**Value**
‘ROOT’	⟨nsURI=‘http://bpmn_simplified’,class=‘RootEObject’⟩
‘bp1’	⟨nsURI=‘http://bpmn_simplified’,class=‘BPMNProcess’⟩
‘sn1’	⟨nsURI=‘http://bpmn_simplified’,class=‘StartEvent’⟩
‘sf1’	⟨nsURI=‘http://bpmn_simplified’,class=‘SequenceFlow’⟩
‘t1’	⟨nsURI=‘http://bpmn_simplified’,class=‘Task’⟩
‘sf2’	⟨nsURI=‘http://bpmn_simplified’,class=‘SequenceFlow’⟩
‘t2’	⟨nsURI=‘http://bpmn_simplified’,class=‘Task’⟩
‘sf3’	⟨nsURI=‘http://bpmn_simplified’,class=‘SequenceFlow’⟩
‘g1’	⟨nsURI=‘http://bpmn_simplified’,class=‘Gateway’⟩

Containment Map	
Key	**Value**
‘sn1’	⟨container=‘bp1’,featureName=‘startEvents’⟩
‘sf1’	⟨container=‘bp1’,featureName=‘sequenceFlows’⟩
‘t1’	⟨container=‘bp1’,featureName=‘tasks’⟩
‘sf2’	⟨container=‘bp1’,featureName=‘sequenceFlows’⟩
‘t2’	⟨container=‘bp1’,featureName=‘tasks’⟩
‘sf3’	⟨container=‘bp1’,featureName=‘sequenceFlows’⟩
‘g1’	⟨container=‘bp1’,featureName=‘gateways’⟩

Figure 2.20.: NoSQL database persistence of the BPMN Loan model in NeoEMF Map (partial)

processors (such as Microsoft Word) and wiki software packages (such as TWiki [46]). This section presents an overview of state-of-the-art, commonly used version control systems. It discusses the different types, their benefits and drawbacks and finally their limitations when used for collaborative model-driven development. There are two types of file-based version control systems:

Centralized Systems Centralized version control systems use a single server containing all versions of the files, to which clients connect and retrieve or update revisions. They vary according to the strategy they use to handle how new revisions are placed:

Pessimistic Locking Systems In this paradigm, also referred to as file locking systems, one user "checks out" a file and receives write access to it (other users still have read access). This file is now locked until the user "checks in" a revised version of the file or cancels their checkout of the file. Benefits of this approach include usability (being simple to understand and operate) as well as alleviation of merge conflicts (as there are no merges). Drawbacks include lack of practicality as a user locking the file for too long may result in unease to the others and could render the revision system ineffective as users may result to keeping local copies with their changes. Examples of such systems are the PVCS Version Manager [47] and ClearCase [48].

Optimistic Locking Systems In this paradigm, multiple users may edit the same file simultaneously. The first developer to "check in" any changes will update the version number and any further attempts will result in a conflict arising. These can be resolved in various ways, such as the system providing facilities to merge such files (usually limited to textual files) or, in the worst case, the user having to update their revision to the latest one (head revision) before being able to commit their changes. Benefits include concurrency (multiple users can write to the same file) and, for textual files, automated or semi-automated merging capabilities to resolve conflicts. Drawbacks include the need for constant updating to the latest revision to ensure that merge conflicts are limited. Examples of such systems are Subversion (SVN) [45], which also offers the optional locking of files (effectively being able to become a pessimistic locking system, on demand) and the Concurrent Versions System (CVS) [49].

Distributed Systems In distributed version control systems, users have local copies of the entire repository (files and history). This most notably allows for off-line version control (updating, checkpointing etc.) of files. Such systems vary in the way they handle updates:

Open Systems In open systems, updating is primarily done by merging revisions resulting in extensive branching of the available versions. Benefits include extensive support for branches, which allows for different paths to always be available for use and resulting in less (or no) need for resolving conflicts. Drawbacks include complexity in the use model as well as the possibility of branching so much that it becomes unusable. Examples of such systems are Git [50], Mercurial [51] and Bazaar [52].

Replicated Systems Replicated systems use similar principles as replicated databases. Version commits act in the same way as distributed database commits would, and create a single new revision. Benefits include having a similar use model to locking centralized systems, while having the benefits of distributed histories and off-line support. Drawbacks include lack of support for branching and introduction of merge conflicts. An example of such systems is Code Co-op [53].

File-based Versioning of Models Models persisted as files (both textual or binary in nature) can be versioned in file-based version control systems. Such systems operate on the file level; for example optimistic locking systems parse files line by line to detect changes in order to perform the merge part of their copy-modify-merge strategy [54–56]. This results in graph structures (like models) not having the ability of being handled at an appropriate level of abstraction but as plain text files instead. This limitation makes model-level change comparison between different versions of a model difficult, and model merging even more so, especially if small changes to the underlying model can cause a large change in the structure of the model file [57].

Model Fragmentation To avoid having to version control large monolithic model files, models are are physically separated into several smaller interconnected fragments to avoid transferring large files across the network. With the advent of modeling frameworks such as EMF which provide first-class support for inter-file references and robust model comparison tools such as *EMF Compare* [58] which are able to compare model fragments, this approach inherits all the advantages of working with robust and widely-adopted version control systems. However, compared to model-centric repository approaches discussed below, it demonstrates a significant shortcoming: the visibility of each developer is limited to the model fragments they have checked out in their local workspace. As such, using this approach makes it impossible to compute queries of global nature without fetching and inspecting all the model fragments from the remote repository every time. Obviously, as the number of model fragments in the repository grows, this approach becomes increasingly inefficient as the I/O overhead of loading these separate entities increases (with respect to the relative overhead of the local workspace).

In order to benefit from this technique, when a single monolithic model file is provided, a way to meaningfully split it into fragments would be required. Tools such as EMF-Fragments [59] can be used to automate this task.

In the BPMN Loan example, the files containing the Loan model and metamodel (whether they are in the form of standalone XMI files or in a database structure) would be versioned in the version control system (VCS). Every time the model needs to be accessed, the HEAD revision in the VCS will be retrieved and loaded by the relevant modeling tool used.

2.2.2.2. Model-based Versioning

An alternative to using file-based version control is to leverage model comparison technologies and live change events in order to enable model-element level comparison and updating on versioned models. Whether the models themselves are versioned in a classical file-based VCS or on a custom server using a proprietary storage technology, this approach allows for collaboration and resolution of conflicts on the model level instead of delegating it entirely to the client.

ModelCVS ModelCVS[19] [60] attempts to introduce interoperability between heterogeneous modeling tools. The presented prototype uses EMF and SVN as its proof of concept. Models (and metamodels) are stored in a repository that can allow different tools (working with different modeling languages) to check out a version of a model, edit it and commit. It aims at handling conflicts at the model element level as opposed to the usual file level that version control software usually works on (referred to as semantic versioning).

Semantic versioning is achieved by augmenting every commit of a new model version into the VCS repository with a computed change summary file. Every time a new commit is made, if the last revision in the repository is the direct ancestor of the incoming working copy then the commit can be directly accepted as there have been no changes between the commit and the previous version of the model. In every other case, a semantic comparison is performed by using the accumulated computed change summary file (created by combining all of the change summary files of all commits which were performed between the current commit and the latest commit checked out by the

[19] `http://www.modelcvs.org/versioning/index.html`

working copy) and the computed change summary file created by comparing the current commit model to its ancestor in the working copy. When all conflicts are resolved (either automatically or manually by the commiter) the tool now allows the version to be commited.

As the tool does not seem to have scalability as a primary concern, handling large models is done through the persistence default mechanism of each language (such as XMI for EMF models) and so will subsequently suffer from the same issues as the ones described in 2.2.1. Nevertheless, the architecture of this project may be suitable for gaining insight in integrating a version control system for use in this research project, as it does use a reliable version control system and integrates it with a repository of models.

In the BPMN Loan example, the files containing the Loan model and metamodel would be versioned in the SVN repository. Every time the model needs to be accessed, the HEAD revision will be retrieved and loaded by the relevant modeling tool used.

The Connected Data Objects Repository (CDO) Another approach is to store the models on a dedicated remote model repository [61]. CDO allows users to access models stored on various possible back-end stores that it can use to persist its repository. Its API is an extension of EMF's (explained in more detail below) and allows for a seamless use of a remote store for accessing and manipulating models. CDO supports multiple different back-ends such as relational databases (for example it can use teneo-hibernate described in Section 2.2.1.2 to store models into MySQL) and non-relational stores such as the MongoDB NoSQL store.

Object-Relational Mapping CDO[20] handles *EObject*s as *CDOObject*s that extend the *EObject* class by adding CDO-specific metadata. To store an EMF model on CDO, there are three main paths to pursue[21]: The first is to migrate a *Resource* (for example an *XMIResource*) to a *CDOResource* (by copying all its contents to a new *CDOResource*). The second is to use a *GenModel* (EMF generator model) to create *CDOObjectImpl* objects by migrating the *.genmodel* file using the CDO Model Migrator. The third is to use *DynamicCDOObjectImpl* that result from new dynamic model elements added to a

[20] `http://wiki.eclipse.org/CDO/Client`
[21] `http://wiki.eclipse.org/Preparing_EMF_Models_for_CDO`

CDO session's package registry. The model files used by CDO can be annotated (such as with JPA - *EAnnotations*) in order to provide a tool for customization in the storage of the model; such annotations are not directly needed by CDO and are only useful if the back-end store supports them (such as Teneo/Hibernate, for example).

From the client-side, the regular EMF API can be directly used after a connection (session) has been established, but for using advanced CDO-specific functionality (such as *CDOView* that allows queries directly to the CDO store, or *CDOTransaction* that allows for savepoints and rollbacks), additional dependencies to CDO have to be included. Furthermore, a built-in CDO User Interface (UI) is provided for accessing, manipulating and querying models stored in the repository. This client architecture is seen in Figure 2.21.

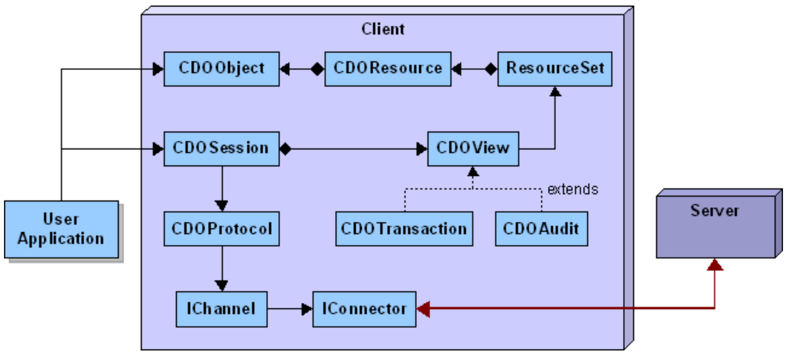


Figure 2.21.: CDO Client high-level architecture, from [61]

On the server-side, this repository allows any form of storage to be easily plugged in and is as scalable as the chosen back end but still has several limitations, such as ones regarding its version control.

Collaborative Development As CDO is a remote (on-line) store of models, it supports multi-user access and updating to models. The user can work with *CDOSession*(s). These can be either automatically or manually refreshed and will allow for consistency with the latest revision of the model. Elements can either be explicitly locked for updating or the process can be automated and delegated to the CDO framework. On conflict detection by the client (with passive updates or after a refresh, for example), the ability to commit fails, but the *CDOConflictResolver* interface allows for the ability to manually resolve specific conflicts by the user. On the server side the entire transaction is rejected if a conflict is found (this is mainly to handle network race conditions).

Version control CDO uses Audit Views to handle the ability to access historical data. This feature allows for an application to have a read-only view of the state of the model repository at a specific time in the past. As such, CDO has no way to allow parallel versions (branches) to be stored, should they be needed, as well as no effective way to handle major conflicts (for example after an off-line copy of the model tries to be synchronized with a largely changed model in the repository). Even though the versioning is model-specific and can detect conflicts on a model level, it is limited by the fact that past versions cannot be checked out as the latest version.

Example Back-end store for CDO An example back-end store is the Teneo/Hibernate back-end. As illustrated in Figure 2.22[22], this allows CDO to seamlessly leverage this technology to perform model storage, updates and queries.

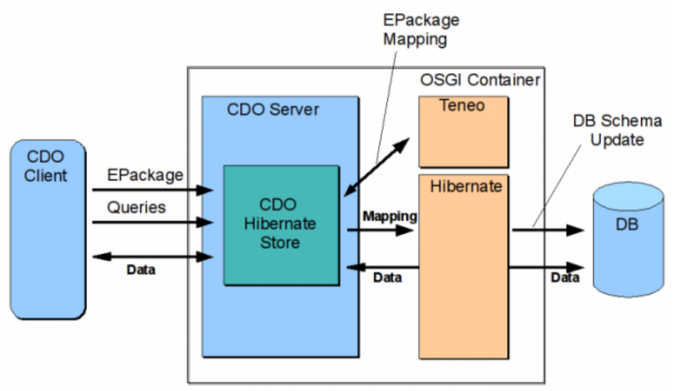


Figure 2.22.: The CDO-Teneo/Hibernate Runtime layer

As can be expected, this back-end will also have the same benefits and suffer from the same issues as the standalone Teneo/Hibernate store described above.

In the BPMN Loan example, the CDO back-end would contain the Loan model and metamodel. Refer to the Teneo/Hibernate example found in Section 2.2.1.2 for more details, as CDO wraps around this technology.

[22]http://wiki.eclipse.org/CDO/Hibernate_Store/Architecture

EMFStore EMFStore[23] [62] is a tool which uses a MongoEMF-based back-end (described in Section 2.2.1.4) to create a model repository for versioned models. It offers model-element level versioning so multiple developers can manipulate and resolve conflicts on EMF models at a model-element level.

Version Control The versioning system provided by EMFStore allows for model-element level history of versions and differencing of versions, branching and tagging. Hence any version in the server can be retrieved on demand as well as visualized in a history browser. EMFStore uses a change-based versioning strategy whereby historic information is kept as a collection of changes which have been made from the initial version to the current version; the types of changes can be seen in Figure 2.23[24]. If a commit has merge conflicts, a merge editor allows for them to be manually resolved one at a time.

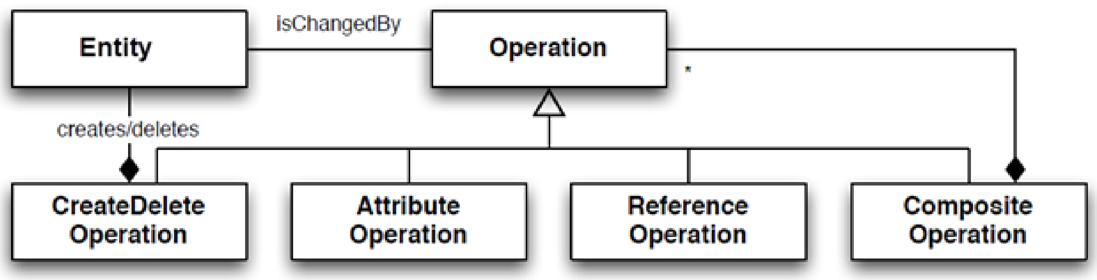


Figure 2.23.: Permitted changes for EMFStore

Finally, an extensible user interface is offered as a collection of Eclipse views[25]:

- Model explorer, containing the projects with their model contents
- Selected element, displaying the current model element and its properties (allowing editable properties to be changed)
- Model repositories, displaying the connected repositories and allowing relevant operations to be performed: commits will display a detailed model-element level set of changes to be commited; updates will display a similar set of model-level

[23] http://eclipsesource.com/blogs/2014/04/17/emfstore-1-2-0-released/
[24] http://eclipsesource.com/blogs/tutorials/emfstore-versioning-history-and-branching/
[25] http://eclipsesource.com/blogs/tutorials/getting-started-with-emfstore/

incoming changes; conflict resolution (merging) offers manual resolution for each model-level conflict.

- History browser, displaying the time-line of changes in the selected repository including details of each model-level change and the branching structure

In the BPMN Loan example, the MongoEMF MongoDB back-end would contain the Loan model and metamodel. The reader can refer to the MongoEMF example found in Section 2.2.1.4 for more details.

Commercial Tools Various commercial tools offer model repositories for a variety of modeling technologies:

Modelio – UML modeling environment Modelio[26] is an open-source modeling platform for the development of UML models. Various modules have been developed for this tool, some free and some commercial. One of the commercial modules is called the Modelio Teamwork Manager Module[27] (this is part of the (commercial license) Modeliosoft Modelio distribution as of version 3). This module enables the collaborative development of Modelio models by integrating an SVN-based model-level versioning and comparison tool into Modelio.

It offers the usual SVN options for commiting and updating, while also providing model element level diffing between two versions (and merging them if required). Furthermore it uses the SVN lock/unlock functionality for model-aware locking of atomic units, which are either:

- An instance of a UML diagram, such as an *Activity* diagram or a *Class* diagram.
- An instance of a UML type, such as a *Class*, an *Interface* or a *Stereotype*.

This is possible as every instance of a diagram or type is stored in its own separate *.exml* file in the Modelio project structure of the SVN. As such, a developer can choose to lock any amount of model elements or diagrams to ensure only they can make changes to them until the lock is released.

The structure of a Modelio project in an SVN-based repository is as follows:

[26] `https://www.modelio.org/`
[27] `https://www.modeliosoft.com/en/modules/teamwork-manager.html`

- Top-level folder containing the Modelio project, named after the project (all other elements mentioned below are inside this folder)
- An “admin” folder containing various metadata items such as the Modelio meta-model version used in this project
- A “model” folder containing:

    A set of folders, one for each UML type and UML diagram defined in the current version of UML used, each containing:

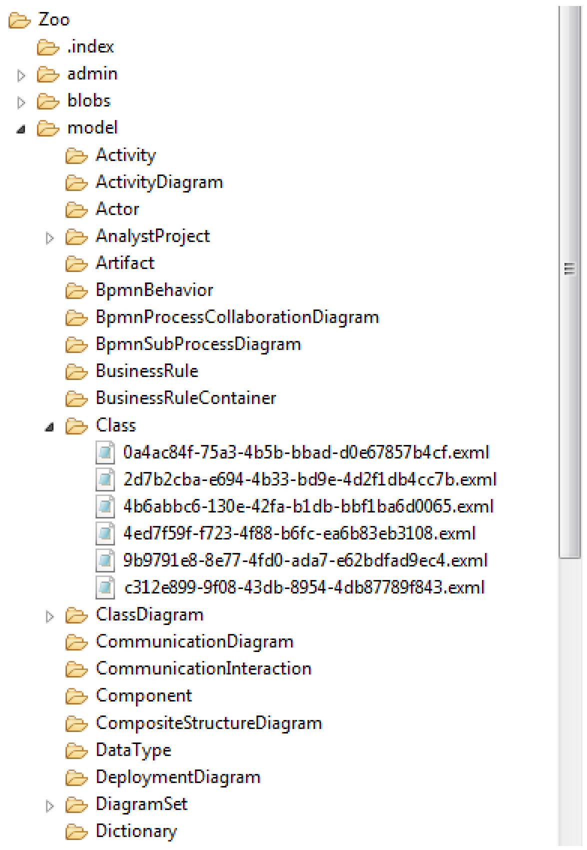
    A set of files, one for each instance of a relevant class/diagram in the project. For example in the folder named “Actor” there will be a set of files, one for each *Actor* instance found in the project; folders with no instances will exist but be empty.

This structure is seen in Figure 2.24 and is the same whether the Modelio project is stored locally or versioned in a repository.

Furthermore, it offers a detailed information window depicting the current state of the repository with respect to:

- Locked elements (by the current user)
- Locked elements (by other users)
- Modified elements (by the current user – with respect to the current repository version)
- Unversioned elements
- Added elements (elements which have been added to the version but have not yet been committed in the repository)
- Out of date elements (elements which have been modified by another user – not yet updated locally)

This architecture offers the advantages of model-element level locking as well as diagram level locking operations but will suffer in performance and resource use, when large models containing millions of fragments (i.e. millions of model elements) are used, as they will all have to be loaded into memory.



```
Zoo
  .index
  admin
  blobs
  model
    Activity
    ActivityDiagram
    Actor
    AnalystProject
    Artifact
    BpmnBehavior
    BpmnProcessCollaborationDiagram
    BpmnSubProcessDiagram
    BusinessRule
    BusinessRuleContainer
    Class
      0a4ac84f-75a3-4b5b-bbad-d0e67857b4cf.exml
      2d7b2cba-e694-4b33-bd9e-4d2f1db4cc7b.exml
      4b6abbc6-130e-42fa-b1db-bbf1ba6d0065.exml
      4ed7f59f-f723-4f88-b6fc-ea6b83eb3108.exml
      9b9791e8-8e77-4fd0-ada7-e62bdfad9ec4.exml
      c312e899-9f08-43db-8954-4db87789f843.exml
    ClassDiagram
    CommunicationDiagram
    CommunicationInteraction
    Component
    CompositeStructureDiagram
    DataType
    DeploymentDiagram
    DiagramSet
    Dictionary
```

Figure 2.24.: Directory structure of a Modelio project [for a Zoo model]

In the BPMN Loan example, the BPMN metamodel would be internally stored in the tool and the model would be stored as a collection of *.exml* files in their respective folders, one for each model element and diagram included in the model, as described above.

MagicDraw – UML modeling tool MagicDraw[28] is a commercial UML tool offering a collaboration module. This functionality allows for model-level versioning and collaboration through the Teamwork Server version control system offered. There are three available version control system alternatives offered by the Teamwork Server: Native, SVN and ClearCase.

A repository using the Native version control system uses a proprietary format to store versioned models as well as user credentials and access rights. On the other hand

[28] `http://www.nomagic.com/products/magicdraw.html#Collaboration`

a repository using a SVN or ClearCase VCS only stores user login names and actual authentication is performed directly with the respective VCS used, upon user a logging into the system.

To the user, the choice of the actual version control used is seamless, as they are offered the same functionality in all cases:

- Server administration – offering overview of users, projects and logs as well as options to change the type of repository or its location
- Client visualizations – various views are offered or affected by working on a project hosted on a Teamwork Server version control system:

  Lock View – displays the currently locked elements such as classes, relations and diagrams (these are the three types of data items MagicDraw distinguishes)

  Diagrams View and Containment View – both these views display current privileges of the user with respect to their contents. For example if the user has locked a diagram for editing, its link in the view is black and the name is next to it displaying they have locked it; any element not locked by the user is displayed in gray (denoting they cannot edit it)

  Editing windows – any window offering edit operations on elements will have the various add/remove/edit operations only enabled if the current element is locked by the user for editing

The structure of a Native repository is as follows:

- Top-level folder containing the entire repository, named after the repository itself (all other elements mentioned below are inside this folder)
- A *projects.xml* file containing information on the projects present in this repository
- A *users.xml* file containing information on the users registered to this repository – all information except the password is in plaintext
- A *.properties* file containing global repository properties, such as its unique identifier
- A *.log* file containing the server logs

- Folders for each project in this repository, containing:

  A folder containing project-related user information with:

    A folder for each registered user, containing a collection of *.zip* archives with the user's permissions, one archive for each version, containing various files, including a personal project options file containing a (non-human readable) serialized form of the user's preferences

  A *versions.xml* file containing information on the versions present in the project

  A *checkout.xml* file containing information on elements currently checked out (locked) by users

  A collection of *.zip* archives with the contents of each version of the project (complete copies of the entire project are kept in each version), each containing various (xml-based) files, including:

    A file with the contents of the entire model the project uses

    Various files with metadata about element versions, dependencies, project information etc.

The structure of an SVN-based repository is very similar to a native one, with the main difference being that instead of storing a *.zip* archive for each version of the project, only the latest version is stored (as it is versioned by the SVN repository itself, and hence all previous versions can be accessed).

In the BPMN Loan example, the BPMN metamodel would be internally stored in the tool and the model would be stored in MagicDraw's proprietary *.mdxml* format files.

MetaEdit+ – Domain-Specific Modeling environment MetaEdit+ is made up of two separate tools[29], one for creating domain-specific languages (DSL's) and one for using a DSL for defining models conforming to that DSL. Using the MetaEdit+ Workbench the developer can create their custom DSL and generate a MetaEdit+ Modeler to be used to instantiate this DSL. The MetaEdit+ Modeler can be customized depending on the needs of the DSL creator to better suit the specific use-case. MetaEdit also offers a collection of over 50 default languages which can be used if needed without the need for any metamodeling.

[29] http://www.metacase.com/products.html

MetaEdit+ offers an object database-based model server for storing models[30]. The back-end used is a proprietary version of the ArtBASE database system originally developed by ArtinApples Ltd.

The structure of such a server is as follows[31]:

- Top-level folder containing the entire repository, commonly named after the tool, such as "MetaEdit+ 5.1" (all other elements mentioned below are inside this folder)
- An *artbase.roo* file containing the names and paths of all repositories present on this server
- A set of folders, one for each repository, containing:

    A *manager.ab* file containing names and paths of the areas in the repository as well as the usernames and (encoded) passwords of users as well as any other information needed such as disk mappings. An area is the persistence-layer name for the project concept in the tool.

    An *areas* folder, containing:

        A set of folders, one for each area, each containing a set of files which make up the disk storage of the database used to store the model and any relevant project information

    A users folder, containing a folder for each user in the repository, each with a set of files with relevant user information

    A *backup* folder containing the last successfully commited version of the entire repository with:

        An *areas* folder with all the backup areas (with the same structure as the original areas folder)

        A *manager.ab* file containing the backup of the access control of the repository

Change propagation is performed using an ACID transaction system. Changes are only propagated to the repository (and to other users) when a transaction is commited.

[30] `http://www.metacase.com/papers/Mature_Model_Management.html`
[31] `http://www.metacase.com/support/51/manuals/sysadmin/sa.html`

Hence any user working in a transaction will not see any changes made to the repository until they open a new transaction by ending their current one.

Collaboration is supported when a multi-user repository is used. On the metamodel level, the entire metamodel is locked when a user is working on it and cannot be altered until the lock is released[32]. On the model level, each user will automatically be given locks on items they are working on, based on the data model of MetaEdit+:

- Conceptual Graphs – these represent the actual model data in the project
- Representational Graphs/elements – these represent the visual representation of various subsets of the actual data in the project (editors contain these views)
- Conceptual objects/relationships/roles/properties – these represent the various model objects and their interactions or attributes

More specifically, when a user opens an editor on a representational graph they will attempt to get both locks on the representational graph itself as well as the conceptual graph underlying it. If the user gets both locks they will be able to change the layout as well as the contents of this graph. If they only get the lock for the representational graph, they will only be able to change the layout of the graph or add/remove representational elements of currently existing conceptual elements. Similarly if they only get the conceptual lock, they will not be able to change the layout in the editor but will be able to alter the data in the elements. These locks are kept until released (e.g. the editor is closed), possibly through multiple transaction commits, should the user want to keep them.

On the other hand single element locks, such as ones needed for dialogs used in editing object properties, only hold temporary locks for as long as they are open, to facilitate multi-user editing. Similarly to other locks, failure to obtain one will simply offer a read-only editor graying out any "OK" option for applying changes.

The proposed approach for versioning models stored in this repository is to use classical file-based version control systems and store a zipped version of the entire repository every time a new version should be saved[33]; MetaEdit+ offers scripts to aid in incorporating this strategy into the tool's normal workflow but no integrated versioning system or way

[32] `http://www.metacase.com/faq/showfaq.asp?cate=MWB`

[33] `http://www.metacase.com/support/45/documentation/versioning.html`

to view any changes made in the repository, other than displaying the latest commited version.

In the BPMN Loan example, the MetaEdit+ back-end database (ArtBASE) would contain the Loan model and metamodel represented in its own proprietary format.

Table 2.1 overviews the capabilities of all the model persistence and versioning tools reviewed in this section.

Table 2.1.: Overview of state-of-the-art model persistence and versioning mechanisms

Tool	Persistence Format	Persistence Type	VCS	Fragmentation
EMF	XMI	text	✗	✓
Teneo/Hibernate	Relational DB	binary	✗	✗
Morsa	Document DB	binary	✗	✗
MongoEMF	Document DB	binary	✗	✗
NeoEMF$^{\text{[Graph]}}$	Graph DB	binary	✗	✗
NeoEMF$^{\text{[Map]}}$	Key/Value DB	binary	✗	✗
ModelCVS	XMI	text	✓	✓
CDO	Various DBs	binary	✓	✗
EMFStore	Document DB	binary	✓	✗
Modelio	XML	text	✓	✓
MagicDraw	XML	text	✓	✗
MetaEdit+	Object DB	binary	✓	✗

2.2.3. Model Querying

As model querying provides the primary means for retrieving information from models, all modeling technologies presented here need to support it in some form. Any modeling tool which needs to build upon one of these technologies will have to support querying the models in order to perform any operations on them, such as transformations into lower level artefacts.

2.2.3.1. Querying Technologies

Model querying is often reliant on the technology used to manage the model and its in-memory representation, or on tools built upon this technology offering additional functionality.

EMF-based models Persistence and versioning technologies relying on EMF for their in-memory model representation (EMF-XMI, Teneo/Hibernate, Morsa, MongoEMF, NeoEMF, ModelCVS, CDO and EMFStore) can all use the EMF *Resource* API to perform model traversal and retain the results they need in order to answer queries about models. This API allows for:

- retrieval of all the direct contents of a resource (i.e. only the root model elements)
- retrieval of all contents of the resource (as an iterator over all model elements in the resource)
- retrieval of the value of a specific feature of a model element (i.e. an attribute or reference value)
- retrieval of the meta-type of each model element

Furthermore it offers automatic resolution of inter-resource references and dependencies.

While this API offers a complete capability for navigating a model/metamodel and retrieving all of the information stored within the model, it can be seen as inefficient in various cases:

- In order to retrieve a single model element one has to either retrieve it by identifier, iterate the containment hierarchy of the resource or iterate the flattened collection of all model elements within the resource. In the worst case, one would have to iterate through the entire contents of the resource if the model element in question is the last one just to retrieve this one model element.
- In order to find all elements of a specific type, one would have to iterate through the resource and collect all model elements of interest by retrieving the type of each one and comparing it to the required type.
- In order to perform multiple such operations, it is to the discretion of the client code to use caching and avoid multiple traversals. Furthermore there is no support for

indexing facilities which could support complex queries (such as the one presented below).

For example in EMF, a query requesting all Tasks in the BPMN Loan model presented in Section 2.1.3, which have *executionTime* greater than 10 minutes (600 seconds) [from now on referred to as the “slow task query”] would be written in Java as follows:

```
//create the file resource from the bpmn loan model
Resource resource = resourceSet.getResource(URI.createFileURI(
   "bpmn_loan.model"));
//get the root of the resource, i.e. the BPMNProcess instance
EObject process = resource.getContents().get(0);
//get the BPMNProcess type
EPackage bpmn = EPackage.Registry.INSTANCE.getEPackage(
   "http://bpmn_simplified");
EClass bpmnProcessEClass = (EClass) bpmn.getEClassifier(
   "BPMNProcess");
//get the Tasks in this BPMNProcess
List<EObject> tasks = process.eGet(bpmnProcessEClass.
   getEStructuralFeature("tasks"));
//get the Task type
EClass taskEClass = (EClass) bpmn.getEClassifier("Task");
//only keep the slow tasks
for(EObject task : tasks)
 if(task.eGet(taskEClass.getEStructuralFeature(
    "executionTime")) <= 600)
   tasks.remove(task);
//tasks now contains the appropriate model elements for the query
```

Various technologies extend this API in order to provide more performant querying solutions. For example Morsa has introduced the *Morsa Query Language* [63]. The abstract syntax of this language is seen in Figure 2.25 from [63].

The goal is to provide a language consistent with SQL (using SELECT – FROM – WHERE clauses) while providing constructs for EMF-based concepts (EClass – EObject etc.). For example the slow task query in the Morsa Query Language in Java would look like:

```
//get the root of the BPMN Loan EMF resource exposed by Morsa, i.e.
   the BPMNProcess instance
```

```
EObject bpmnProcess = (EObject)resource.getContents().get(0);
//create the query ( 'GT("attr",y)' represents the value of attribute
     "attr" is greater than 'y' )
MorsaQuery query = SELECT("http://bpmn_simplified/Task")
   .FROM(bpmnProcess)
   .WHERE(GT("executionTime",600)).done();
//retrieve the results from the resource connected to the Morsa back-
   end
Collection<EObject> result = morsaResource.query(query);
```

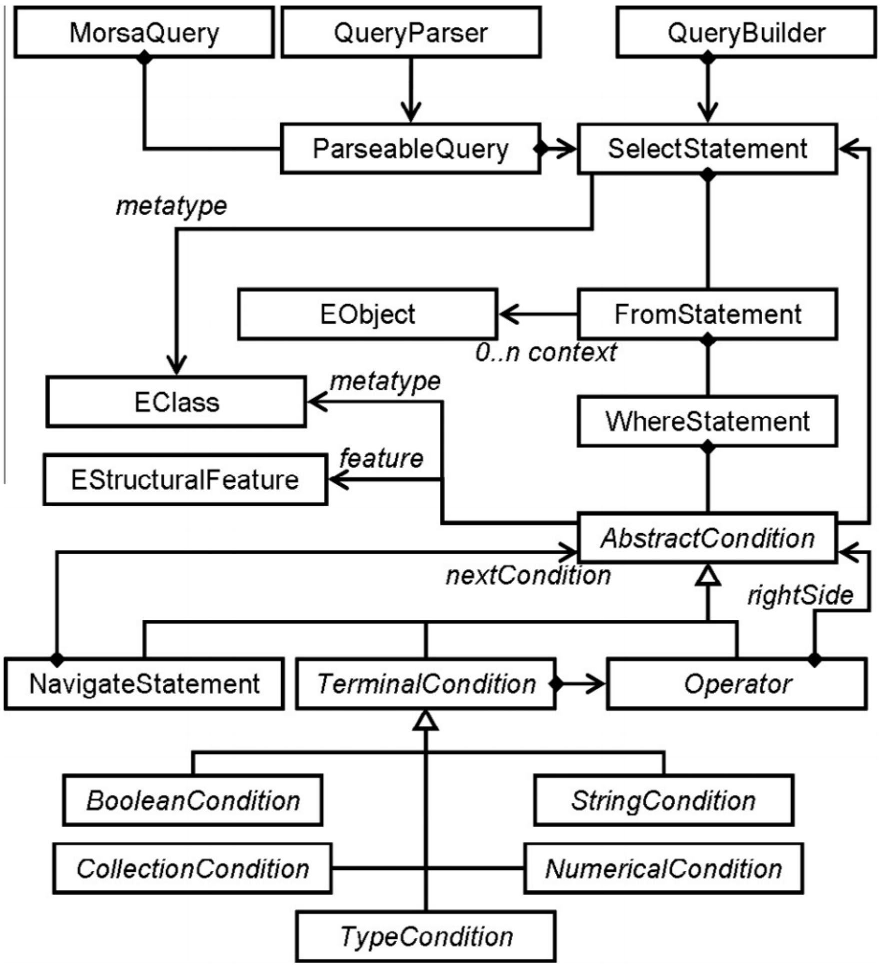


Figure 2.25.: The abstract syntax of the Morsa Query Language, from [63]

MongoEMF[34] supports the *MongoDB native* query language[35], that can be used to query its underlying store. This simple language allows retrieval of documents (in this case model elements) from a MongoDB collection (i.e. of a specific model Type in MongoEMF) by filtering on feature values; it also allows for limiting the number and sorting the results, as well as choosing which features are included or excluded from the resulting elements.

For example the slow task query for MongoEMF would be presented in Java as follows (this assumes that all instances of Task are stored in a MongoDB collection named "tasks"):

```
//create the query to the store ( '$gt' represents '>' )
String query = "{ filter: { executionTime: { $gt: 600 } } }";
//create the resource set
ResourceSet resourceSet = resourceSetFactory.createResourceSet();
//create the resource keeping the results, passing it the query
Resource resource = resourceSet.getResource(URI.createURI(
   "mongodb://localhost/app/tasks/?" +query), true);
//retrieve the results (located at the root of the query resource)
EReferenceCollection users = (EReferenceCollection) resource.
   getContents().get(0);
```

UML-based models Technologies using UML (Modelio and MagicDraw) can leverage the Object Constraint Language (OCL[36]) as an abstraction over the actual technology used to persist and read their models.

The original purpose of OCL was to offer a side-effect free way of providing constraints on UML models in order to help describe them and to validate their correctness but the expression language can be used for creating any read-only query on model. As such OCL is being used as a more natural way of expressing queries on UML models [15].

MagicDraw directly supports use of OCL constraints and even though Modelio does not directly support OCL, the PyAlaOCL Modelio Integration tool[37] uses Modelio's Jython[38] scripting capabilities for supporting OCL expressions.

[34] https://github.com/BryanHunt/mongo-emf/wiki/User-Guide
[35] https://docs.mongodb.org/manual/core/read-operations-introduction/#query-interface
[36] http://www.omg.org/spec/OCL/
[37] pyalaocl.readthedocs.org/en/latest/modelioIntegration.html
[38] http://www.jython.org/

For example, the diverging gateway query in OCL would be written as follows:

```
BPMN::Task.allInstances() -> select( st | st.executionTime > 600 )
```

MetaEdit+ models MetaEdit+ does not offer a dedicated querying facility as it's developers did not deem it useful in its 20 year review[39]. It relies on its various visual editors for allowing users to navigate through their models and finding the information they need; furthermore users can leverage the generation capabilities of the tool to produce reports or code with the contents of their intended query. Finally MetaEdit+ models can be exported to XML and prototype work is available for their loading by other tools such as EMF [64] and hence queried using their capabilities.

2.2.3.2. Repository Querying

Querying of model repositories is more complex than that of models loaded into memory. Concerns such as network overhead and server availability/resource use need to be taken into account when developing a query interface for a model repository.

EMF-based repositories expose their contents as an EMF *Resource*, often using a lazy loading approach and caching strategies, for retrieving contents on demand, in order to avoid network overload.

Similarly to various persistence technologies, many EMF-based repositories offer complementary querying capabilities to the basic EMF resource paradigm. For example CDO offers an in-built execution engine for evaluating OCL expressions (using the Eclipse MDT OCL engine[40]); hence users are able to write OCL expressions to run against models stored in CDO with the query being evaluated on the server side and the results returned to the client.

For example running the diverging gateway query in OCL for CDO in Java would be presented as follows:

```
//get the session and view hosting the query
CDOSession = getSession();
CDOView view = session.openView();
//get the BPMNProcess instance
BPMNProcess process = (BPMNProcess)resource.getContents().get(0);
```

[39] http://www.metacase.com/papers/MetaEdit+_at_the_Age_of_20.pdf

[40] http://www.eclipse.org/modeling/mdt/?project=ocl

```
//create the OCL query on this BPMNProcess
CDOQuery query = view.createQuery( "ocl", "self.tasks -> select(
   st | st.executionTime > 600 )" , process );
//retrieve the results
List<ParallelGateway> result = query.getResult();
```

MagicDraw offers the ability to query both on the metamodel and the instance level using Java, scripting languages such as Jython and the Velocity Template Language[41].

Modelio offers a *QueryDefinition* API[42] for extending the tool by offering a querying facility (compatible with EMF).

As mentioned above, MetaEdit+ does not directly offer a textual query functionality but uses its extensive diagram editors to facilitate this process[43].

2.2.4. Summary

This section presented the current state-of-the-art tools and technologies used for model persistence and versioning, discussing their different features and capabilities; it also introduced model querying and how it is handled by the various persistence technologies and repositories presented in this section. The next chapter synthesizes the findings of this review and identifies the challenges related to collaborative development of large (collections) of models that the rest of the work will target.

[41] http://www.nomagic.com/files/manuals/MagicDraw%20ReportWizard%20Template%20Creation%20Tutorial.pdf
[42] https://www.modelio.org/documentation/javadoc-3.4/org/modelio/metamodel/uml/infrastructure/matrix/QueryDefinition.html
[43] http://metaphor.it.jyu.fi/loppurap/mearch.html

3. Analysis and Hypothesis

Summarizing the discussion in Chapter 2, this chapter identifies the limitations of today's state-of-the-art modeling tools, identifies an approach which can be used to tackle some of the scalability issues described in Section 2.1.2.1 and presents the research hypothesis of this work. It then lists the proposed research objectives for assessing the validity of this hypothesis and discusses the scope of the research.

3.1. Analysis

In a collaborative environment, models need to be version-controlled and shared among many developers. As discussed in Section 2.2.2 there are two primary approaches taken for versioning models. The first is using file-based version control systems such as Git or SVN, which has certain advantages as such version control systems are robust, widely-used and orthogonal to modeling tools, the vast majority of which persist models as files. On the downside, since such version control systems are unaware of the contents of model files, performing queries on models stored in them requires developers to check these models out locally first. As seen in Figure 3.1, this can be particularly inefficient for queries extending to a large number of models (global queries). For example in order to find out how many models in the repository reference the developer's model A, all other models will have to be checked out from the repository (in this case models B and C), and they would subsequently have to be loaded into memory (alongside model A), in order to be amenable to querying. Even if all the fragments are maintained locally (for example if a version control system such as Git is used, as show in Figure 3.2), the issue of having to load them all into memory in order to answer the query remains; for large enough models (or collections of fragments) this will commonly require too many resources to be done. Also, file-based version control systems do not provide support for model-element level operations such as locking or change notifications. To address these limitations, model-specific version control systems such as CDO, EMFStore and

MagicDraw's TeamServer have been proposed. While such systems address some of the limitations above, they require tight coupling with modeling tools, they impose an administration overhead, and they lack the maturity, robustness and wide adoption of file-based version-control systems.

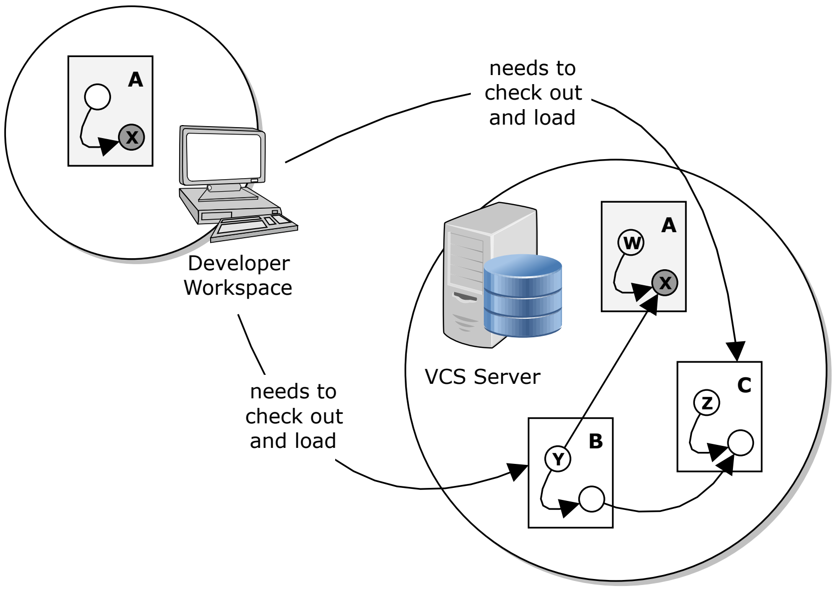


Figure 3.1.: Performing global queries on model fragments stored in a remote VCS repository, adapted from [31]

3.1.1. Model Indexing

In what can be seen as a happy medium between the two approaches to model version control, we introduce the concept of *model indexing*. This is an approach that enables efficient global model-element-level queries on collections of models stored in file-based version control systems. To achieve this, a separate system is introduced (a *model index*), which monitors file-based repositories and maintains a fine-grained read-only representation (graph) of models of interest, which is amenable to model-element-level querying. As seen in Figure 3.3, the developer queries the model indexer directly and does not have to check out or load any additional files.

Such a model index is not aimed at replacing the current model persistence formats (such as XMI, database or model repository) but offer an up-to-date proxy to the model

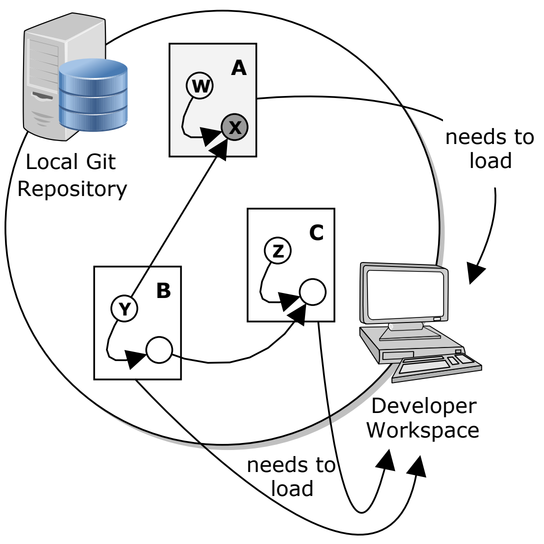


Figure 3.2.: Performing global queries on model fragments stored in a local VCS repository

data. Paralleled with database indexing, the goal of a model index is to offer the ability to retrieve a subset of information in a model, more efficiently than using iteration. As such, in principle, the contents of a model index are determined by the use-case in question and can range from including as much as all of the model features (offering an almost mirrored copy of the model that can be queried much more efficiently – a use-case we use in the evaluation of this work), to as little as no actual information contained in the model but instead only containing other interesting "metadata" items such as timestamps of commits, number of model element-level changes found in each commit or number of model files changed in each commit.

3.2. Research Hypothesis

Such a model indexing system can be used to investigate the following research hypothesis:

> The overhead of computing model-element-level queries over large (collections of) models stored in a file-based VCS can be significantly reduced using a non-invasive model-indexing system orthogonal to the specific VCS or model representation format.

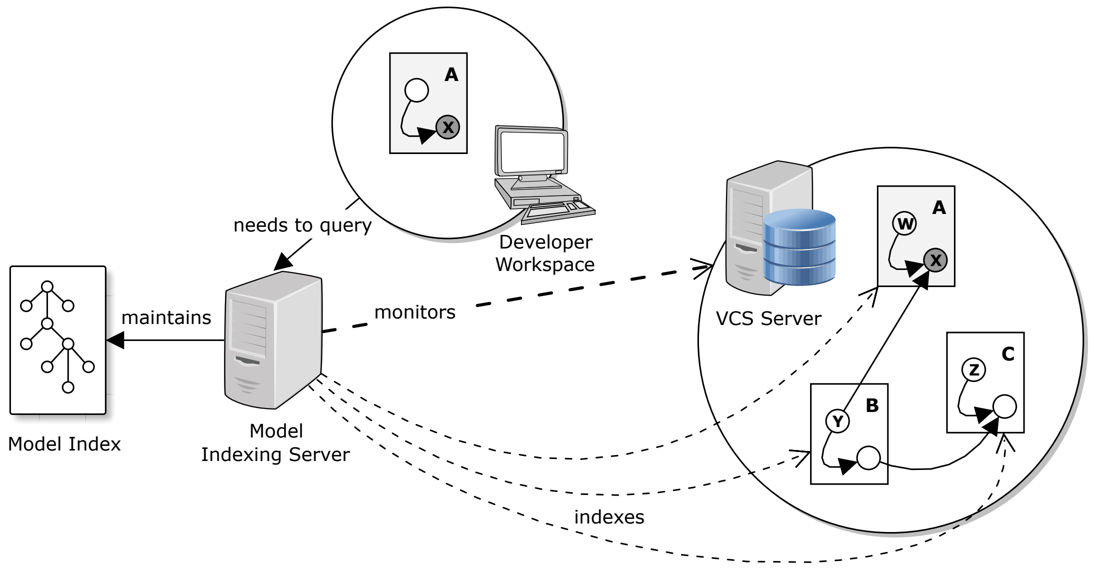


Figure 3.3.: Performing global queries on model fragments stored in a remote VCS repository, using a model indexing system, adapted from [31]

Section 3.3 presents the research objectives used to assess the validity of this research hypothesis and Chapter 4 delves into the details of designing such a model indexing system, with Chapter 5 evaluating its effectiveness and efficiency.

3.3. Research Objectives

In order to assess the validity of the research hypothesis, the following research objectives were defined:

(i) Development of a model indexing system prototype; work presented in [65].

(ii) As a model indexing system has to be performant and scalable to be of use, the back-end used to store the data needs to be appropriately chosen. As such, an evaluation of various appropriate data persistence technologies will be performed, in order to identify the one most fitting for a model indexing system. This work has been presented in [34] as well as Section 5.3.1.

(iii) For a model indexing system to be useful to a variety of tools and domains, it has to be able to support multiple model persistence formats as well as offering an API for the addition of new formats on demand. As such, an extensibility mechanism to support multiple model persistence formats will be offered. The extensible architecture supporting this, as well as example drivers for alternative technologies, are presented in Section 4.3.4.

(iv) To ensure breadth of coverage, an extension of the model indexing system to support multiple file-based VCS will be offered. Details can be found in Section 4.3.3.

(v) In order to gain confidence in the usefulness of the system when non-trivial models are being used, it will be extended to support indexing and querying large (collections of) models.

(vi) An evaluation of index creation and updating in terms of correctness and performance will be performed using models conforming to research objective (v); work presented in [66].

(vii) An evaluation of typical forms of model queries in terms of performance, when compared to querying performed by current state-of-the-art modeling tools, will be performed using models conforming to research objective (v). Work presented in [67].

Chapter 4 addresses objectives (i), (iii), (iv) & (v) and Chapter 5 objectives (ii), (vi) & (vii).

3.4. Scope

In the interest of feasibility, the research (and consequent tackling of the research objectives) is scoped as follows:

- To address research objective (ii), an appropriate back-end will be selected after empirical study comparing various appropriate alternatives. As all possible alternatives cannot be investigated, the study will be limited to some of the technologies prominently used in today's state-of-the-art modeling tools (identified in Section 2.2.1) such as XMI, relational databases and NoSQL graph stores like Neo4J and OrientDB.
- To address research objective (iii), the framework will support the indexing of three

independently-implemented model persistence formats comprising EMF-based models as well as BIM[1] IFC[2] and Modelio UML models.

- To address research objective (iv), the models will be stored in file-based version control systems like SVN and Git and will be updated through the built-in facilities of these systems.
- To address research objective (v), large models, of the order of millions of model elements, will be indexed by the model indexing system; also, large collections of (hundreds) of smaller models will also be indexed. This aims at covering the cases where a few large monolithic models are present as well as when many (possibly fragmented) smaller models are used instead.
- To address the querying aspect of research objective (v), the framework will support EOL as a query language (as a representative of a wide array of OCL-like languages and variants of OCL embedded in languages such as ATL and Acceleo). Alternative querying methods such as use of native Java or back-end specific languages will be investigated in order to contrast their relative expressiveness, usability, maintainability and performance.
- To address research objectives (vi) and (vii), empirical studies will be performed regarding model insertion, model updating and model querying in terms of correctness and performance. The scope of these studies will be limited to the above-mentioned formats, systems and sizes, as well as an appropriate class of queries, representative of ones commonly performed in MDE.

[1] Building Information Modeling: `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=52155`
[2] Industry Foundation Classes: `http://www.iso.org/iso/catalogue_detail.htm?csnumber=51622`

4. Hawk: Scalable Model Indexing Framework

This chapter presents *Hawk*, a model indexing framework and proof of concept implementation used to assess the validity of the research hypothesis. Hawk monitors locations where file-based models are stored and it then parses these models in order to maintain a unified read-only representation of the contents of the models in its model index. This index is consistent with the state of the model files and is amenable to efficient querying of its contents. In this chapter, the functionality and architecture of the framework are outlined followed by the design and implementation details of the prototype, including details of the various key processes, procedures and algorithms used.

4.1. System Capabilities

In order to address the research objectives identified in Section 3.3, the proposed framework aims at delivering the following capabilities:

1. Use a scalable back-end so that it can accommodate large collections of models in the order of millions of model elements or large collections of (possibly fragmented) models;
2. Provide an efficient internal representation of the models it indexes, amenable to efficient querying;
3. Support modeling technologies which use cross-file references to store fragmented models;
4. Support incremental approaches for updating its contents in order to avoid unnecessary overheads when making small updates on large collections of models;

5. Work with diverse file-based version control systems (e.g. SVN, Git) and modeling technologies (e.g. EMF, IFC);
6. Offer a facility through which modeling & model management tools can query it;
7. Support use of database indexing and attribute caching to improve the performance of certain classes of queries;
8. Provide extensibility mechanisms for accommodating new types of Back-ends, VCSs and model representation formats;
9. Be orthogonal: the VCS repositories should not need to be modified or configured in any way. This tackles the need for orthogonality of the research hypothesis;

Table 4.1 links the system capabilities to their relevant research objectives, found in Section 3.3.

Table 4.1.: Traceability of System Capabilities

System Capability	Research Objective(s)
1	(ii)
2	(v)
3	(v)
4	(v)
5	(iv)
6	(v)
7	(v)
8	(iii), (iv)

4.2. System Architecture

To satisfy the requirements outlined above, Hawk can be configured to monitor a set of VCS servers for model-related events (e.g. creation of new models, modification of existing models) and maintain a copy of the latest version of all interesting (based on its configuration) models in these servers in a scalable model index (defined in Section 3.1.1), the details of which are discussed in the sequel. Such an architecture avoids the problem of having to fully load the models every time it needs to answer a query by front-loading this effort (into its persisted model index) and only having to propagate any changes to the models (into the model index) as they happen. Modeling and model management tools can then perform global queries through Hawk, exploiting its fast, scalable and synchronized model index. With regard to the VCS servers, Hawk acts as a standard read-only client to satisfy the requirement of non-invasiveness.

4.2.1. System Components

Figure 4.1 illustrates the components that comprise Hawk, and which are discussed below. Figure 4.2 shows the related API provided by Hawk for these components. These interfaces are discussed in detail throughout this section (Section 4.3). The detailed API of each interface element can be found in Appendix A.

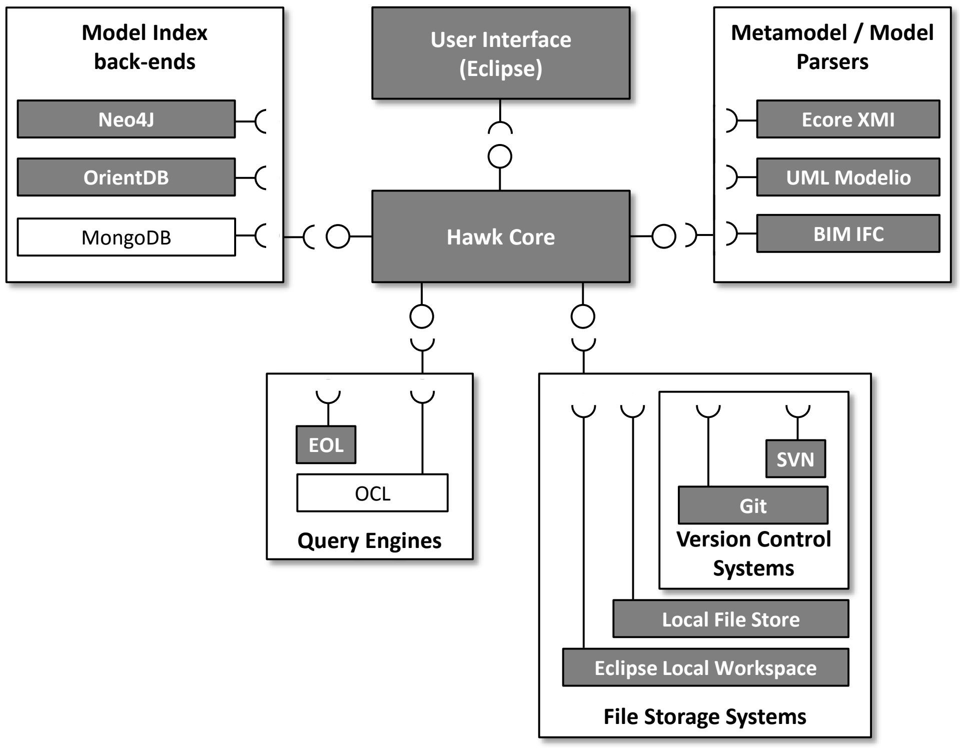


Figure 4.1.: Components of the model indexing system (Hawk)

- **Model parser components:** these components provide parsers for specific model persistence formats, such as EMF models persisted in XMI or Modelio models in XML. These parsers take as input the contents of a file stored in version control systems and produce as output a uniform in-memory representation ("resource") of the model.

- **Model indexing components:** these components, specific for each back-end used (such as a Neo4J NoSQL database), receive a resource created by the appropriate model parser, and insert/update it into the database. The structure of the store assumes that the back-end provides a mechanism for rapidly accessing specific elements using a key (for example by using embedded database indexes, such as the Apache Lucene[1] Index)[2].

- **File Storage/VCS components:** specific for each system, these components compute the set of changed files (added, removed or updated) with respect to the current local model index revision (and the current latest version of the files in the system/repository).

- **Core component:** is responsible for initializing, managing and gracefully terminating Hawk. It is responsible for synchronizing the model index with its relevant version control system(s), either at time intervals defined by some heuristic or when instructed to do so through its interface. This comprises querying the VCS for changed files (with respect to the current local model indexer revision), providing these files to the appropriate model parser (in order to create in-memory resources) and finally passing the returned resources to the appropriate model index(es) so that they can be synchronized with the current contents of the model indexer.

- **GUI component:** provides a front-end to Hawk, thus allowing for model indexes to be added/removed and managed. It can also be used to perform queries on the model indexes using any of the currently active query engines. Hawk can also be used without its graphical user interface should the user want to manage Hawk programmatically or even without depending on external frameworks (such as Eclipse's[3] implementation of the OSGI[4] framework).

- **Query API:** this component provides a bridge between Hawk and modeling and model management tools that need to query its model indexes.

[1] `http://lucene.apache.org/core/`

[2] this assumption is made as most popular relational/NoSQL stores (such as MySQL, MongoDB and Neo4J) include database indexes

[3] `https://eclipse.org/`

[4] `http://www.osgi.org/Main/HomePage`

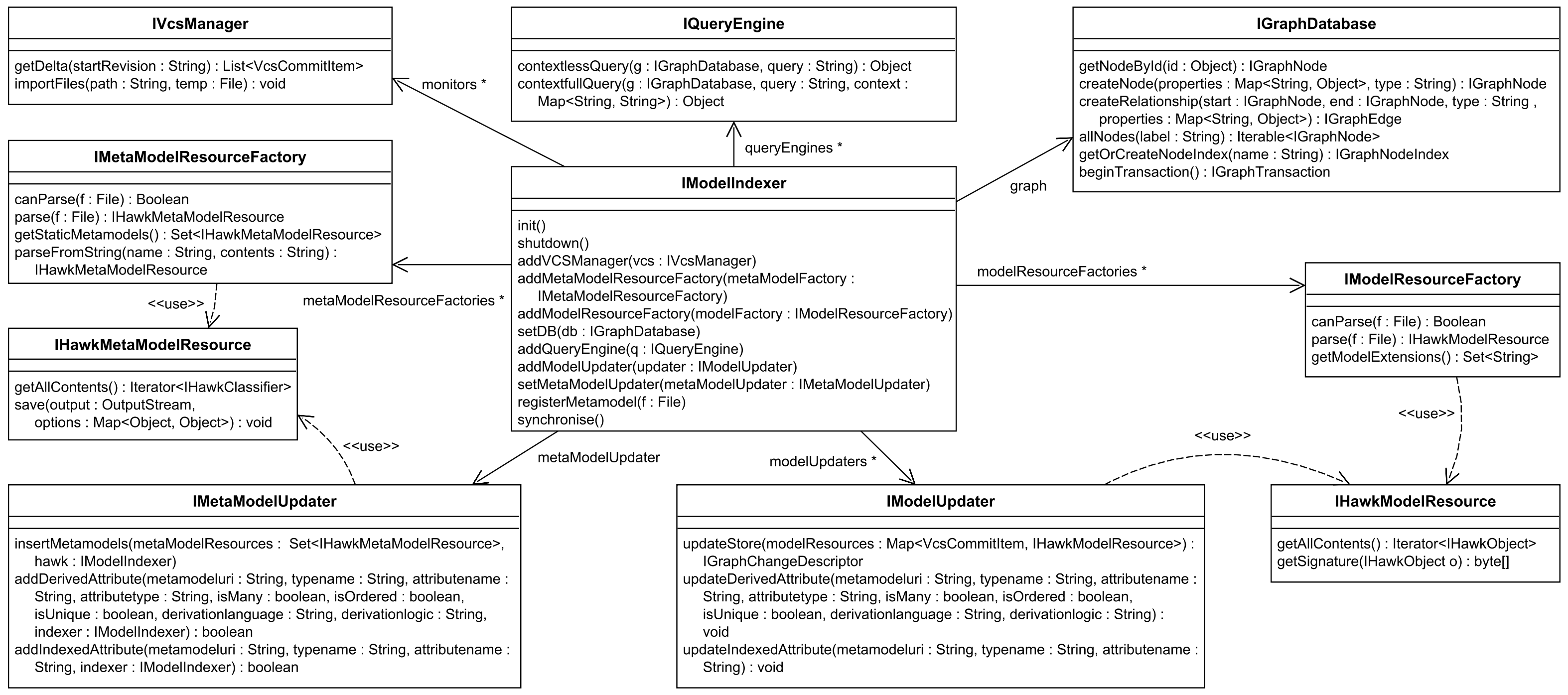

IVcsManager
getDelta(startRevision : String) : List<VcsCommitItem>
importFiles(path : String, temp : File) : void
IQueryEngine
contextlessQuery(g : IGraphDatabase, query : String) : Object
contextfullQuery(g : IGraphDatabase, query : String, context : Map<String, String>) : Object
IGraphDatabase
getNodeById(id : Object) : IGraphNode
createNode(properties : Map<String, Object>, type : String) : IGraphNode
createRelationship(start : IGraphNode, end : IGraphNode, type : String , properties : Map<String, Object>) : IGraphEdge
allNodes(label : String) : Iterable<IGraphNode>
getOrCreateNodeIndex(name : String) : IGraphNodeIndex
beginTransaction() : IGraphTransaction
monitors *
queryEngines *
graph
IMetaModelResourceFactory
canParse(f : File) : Boolean
parse(f : File) : IHawkMetaModelResource
getStaticMetamodels() : Set<IHawkMetaModelResource>
parseFromString(name : String, contents : String) : IHawkMetaModelResource
IModelIndexer
init()
shutdown()
addVCSManager(vcs : IVcsManager)
addMetaModelResourceFactory(metaModelFactory : IMetaModelResourceFactory)
addModelResourceFactory(modelFactory : IModelResourceFactory)
setDB(db : IGraphDatabase)
addQueryEngine(q : IQueryEngine)
addModelUpdater(updater : IModelUpdater)
setMetaModelUpdater(metaModelUpdater : IMetaModelUpdater)
registerMetamodel(f : File)
synchronise()
modelResourceFactories *
IModelResourceFactory
canParse(f : File) : Boolean
parse(f : File) : IHawkModelResource
getModelExtensions() : Set<String>
<<use>>
metaModelResourceFactories *
IHawkMetaModelResource
getAllContents() : Iterator<IHawkClassifier>
save(output : OutputStream, options : Map<Object, Object>) : void
<<use>>
<<use>>
<<use>>
metaModelUpdater
modelUpdaters *
IMetaModelUpdater
insertMetamodels(metaModelResources : Set<IHawkMetaModelResource>, hawk : IModelIndexer)
addDerivedAttribute(metamodeluri : String, typename : String, attributename : String, attributetype : String, isMany : boolean, isOrdered : boolean, isUnique : boolean, derivationlanguage : String, derivationlogic : String, indexer : IModelIndexer) : boolean
addIndexedAttribute(metamodeluri : String, typename : String, attributename : String, indexer : IModelIndexer) : boolean
IModelUpdater
updateStore(modelResources : Map<VcsCommitItem, IHawkModelResource>) : IGraphChangeDescriptor
updateDerivedAttribute(metamodeluri : String, typename : String, attributename : String, attributetype : String, isMany : boolean, isOrdered : boolean, isUnique : boolean, derivationlanguage : String, derivationlogic : String) : void
updateIndexedAttribute(metamodeluri : String, typename : String, attributename : String) : void
IHawkModelResource
getAllContents() : Iterator<IHawkObject>
getSignature(IHawkObject o) : byte[]

Figure 4.2.: Interfaces of Hawk

4.3. System Design

This section details the design of Hawk, presenting how the various API elements it offers are used, as well as its core algorithms and procedures for indexing updating and querying. It then describes the various optimizations performed to enhance the performance of Hawk in various ways, such as for model updates or model queries.

4.3.1. Hawk Model Index Structure

Hawk persists information extracted from models as a global property graph. In Figure 4.3 we can see how storing the BPMN Loan model in this manner looks like (as the whole model would not fit in a single page, only a representative subset of the model is shown).

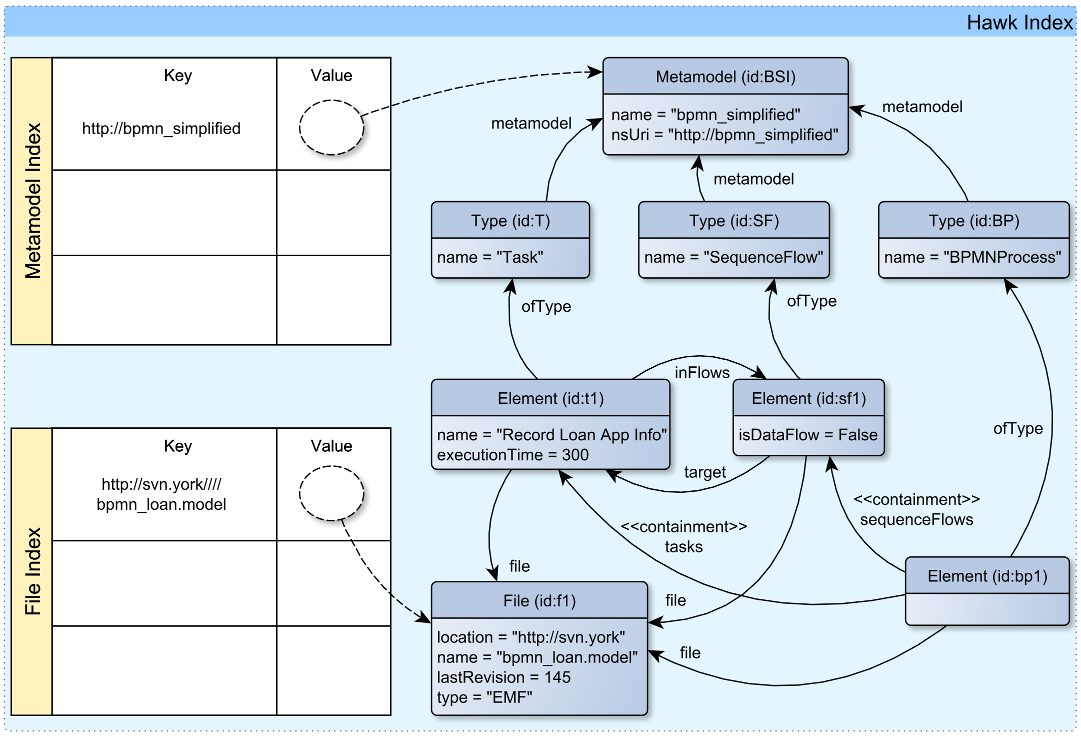


Figure 4.3.: The BPMN Loan model index – stored as a property graph (partial)

In general, a Hawk model index typically contains the following entities:

- **File nodes**. These represent files in a repository and contain information on the file such as the repository path, file name, current revision and type. They are linked with relationships to the *Element*s they contain. For example, in Figure 4.3 node with id:f1 is a file node.

- **Metamodel nodes**. These represent metamodels and contain their names and their unique namespace URIs (in EMF, these would be *EPackages*[5]). They are linked with relationships to the (metamodel) *Type*s they contain. For example, in Figure 4.3 node with id:BSI is a metamodel node.
- **Type nodes**. These represent metamodel types (*EClass*es in EMF terminology) and contain their name. They are linked with relationships to their (model) *Element* instances. For example, in Figure 4.3 node with id:T is a type node.
- **Element nodes**. These represent model elements (*EObject*s in EMF terminology) and can contain their attributes (as properties) and their references (to other model elements) as relationships to them. For example, in Figure 4.3 node with id:t1 is an element node.
- **Database Indexes**. Metamodel nodes and File nodes are indexed[6] in the store, so that their nodes can be efficiently accessed for querying (commonly used as starting points for complex graph traversal queries). For example, on the left hand side in Figure 4.3, the two indexes can be seen.

4.3.1.1. Back-end Persistence

As mentioned above, Hawk focuses on the use of Graph-based NoSQL databases for its back-end as they were identified to be the most performant [41], when dealing with large collections of models identified in Section 2.1.2. Initial evaluation of these stores was performed between two largely used graph databases – Neo4J and OrientDB. Even though both stores expose a graph API with nodes and relationships, the actual core persistence of OrientDB is as a document store and this seemed to degrade its performance in all of the benchmarks performed; detailed analysis of this evaluation is presented in Section 5.3.1. As such it was decided to focus on the more performant Neo4J store for creating the proof of concept back-end for Hawk. The OrientDB back-end has also been kept up to date in order to provide an alternative back-end for Hawk, but will not be focused upon in this work.

[5] we choose to draw parallels with concepts from EMF as they are well-understood and unambiguous
[6] `http://components.neo4j.org/neo4j-lucene-index/snapshot/`

4.3.1.2. Background: Neo4J

Neo4J is a Java-based graph database; it was one of the first such technologies and has matured to enterprise standards in the past years[7]. It claims to be one of the fastest and most scalable native graph databases available and has positive feedback from multiple companies which use it. It offers ACID transaction support and a proprietary query language – Cypher[8], which enables querying a graph-like structure in a simple and efficient way; an example of a query written in Cypher can be found in Section 5.2.1.1.

It stores its data in multiple files on disk, each with specific content, some of which are listed below:

1. a file with the nodes of the graph
2. a file with the relations between the nodes of the graph
3. an index file of the node properties
4. a file with the node property values (that are Strings)
5. a file with the node property values (that are Arrays)
6. a set of files storing (Lucene[9]) index information
7. a set of files containing the latest logical logs (the latest transactions commited to the store)

As such, should a non-cached (in memory) element be required, it is retrieved from its relevant file. For example if a node with ID X is needed, the node file is searched to retrieve it; if subsequently the value of the property with name Y is needed for node X, the relevant property file is searched. Empirical tests have shown that these files are searched from their beginning to their end (or until the process using them has retrieved all it needs and breaks) and are ordered by ascending order of ID values (that are – Java *Long* – integers). The tests accessed nodes of varying internal IDs (which are known beforehand) and showed that the time it took to retrieve the relevant node

[7] `http://neo4j.com/`

[8] `http://neo4j.com/docs/stable/cypher-query-lang.html`

[9] Neo4J supports the use of a customly configured embedded Lucene index implementation for indexing commonly accessed node properties (either automatically or manually). In section 4.3.8.3 we discuss how the use of such custom manual indexes can be used to greatly increase the performance of various types of queries on Hawk

was proportional to the relative positioning of the node in the file (the larger the id the longer it took). Furthermore, this process took a constant time when running in Cypher, regardless of the IDs of the nodes being retrieved, as the whole file was read in each run (as there is no way to forcefully end the run after the needed results are found such as using a *break* statement in Java).

It is worth noting that this interaction is only relevant for nodes not cached in memory by the database, something which will happen automatically after the first time a node is accessed and until the database decides to remove it (for example if it reaches a memory threshold) or the machine is restarted (as simply shutting down the JVM is not sufficient to clear the memory mapped file from the operating system – at least for a Windows machine).

4.3.2. Hawk Mapping Layers

In order for a collection of heterogeneous model files (stored on VCSs) to be stored as a global property graph (such as the one presented above), two mapping layers had to be introduced by Hawk.

4.3.2.1. Model Layer

This layer provides a set of abstractions for representing heterogeneous models and metamodels in memory. Inspired by EMF's respective abstractions (and following the structure seen in Figure 2.3), *metamodel resources* contain types/meta-classes (that are grouped in packages), which have typed attributes and references, as well as annotations. *Model resources* contain objects representing model elements, which have values for the attributes and references of their type. The requirements of this layer are to provide a minimal interface for such elements so that wrapping current modeling technologies to them (so that they can be indexed in Hawk) is made as simple as possible.

The important interfaces in this layer are described below (and seen in Figure 4.4):

- *IHawkMetaModelResource* This interface provides the in-memory representation of a metamodel. It can contain:

  IHawkPackage This interface represents a uniquely identifiable collection of metamodel type. It has a namespace URI to be identified by as well as a name; it offers methods for retrieval of any specific *IHawkClassifier* (*IHawkClass* or *IHawk-DataType*) it contains (by name), or all of them.

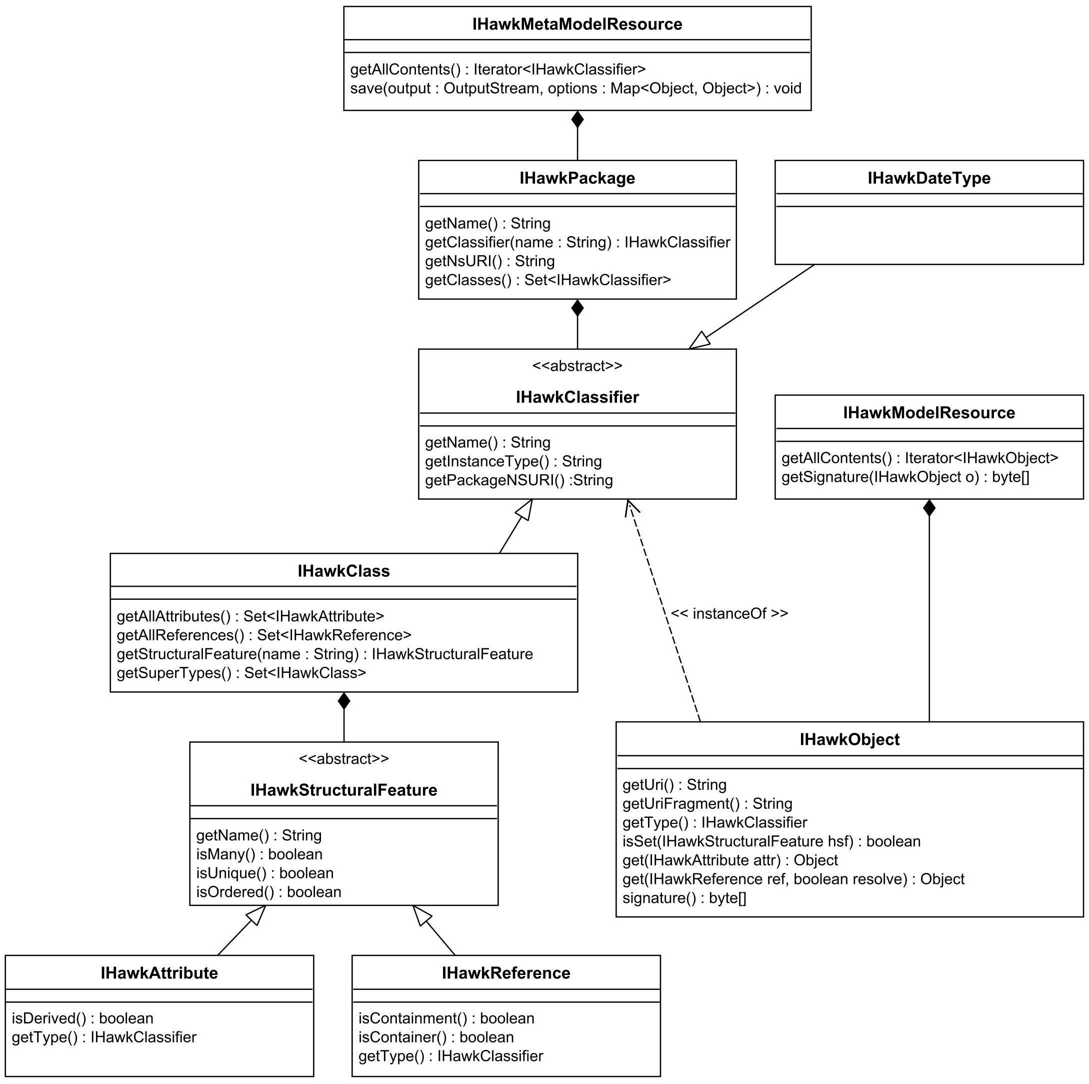

IHawkMetaModelResource
getAllContents() : Iterator<IHawkClassifier>
save(output : OutputStream, options : Map<Object, Object>) : void
IHawkPackage
getName() : String
getClassifier(name : String) : IHawkClassifier
getNsURI() : String
getClasses() : Set<IHawkClassifier>
IHawkDateType
<<abstract>>
IHawkClassifier
getName() : String
getInstanceType() : String
getPackageNSURI() :String
IHawkModelResource
getAllContents() : Iterator<IHawkObject>
getSignature(IHawkObject o) : byte[]
<< instanceOf >>
IHawkClass
getAllAttributes() : Set<IHawkAttribute>
getAllReferences() : Set<IHawkReference>
getStructuralFeature(name : String) : IHawkStructuralFeature
getSuperTypes() : Set<IHawkClass>
IHawkObject
getUri() : String
getUriFragment() : String
getType() : IHawkClassifier
isSet(IHawkStructuralFeature hsf) : boolean
get(IHawkAttribute attr) : Object
get(IHawkReference ref, boolean resolve) : Object
signature() : byte[]
<<abstract>>
IHawkStructuralFeature
getName() : String
isMany() : boolean
isUnique() : boolean
isOrdered() : boolean
IHawkAttribute
isDerived() : boolean
getType() : IHawkClassifier
IHawkReference
isContainment() : boolean
isContainer() : boolean
getType() : IHawkClassifier

Figure 4.4.: The Hawk Model Layer

IHawkClass This interface represents a metamodel class (a type). It has a name, unique within its *IHawkPackage*; it offers methods for retrieval of any specific structural feature (attribute or reference) it contains, all attributes it contains, all references it contains and all of its supertypes (other *IHawkClass*es it is a subclass of); it can be marked as abstract or interface.

IHawkDataType This interface represents a metamodel data type. This classifier only has a name, unique within its *IHawkPackage*; all other implementation details are left to the implementer.

IHawkAttribute This interface represents an attribute of an *IHawkClass*. It has a name, unique within its *IHawkClass*; it can be marked as derived, unique, ordered or with multiplicity greater than 1 (i.e. *isMany()*).

IHawkReference This interface represents a reference of an *IHawkClass*. It has a name, unique within its *IHawkClass*; it can be marked as containment, unique, ordered or with multiplicity greater than 1 (i.e. *isMany()*).

- *IHawkModelResource* This interface provides the in-memory representation of a model. It contains:

  IHawkObject This interface represents a model element. It has a URI, unique within the *IHawkModelResource*; it has a type (an *IHawkClassifier*); it has methods for retrieval of a specific attribute or reference value as well as to indicate whether a structural feature (attribute/reference) is currently set for this element; it has a *signature* (proxy to its current state); the value of this proxy should be unique every time any model-level change happens to this object (such as an attribute value being edited or a reference added/removed – more information on this can be found in Section 4.3.5.2).

4.3.2.2. Graph Layer

Extensive benchmarking showed that graph databases such as Neo4J and OrientDB perform significantly better than other technologies (e.g. relational databases) [34, 41] for the types of queries that are of interest to a system like Hawk; model queries are commonly comparable to graph traversal operations in contrast to being largely comprised numeric comparisons or other operations which would be amenable to other forms of storage. To avoid coupling with a specific graph database, this layer (the *IGraphDatabase* API) aims at providing a uniform interface for querying and manipulating graph databases in an implementation-independent manner. For example, key methods allow retrieving the persisted object (graph node) with a specific id in the store, creating a new node in the store, linking two nodes with a relationship between them and creating and accessing database indexes (used to enhance the query performance of certain classes of queries). It is worth noting that implementations of this layer can conceptually be used to connect to any back-end technology, but will suffer in performance if the data model is not similar with the graph model used here.

This layer consists of the following interfaces (seen in Figure 4.5):

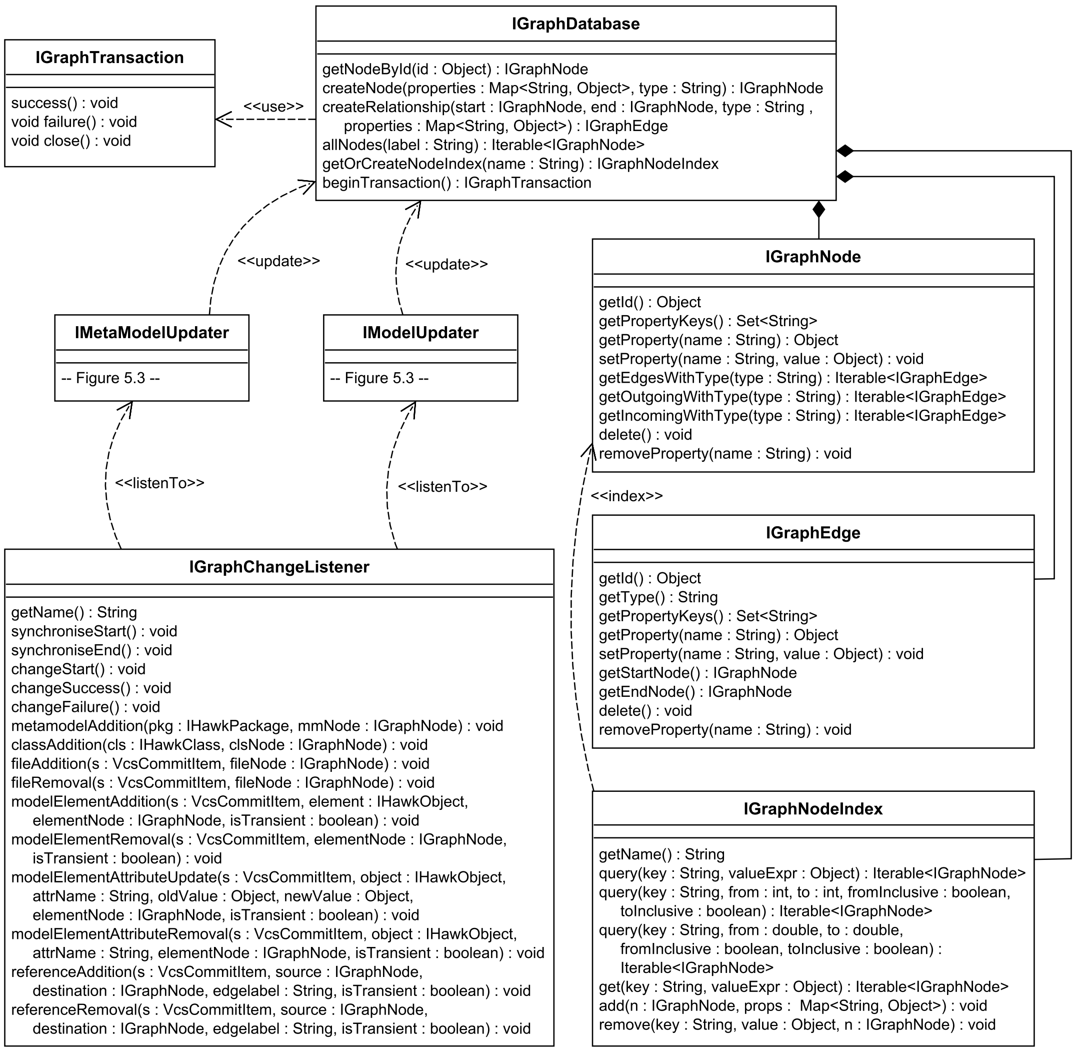

IGraphDatabase
getNodeById(id : Object) : IGraphNode
createNode(properties : Map<String, Object>, type : String) : IGraphNode
createRelationship(start : IGraphNode, end : IGraphNode, type : String , properties : Map<String, Object>) : IGraphEdge
allNodes(label : String) : Iterable<IGraphNode>
getOrCreateNodeIndex(name : String) : IGraphNodeIndex
beginTransaction() : IGraphTransaction
IGraphTransaction
success() : void
void failure() : void
void close() : void
<<use>>
<<update>>
<<update>>
IMetaModelUpdater
-- Figure 5.3 --
IModelUpdater
-- Figure 5.3 --
<<listenTo>>
<<listenTo>>
IGraphChangeListener
getName() : String
synchroniseStart() : void
synchroniseEnd() : void
changeStart() : void
changeSuccess() : void
changeFailure() : void
metamodelAddition(pkg : IHawkPackage, mmNode : IGraphNode) : void
classAddition(cls : IHawkClass, clsNode : IGraphNode) : void
fileAddition(s : VcsCommitItem, fileNode : IGraphNode) : void
fileRemoval(s : VcsCommitItem, fileNode : IGraphNode) : void
modelElementAddition(s : VcsCommitItem, element : IHawkObject, elementNode : IGraphNode, isTransient : boolean) : void
modelElementRemoval(s : VcsCommitItem, elementNode : IGraphNode, isTransient : boolean) : void
modelElementAttributeUpdate(s : VcsCommitItem, object : IHawkObject, attrName : String, oldValue : Object, newValue : Object, elementNode : IGraphNode, isTransient : boolean) : void
modelElementAttributeRemoval(s : VcsCommitItem, object : IHawkObject, attrName : String, elementNode : IGraphNode, isTransient : boolean) : void
referenceAddition(s : VcsCommitItem, source : IGraphNode, destination : IGraphNode, edgelabel : String, isTransient : boolean) : void
referenceRemoval(s : VcsCommitItem, source : IGraphNode, destination : IGraphNode, edgelabel : String, isTransient : boolean) : void
IGraphNode
getId() : Object
getPropertyKeys() : Set<String>
getProperty(name : String) : Object
setProperty(name : String, value : Object) : void
getEdgesWithType(type : String) : Iterable<IGraphEdge>
getOutgoingWithType(type : String) : Iterable<IGraphEdge>
getIncomingWithType(type : String) : Iterable<IGraphEdge>
delete() : void
removeProperty(name : String) : void
<<index>>
IGraphEdge
getId() : Object
getType() : String
getPropertyKeys() : Set<String>
getProperty(name : String) : Object
setProperty(name : String, value : Object) : void
getStartNode() : IGraphNode
getEndNode() : IGraphNode
delete() : void
removeProperty(name : String) : void
IGraphNodeIndex
getName() : String
query(key : String, valueExpr : Object) : Iterable<IGraphNode>
query(key : String, from : int, to : int, fromInclusive : boolean, toInclusive : boolean) : Iterable<IGraphNode>
query(key : String, from : double, to : double, fromInclusive : boolean, toInclusive : boolean) : Iterable<IGraphNode>
get(key : String, valueExpr : Object) : Iterable<IGraphNode>
add(n : IGraphNode, props : Map<String, Object>) : void
remove(key : String, value : Object, n : IGraphNode) : void

Figure 4.5.: The Hawk Graph Layer

- *IGraphDatabase* This interface represents the back-end persistence store used for Hawk. It offers methods for creating and getting nodes, relationships and database indexes.

- *IGraphNode* This interface represents a single graph node in an *IGraphDatabase*. It has a unique identifier and offers accessor methods for its properties and relationships (edges). Implementations should as much as possible keep these objects as lightweight as possible, with lazy loading of any features, as large quantities of them are meant to be passed around during program execution.

- *IGraphEdge* This interface represents a single relationship (edge) between two *IGraphNode*s. It has a unique identifier and offers methods for getting the nodes it is connected to (referred to as the start node and the end node) as well as accessor methods for its properties.

- *IGraphTransaction* This interface represents a transaction operation on the database. If supported, it should ensure that the usual ACID operations are enforced and that no read/write operations can be performed on the store outside transactions. Implementations that do not natively support transactions should create pseudo-transactions (not using the native store's API but a Java-based algorithm) as failure to encapsulate change in atomic events may cause irreversible inconsistency in the store. It is worth noting that for various NoSQL stores (such as Neo4J), their API can provide a *batch* mode whereby the database is rendered unavailable while some computationally demanding insert operation is taking place (in order to greatly speed up the execution time of this operation). In this mode, even though transactions are not supported, consistency is maintained as the database is only available before or after this mode is used so to a user it can be abstracted as a single "transaction" taking place (whereby no queries can be made while this "transaction" is being processed).

- *IGraphNodeIndex* This interface represents a database index consisting of a collection of nodes. The primary use of such a database index would be to be able to retrieve a certain class of nodes (such as types or files) without having to navigate the entire store using a blind search. It offers accessor methods for indexed nodes as well as query methods for retrieval.

- *IGraphChangeListener* This interface allows the forwarding of any changes made to the contents of Hawk to anyone listening, for example all the changes performed during a single synchronization process can be propagated to a client interested in them. This (simple yet effective) notification framework's goals are twofold: firstly it allows for implementation-specific handling of changes, allowing each listener the autonomy it may require; secondly it does not add any inherent overhead to Hawk if no one is listening as the update events will just be lost without having to be managed/stored in any way.

4.3.3. Version Control Managers

In order for Hawk to obtain the models it needs to index, as well as the updated model files every time a model is changed, it requires managers for the various version control systems it can monitor. As Hawk aims to be an orthogonal system to current versioning technologies used, it is not given any special privileges when communicating with the version control system (VCS) in which the original model files are stored. It is not able to configure or change the VCS in any way or have any different access privileges than the ones of the user connecting to the VCS directly.

As such, we assume that only the basic monitoring and fetching operations provided by file-based version control systems are available. As Hawk locally stores the current version of the files it is monitoring, it can easily retrieve only changed files from the VCS and analyze them for propagating changes. Hawk will poll its registered VCS repositories for changes according to its current update strategy heuristic (default settings as well as the synchronization strategy Hawk uses are found in Section 4.3.7) and can also be prompted to synchronize on demand through its interface.

4.3.3.1. SVN Manager

This component allows Apache Subversion (SVN) repositories to be monitored by Hawk. Every time Hawk performs a poll of an SVN manager the relevant SVN operations are called to retrieve all files which have changed between the current version stored in Hawk (possibly none if the model is not yet in Hawk) and the current HEAD version on the SVN. When the file identifiers of these changed files are obtained, Hawk then discards any files it does not need (non-model files or files of models Hawk cannot parse – this is achieved by comparing each file path to the accepted file extensions of each registered model resource factory) and passes the rest of them to their relevant model resource factories to be converted to in-memory resources and updated in Hawk, one at a time (to avoid using an unnecessarily large amount of memory).

4.3.3.2. LocalFolder Manager

A folder stored on the local disk of the Hawk server can be used as a bare-bone VCS, whereby a version of a file is defined by its numeric "last modified date", all major operating systems expose, with the obvious limitation that only the latest version of a file is available for retrieval. As version control systems commonly have a top-level repository version as well as individual file versions, we work under the assumption that the repository version (of a LocalFolder) changes if any of its contents change (any file has been added, deleted or edited).

This manager is of use if Hawk runs on a local computer with models not stored in a version control system as well as providing an easy way to test the other Hawk components (as manipulating files on a local folder requires very little effort).

4.3.3.3. Git Manager

As Git is a distributed version control system, it works under the assumption that each user is working with a clone of the repository available locally. As such, using a minor extension of the abovementioned *LocalFolder* driver, Hawk can monitor a local Git repository. This driver is optimized to ignore various paths in the repository (ie: the *.git* folder) to avoid any unnecessary overhead when crawling the file structure for discovering any new/changed/deleted files.

4.3.3.4. Workspace Manager

Hawk supports indexing models located in a local Eclipse Workspace. This driver is an extension of the *LocalFolder* driver (presented in Section 4.3.3.2) that can also use Eclipse's notifications[10] to inform Hawk whenever a model file has changed, instead of only relying on Hawk's periodic updates (presented in Section 4.3.7). As such, not only will Hawk be instantly notified whenever a model change has occurred (instead of waiting for its periodic update to occur), but it can also have the option of totally turning off its periodic updates and only rely on these notifications, improving its efficiency.

4.3.4. Metamodel/Model Resource Factories

Knowing which version control systems to monitor, Hawk now needs to know about which metamodels/models it is interested in so that it can index them. Resource factories are used to create in-memory representations (resources) for metamodels and models persisted on disk. The file(s) in question will have to either be parsed into Hawk resources directly, or parsed into their native in-memory representations and then converted into Hawk resources. These files need to be able to be loaded into memory in order to extract

[10] `http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_events.htm`

the necessary model information to be indexed, and it is assumed that each file can be loaded separately (either as an entire model or as a fragment of a larger model). Below we detail the various factories currently implemented.

4.3.4.1. EMF Resource Factories

These components allow for the parsing of EMF XMI-based metamodel and model files into in-memory Hawk resources.

Metamodel Resource Factory A metamodel in Hawk is defined as a uniquely identifiable collection of metaclasses. In the model layer of Hawk (Section 4.3.2.1) a metamodel is an *IHawkPackage* which has a unique global namespace identifier and contains a collection of *IHawkClass*es. As such, any metamodel file added to Hawk needs to first be converted into this in-memory representation before it can be registered to Hawk; all *IHawkPackage*s (and their contents) will be contained in an *IHawkMetaModelResource* which will then be used by Hawk to register the metamodel.

For EMF this conversion is relatively straightforward as it already offers an in-memory metamodel resource representation, containing *EPackage*s with *EClass*es. As such, the implementation of this component comprises producing wrappers around the EMF resource and its contents and exposing the relevant information needed, all of which is already contained in the resource. More specifically:

- EMFPackage (lightweight wrapper around EPackage) implements IHawkPackage – representing in-memory uniquely identifiable metamodels
- EMFClass (lightweight wrapper around EClass) implements IHawkClass – representing in-memory classes
- EMFReference (lightweight wrapper around EReference) implements IHawkReference – representing in-memory references
- EMFAttribute (lightweight wrapper around EAttribute) implements IHawkAttribute – representing in-memory attributes
- EMFDataType (lightweight wrapper around EDataType) implements IHawkDataType – representing in-memory data types

Model Resource Factory A model index in Hawk is defined as comprising information extracted from a collection of model elements, possibly originating in multiple files, that are defined by (instances of) a metamodel. An in-memory representation of a model is through *IHawkModelResource*s created by parsing the file-based persistence, and containing a collection of *IHawkObject*s. Each such object has a type (the *IHawkClass* it is an instance of) as well as values (possibly unset) for each feature (attribute/reference) of its type (and supertypes).

For EMF this conversion is relatively straightforward as it already offers an in-memory model resource representation, containing *EObject*s with values for their relevant features. As such, the implementation of this component comprises producing wrappers around the EMF resource and its contents (the *EObject*s).

It is worth noting that due to the fact that Hawk does not need any external information other than the XMI model file (and its respective metamodel – identified by its unique namespace – to be registered beforehand), it does not matter which modeling environment the models come from as the resulting models files should be indistinguishable to Hawk.

BPMN Resource Factories A plugin for allowing BPMN models using the Eclipse BPMN Modeler Tool[11] to be indexed into Hawk was created, using the abovementioned EMF resource factories and adding the relevant BPMN metamodels using the *getStaticMetamodels()* method provided by Hawk's *IMetaModelResourceFactory* API (Figure 4.2).

4.3.4.2. Other Resource Factories

Two other resource factories were created for Hawk, in order to allow it to index other technologies than basic EMF. This work uses the aforementioned EMF factories as a blueprint and was done in collaboration with Dr. Seyyed Shah (research associate of the MONDO project) and Dr. Antonio Garcia Dominguez (also a research associate of MONDO). The drivers have since then been kept up to date in order to maintain their architectural alignment with Hawk (as the tool evolved), as part of this work.

Modelio Resource Factories These factories are used to parse UML models created and maintained using the Modelio tool presented in Section 2.2.2.2. These factories

[11] `https://www.eclipse.org/bpmn2-modeler/`

assume that only the freely available open-source modules of Modelio are used, and does not depend on any of the commercial modules.

Metamodel Resource Factory This factory is a minor extension of Hawk's EMF metamodel resource factory as Modelio uses EMF as its in-memory metamodel and model representation format.

As Modelio is a tool that only works on specific languages like UML and BPMN, it uses only specific metamodels (such as that of a specific version of UML – currently using the UML 2.2 specification). Such metamodels are provided to Hawk through the *getStaticMetamodels()* method provided by Hawk's *IMetaModelResourceFactory* API (Figure 4.2).

Model Resource Factory This factory is used to create *IHawkModelResource*s from projects created by the Modelio tool. Such projects are similar in structure to the ones presented in the Modelio SVN-based repository with folders for each UML type (and files inside them for each instance of this type in the model), as well as a variety of other metadata items Modelio needs for its various components.

In order to parse a persisted Modelio UML model project into an in-memory resource the Modelio tool parser is used. This parser requires that all files in the model project are available and consequently uses them to load the entire model into memory as an EMF resource. Hawk then uses this resource to create an *IHawkModelResource*, similarly to how the Hawk EMF model resource factory handles other EMF model resources.

As Hawk usually assumes models are collections of independent files that can be loaded separately while the Modelio parser requires all files to be loaded into memory before providing a model resource, each Modelio model project is assumed to be converted into a single (*.zip*) archive before being versioned in a VCS, to be then indexed by Hawk[12]. This approach to versioning non-file-based model persistence formats (such as database persistence or Modelio projects or any other form of persistence which requires a specific file structure) is recommended by various tools such as Modelio and MagicDraw[13] as it ensures that any version of the model can be read with no errors and is self-contained. On the other hand this results in the whole model structure needing to be retrieved

[12] in principle, a driver which loads each Modelio model file separately can be developed in order to overcome this limitation, as the various files are structured similarly to fragmented XMI EMF models discussed in Section 2.2.1.1

[13] `http://www.metacase.com/support/45/documentation/versioning.html`

every time any change happens inside it; this is not detrimental to the performance of Modelio models as the parser Modelio uses reads all its model fragments each time it loads it into memory (similar to reading a single XMI file in EMF) so a lazy-loading approach is not possible out-of-the-box.

Finally, due to various limitations in how Modelio treats the in-memory models, in order for the plugin to work, we require that one of the Modelio plugins exports one of its packages (that it does not normally export). Specifically we require that the plugin *org.modelio.xmi* exports the *org.modelio.xmi.generation* package (by adding "Export-Package: org.modelio.xmi.generation" to its manifest).

It is worth noting that a Modelio plugin using the commercial version of Modelio would be able to leverage the native Teamwork Manager Module Modelio offers (presented in Section 2.2.2.2) and its resulting repository. In such a case, creating a custom lazy-loading parser could be more performant as it would be able to parse the model one fragment at a time and hence only load changed fragments upon model evolution.

IFC Resource Factories This factory is used to create *IHawkModelResource*s from models using the IFC BIM metamodel. An EMF-based parser for IFC models is available from the OpenBIM[14] project. OpenBIM is an open-source toolkit for developing IFC-based BIM models; such models represent "physical and functional characteristics of a facility"[15].

Metamodel Resource Factory The metamodels required to load such IFC models are currently Ifc2x3tc1 and Ifc4[16] that represent the latest two versions of the EXPRESS schema used to define IFC models. These metamodels are provided to Hawk through the *getStaticMetamodels()* method provided by Hawk's *IMetaModelResourceFactory* API (Figure 4.2)).

Model Resource Factory IFC models are stored in either an XML (ISO10303-28 XML representation of EXPRESS schemas and data) or a STEP (ISO10303-21 physical file structure) format. The OpenBIM parser can use either of these formats to create an in-memory EMF model resource out of an IFC model file. Hawk then uses this resource

[14] `http://www.buildingsmart-tech.org/`
[15] `http://www.buildingsmart.org/standards/technical-vision/`
[16] `http://www.buildingsmart-tech.org/specifications/ifc-releases/summary`

to create an *IHawkModelResource*, similarly to how the Hawk EMF model resource factory handles other EMF model resources.

4.3.5. Metamodel/Model Updaters

Now that Hawk knows where to find file changes and can convert relevant model files into in-memory resources, it needs to be able to efficiently update its model index every time a file is changed (added/removed/edited). In order to achieve this it uses updaters to propagate changes and achieve synchronization.

4.3.5.1. Metamodel Updater

Every time a new metamodel file (or collection of metamodel files) is added to Hawk, they are firstly converted into resources through their appropriate factory, and then these resources are processed by Hawk to be registered to the model index. Every *IHawkPackage* in the resource(s) is a unique metamodel in Hawk, identified by its unique namespace URI, and containing its *IHawkClass*es. Hawk's Graph Layer (Section 4.3.2.2) is used to insert this into Hawk's back-end, which then exposes its knowledge of the new metamodel.

Before this operation is finalized, a consistency check is performed to ensure that the metamodel insertion is self-contained. This check succeeds only if every *IHawkClass* in every *IHawkPackage* in the current metamodel insertion does not have any proxy dependencies. A proxy dependency in this context is any reference the *IHawkClass* may have to another file (such as a superclass/attribute/reference/datatype being only a proxy to another file), which has not already been inserted into Hawk.

This ensures that any metamodel insertion is complete, to avoid partial insertion of metamodels, which would potentially cause failures when attempting to insert models conforming to these (incomplete) metamodels. For example if a collection of interconnected metamodels is found in multiple files, all of the files will have to be inserted into Hawk simultaneously and not one at a time, in order to allow Hawk to ensure a complete metamodel exists in its model index.

This strategy is in contrast to model insertion and updating whereby references to proxy elements are not an issue (as nothing will directly depend on the models) and can be maintained on the fly as new files are added/removed/updated for the models. An important factor to consider is that metamodels are very commonly of negligible size (when compared to the size of models this tool aims at handling), hence the possible performance loss of this extra constraint is seen as very minor when compared to the benefits it provides.

4.3.5.2. Model Updater

When using the default model updater[17], Hawk performs Algorithm 1 every time it finds a changed (added, removed, updated) model file in any of its monitored repositories. This algorithm inserts new models to the index and uses a threshold value (defaulting to accepting up to 50% model element changes – additions/removals/updates) to determine whether it should use a naive or incremental strategy for updating currently indexed models to their new versions. This threshold aims at compromising the memory overhead of an incremental update (on a very large number of changes) with the execution time overhead of a batch update (on a very small number of changes); hence a developer aiming to adjust this to a different value should raise the value if the memory use of Hawk is less important than the execution time of an update and vice versa.

As demonstrated in the Section 5.3.2, using this incremental updating when a model file is already indexed often provides a large performance gain when compared to naively deleting and re-indexing it every time it is modified.

Insertion For the insertion of a model file into Hawk, a process outlined in Algorithm 2 is followed. In this process, the elements of the model file are firstly loaded into memory as a *model resource*. Then, for each such element a node is created in the model index graph with its attributes as properties, and linked (using relationships) to its file and type/supertypes. Finally, for each element its references are used to link the node with other nodes in the graph.

As this process often requires intense resource consumption, the *batch* mode of Hawk's back-end is used (if the specific back-end used supports it). This mode makes the database unavailable until the process is completed, but using Hawk's Neo4J driver has around an order of magnitude better performance in terms of execution time when

[17] as Hawk's architecture allows for pluggable updaters to be used, this discussion is based on using the one constructed in the context of this work

[18] as an incremental update of a large number of elements can potentially be resource consuming, a threshold can be provided to direct any update with too many changes to use naive insertion instead in order to lower the required memory and potentially improve its execution time

Algorithm 1: Hawk update overview

```
1  if model file already indexed then
2      if change of type added/updated then
3          if model change size less than threshold[18] then
4              incremental update
5          else
6              naive update
7          end
8      else if change of type removed then
9          delete indexed elements of file, keeping cross-file reference proxies
10     end
11 else
12     if change of type added/updated then
13         insertion (indexing of model elements)
14     end
15 end
```

compared to on-line (transactional) updating, and using the OrientDB driver is around two times as fast.

Naive Updating A naive update strategy for Hawk simply removes the current indexed version of the model and the re-inserts it from scratch. While this approach can be seen as ineffective (especially if the back-end is not optimized for large numbers of deletions of elements), mainly when dealing with very small numbers of changes, it avoids any overhead needed to perform an incremental update and is thusly suitable when a large numbers of changes is found in the model.

Signature Calculation In order to update a model, an efficient way to determine whether a model element has changed is needed. A signature of a model element is a lightweight proxy to its current state. In order to calculate a meaningful signature for model elements indexed in Hawk (in order to enable support for incremental updates of the model index, as models in it evolve), every mutable feature of the element needs to be accounted for. As such, the following features are used to calculate the signature of

Algorithm 2: Insertion algorithm

```
1  use relevant factory to parse the file into a model resource
2  foreach element in the model resource do
3      create model element node in graph and set its attributes
4      create signature attribute in node
5      create a relationship from this node to its file node
6      create a relationship from this node to its type node (and relationships to
         all its supertype nodes)
7  end
8  foreach element in the model resource do
9      foreach reference in the references of the element do
10         if reference of element is set then
11             foreach referenced element do
12                 if referenced element is not a proxy then
13                     create relationship from this node to the node of the referenced
                         element
14                 else
15                     add new proxy reference
16                 end
17             end
18         end
19     end
20 end
```

each element:

- all of the names and values of its attributes
- all of the names and the IDs (of the target elements) of its references

This works under the assumption that model elements cannot be re-typed during model evolution, which is the case for the popular modeling technologies such as EMF, as well as that model elements have immutable and unique IDs.

A signature can be represented as either a String containing the concatenation of the values listed above or as a message digest (also known as a hash)[19] of this String. The String representation ensures that a unique signature exists for any model state, but suffers in terms of comparison performance as potentially very long Strings will have to be compared. On the other hand, a digest representation allows for rapid comparisons but has a chance (albeit small) for clashes, which show different model states as having the same signature. We decided to use the a SHA-1[20] digest representation as this identifier, to allow rapid comparisons. This signature is used to efficiently find changes in model elements, as detailed below.

Incremental Updating For incremental updating of a model file into Hawk, the process outlined in Algorithm 3 is followed. In this process the signatures of each element are used to efficiently determine which elements have changed. Then, for each new element a node is created, for each changed element its properties and relevant references are updated (keeping dangling cross-file references as proxies in Hawk for consistency), and for each removed element its node is deleted. The complexity of this algorithm is $\mathcal{O}(m + n + c \times avgR)$ where m is the number of model elements in the updated model file, n is the number of model nodes in the model index linked to the (previous version of the) updated file, c is the number of changed elements and $avgR$ is the average number of target elements referenced by all of the changed elements.

This process only alters the part of the model index which has actually changed and as such, it does not need to use more resources than required by the magnitude of the change, potentially saving on memory and execution time.

Incremental Update Example In order to clearly demonstrate how incremental updating works, a simple example is presented here using the BPMN loan running example. In this scenario, assume that the following changes have been made to the original BPMN loan model in Figure 2.7 (the new model is seen in Figure 4.6):

1. The attribute *name* of Task with id *t2* is changed from "Check Applicant Info" to "Verify Applicant Information Authenticity"
2. A new Task with *name* "Notify Authorities of Fraud" is introduced

[19] one-way hash function that takes arbitrary-sized data and outputs a fixed-length hash value

[20] a cryptographic hash function designed by the United States National Security Agency; it produces a 20 byte value

Algorithm 3: Incremental update algorithm

```
1  let nodes be the set containing the ids and pointers to all the nodes (in the model
     index – linked with the updated file)
2  let signatures be the set containing the ids and signatures of the nodes
3  let delta be the set containing changed elements
4  let added be the set containing new elements (to be added to the model index)
5  let unchanged be the set containing elements which are the same
6  foreach node from all nodes in the model index linked with the updated file do
7      add node to nodes
8      add signature of node to signatures
9  end
10 foreach element in elements of model resource do
11     if element id exists in signatures then
12         if element signature not equal to current signature then
13             add element to delta
14         else
15             add element to unchanged
16         end
17     else
18         add element to added
19     end
20 end
   /* add new nodes to model index                                  */
21 foreach element in added do
22     add this new element in model file to model index
23 end
   /* delete obsolete nodes and change node attributes              */
24 foreach node in nodes do
25     if node id exists in delta then
26         retrieve element in delta represented by the node
27         foreach model attribute in element do
28             if attribute value is different to node property value then
29                 update property value
30             end
31         end
32     else if node id does not exist in unchanged then
33         de-reference node (keeping dangling incoming cross-file reference proxies)
34         delete node
35     end
36 end
```

Algorithm 3(cont.): Incremental update algorithm (cont.)

```
   /* change altered references                                        */
37 foreach element in delta do
38     foreach reference in references of element do
39         if reference is set then
40             foreach referenced element in referenced elements of reference do
41                 if referenced element is not proxy then
42                     add id of referenced element to targetIds
43                 else
44                     add new proxy reference to model index
45                 end
46             end
47             foreach relationship in relationships of node linked with the element
               do
48                 if relationship target has id which exists in targetIds then
49                     remove target from targetIds
50                 else
51                     delete relationship as new model does not have it
52                 end
53             end
54             foreach id in targetIds do
55                 add new relationship to model index
56             end
57         else
58             foreach relationship in relationships of node, with the same name as
               the reference name do
59                 delete this relationship
60             end
61         end
62     end
63 end
```

3. The *reject* branch of the Gateway with *name* “Result of Verification” now points to the new Task with *name* “Notify Authorities of Fraud”. This is achieved by deleting the old SequenceFlow it had to the EndEvent and introducing a new one to this new Task.

4. A new SequenceFlow is introduced from the new Task with *name* “Notify Author-

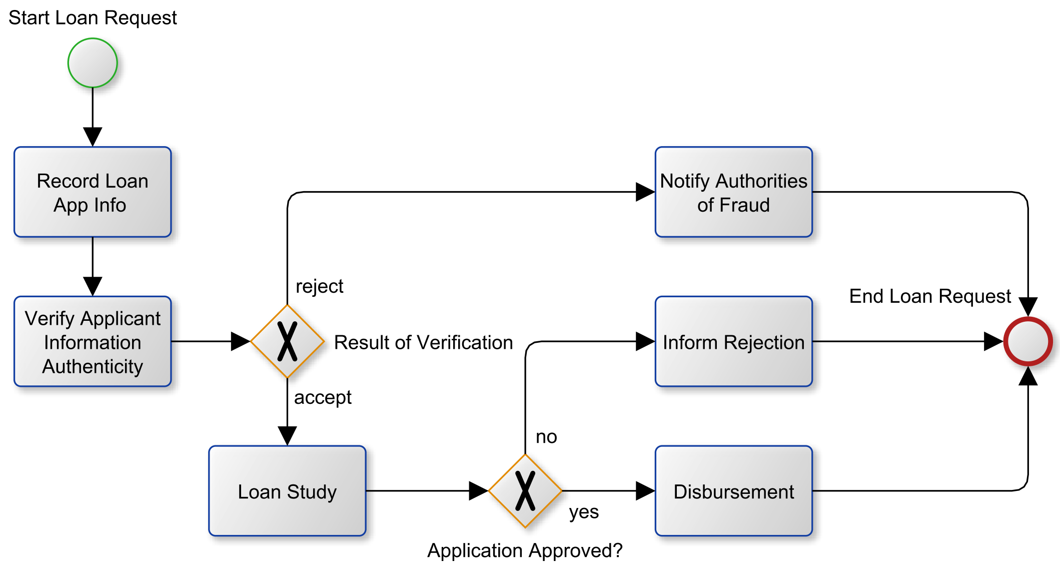


Figure 4.6.: Running example – altered version of the BPMN Loan model

ities of Fraud" to the EndEvent of the model.

In order to synchronize Hawk with the new version of the model the following would occur (all line references are to the incremental update Algorithm 3):

- Line 7 – *nodes* would include the nodes representing the old version of the model indexed in Hawk
- Line 13 – *delta* would include the Task whose name was changed from "Check Applicant Info" to "Verify Applicant Information Authenticity", as its signature has changed. It would also include the Gateway named "Result of Verification" as well as the EndEvent named "End Loan Request" as in both cases their references have changed and hence so has their signature.
- Line 15 – *unchanged* would include the remaining model elements that are found in both versions of the model, as their features are the same and hence so is their signature.
- Line 18 – *added* would include the new Task named "Notify Authorities of Fraud" as well as the two new SequenceFlows introduced, one between the Gateway named

"Result of Verification" and the new Task, and one between the new Task and the EndEvent.

- Line 22 – the three aforementioned added nodes would be created
- Line 29 – the property "name" of the Task in *delta* will be updated to its new value
- Line 34 – the SequenceFlow which used to join the Gateway named "Result of Verification" and the EndEvent named "End Loan Request" is deleted as it is not found in the new model
- Line 51 – the reference *outFlows* of the Gateway named "Result of Verification" is updated to not include the deleted SequenceFlow. Furthermore the reference *inFlows* of the EndEvent will be updated to not include the deleted SequenceFlow.
- Line 55 – the reference *outFlows* of the Gateway named "Result of Verification" is updated to include the new SequenceFlow between the Gateway and the new Task. Furthermore the reference *inFlows* of the EndEvent will be updated to include the new SequenceFlow between the new Task and the EndEvent.

Proxy Reference Resolution A reference to a model element located in another file (or location) is referred to as a proxy reference, whereby only the location of the target object(s) is recorded as opposed to the objects themselves. As Hawk indexes models that can be potentially fragmented (i.e. use proxy cross-file references), it uses the following procedure to manage any cross-file references:

- Proxy reference addition. Whenever a model element (either during insertion or update) has a proxy reference, this information is stored in a proxy reference database index. This reference contains its name and the location (file path and identifier) of the model element(s) it is referencing.
- Proxy reference resolution. During Hawk's synchronization procedure, after each changed model file is handled, any resolvable proxy references are resolved. A proxy is resolvable if its target element is found in Hawk.
- Proxy reference update. During each update of a model element, any previous proxies it had are purged, so that only proxies of the latest version of this element are recorded.

- Reference proxification. Any time an element of Hawk is deleted, any incoming references to that element from elements originating in a file different than the file the element itself originated from, are proxified[21]. Instead of deleting these references and (incorrectly) losing all information about them, instead Hawk keeps them as proxies, using the method described above. As such, consistency is maintained should the deleted element be re-inserted into Hawk at a later time.

4.3.6. Querying Hawk

As Hawk now contains an up-to-date global index of all relevant models in its monitored VCSs, for it to be of practical value, Hawk needs to be able to provide correct and efficient responses to queries made on its model indexes. There are two principal ways of querying a model index:

4.3.6.1. Native Querying

The most straightforward, and arguably the most performant, ways of querying a model index is using the native API of its persistence back-end, or the data-level query language (such as SQL if a relational database is used or Java/Cypher if a Neo4J NoSQL database is used). Nevertheless, it also demonstrates certain shortcomings which should be considered:

- *Query Conciseness* Native queries can be particularly verbose and, consequently, difficult to write, understand and maintain. Examples of such queries can be found in Section 5.2.1.1.
- *Query Abstraction Level* Native queries are bound to the specific technology used; they have to be engineered for that technology and cannot be used for a different back-end without substantial alteration in most cases.

4.3.6.2. Back-end Independent Navigation and Querying

An alternative way to access and query models is through higher-level query languages that are independent of the persistence mechanism. Examples of such languages include

[21] a new proxy reference is added for each such element and stored in the proxy reference database index (as mentioned above in proxy reference addition).

the Object Constraint Language (OCL), the Epsilon Object Language (EOL – from the Epsilon [68] platform) and the Atlas Transformation Language (ATL), which abstract over concrete model representation and persistence technologies using intermediate layers such as the *OCL pivot metamodel* [69] and *Epsilon Model Connectivity* [70] layer.

In terms of execution, queries expressed in such high-level languages can be executed on an in-memory representation of the model, or translated into queries expressed in persistence-level query languages such as SQL and XQuery[22], at compile-time or at run-time. Full translation is only feasible in cases where the high-level and the lower-level query languages are isomorphic in terms of capabilities. This is not always the case: for example, EOL supports dynamic dispatch which is not supported in SQL. Even when full compile-time translation is not feasible, partial translation at run-time has been shown to deliver significant performance improvements as seen in [71].

EOL Query Engine – Epsilon and The Epsilon Model Connectivity Layer (EMC) The Epsilon platform [68] is an extensible family of languages for common model management tasks and includes tailored languages for tasks such as model-to-text transformation (EGL), model-to-model transformation (ETL), model refactoring (EWL), comparison (ECL), validation (EVL), migration (Flock), merging (EML) and pattern matching (EPL). All task-specific languages in Epsilon build on top of a core expression language – the Epsilon Object Language (EOL) – to eliminate duplication and enhance consistency. As Epsilon provides a back-end independent language (EOL), we decided to use it in this work.

As seen in Figure 4.7, EOL – and as such all languages that build on top of it – is not bound to a particular metamodeling architecture or model persistence technology. Instead, an intermediate layer – the Epsilon Model Connectivity layer – was introduced to allow for seamless integration of any modeling back-end.

Epsilon uses a driver-based approach for its EMC layer, where integration with a particular modeling technology is achieved by implementing a *driver* that conforms to a Java interface (*IModel*) provided by EMC. For a more detailed discussion on EMC and the *IModel* interface, the reader can refer to Chapter 3 of [70].

Hawk provides an EMC driver (implementation of the IModel interface) to allow Epsilon to use it as a model provider (and consequently query it using EOL expressions). A simplified class diagram of the relevant classes in Epsilon and Hawk is seen in Figure

[22] `http://journal.ub.tu-berlin.de/eceasst/article/viewFile/108/103`

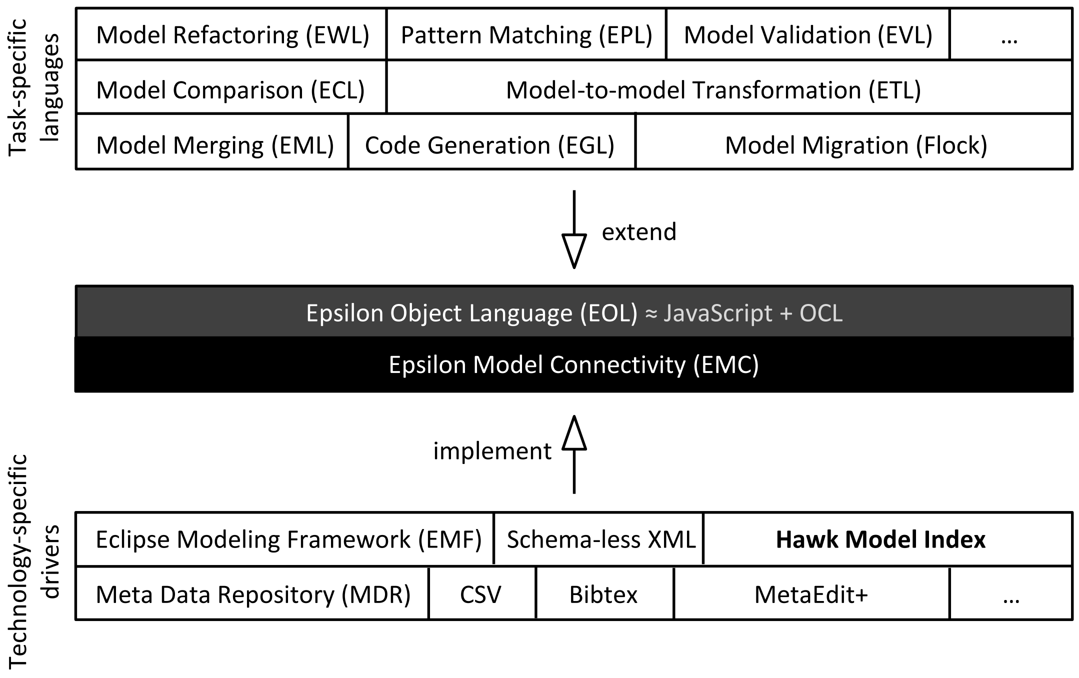


Figure 4.7.: The Epsilon Model Connectivity Layer

4.8. Below we detail the additions made to this driver in order to incorporate the use of custom database indexes for improving query performance.

IModel interface method implementations In order to use Epsilon's EOL to query model indexes stored in Hawk, an implementation of the IModel interface is required. In Table 4.2 we present a description of various methods of interest in the IModel interface and a summary of their implementation details in Hawk. Note that any model element loaded into memory is of Java class *GraphNodeWrapper*. This is a lightweight object which contains only the location of the relevant model element in the store (its 'id' value for example in a Neo4J NoSQL Graph database) as well as a reference to the Epsilon model it is part of; this object can be used to load the element's attributes and relationships on demand.

Reverse reference navigation In the spirit of EMF's *eContainer()* method which allows an *EObject* to get access to its container object, Hawk provides a mechanism for reverse-navigating a containment reference in order to access the container. This feature is embedded into the parser by means of prefixing the relevant reference with "revRefNav_".

Table 4.2.: Interesting methods in the IModel interface

Method	Return Type	Description
allContents()	Collection<?>	Returns a collection containing all of the nodes contained in the model index in the form of *GraphNodeWrapper*s
hasType(String type)	boolean	Returns whether the type *type* exists in the model index by trying to find it through the *Metamodel* (database) index of the store.
getAllOfType(String type)	Collection<?>	Returns a collection containing all of the objects of type *type* in the model index by first invoking hasType(type) and, if successful, finding the type using the *Metamodel* (database) index and then creating a collection of *GraphNodeWrapper*s containing every element which has an *ofType* relationship to *type*.
getTypeOf(Object instance)	Object	Returns the type node of the element *instance* in the model index by directly accessing the node *instance* (as this method is always passed a *GraphNodeWrapper* as the *instance*) and navigating its *ofType* relationship to get the type node. The returned object is a *GraphNodeWrapper*.
isOfType(Object instance, String type)	boolean	Returns whether the node *instance* in this model is of type *type* by first invoking hasType(type) and, if successful, invoking getTypeOf(instance) and performing a String comparison on the resulting names. The *type* can be either fully qualified, including the unique identifier of the metamodel it is contained in, or only the type name.
getPropertyGetter()	IPropertyGetter	Returns the *GraphPropertyGetter* associated with the Hawk repository. The *GraphPropertyGetter* uses Hawk's *IGraphDatabase* API to retrieve properties from objects in Hawk.

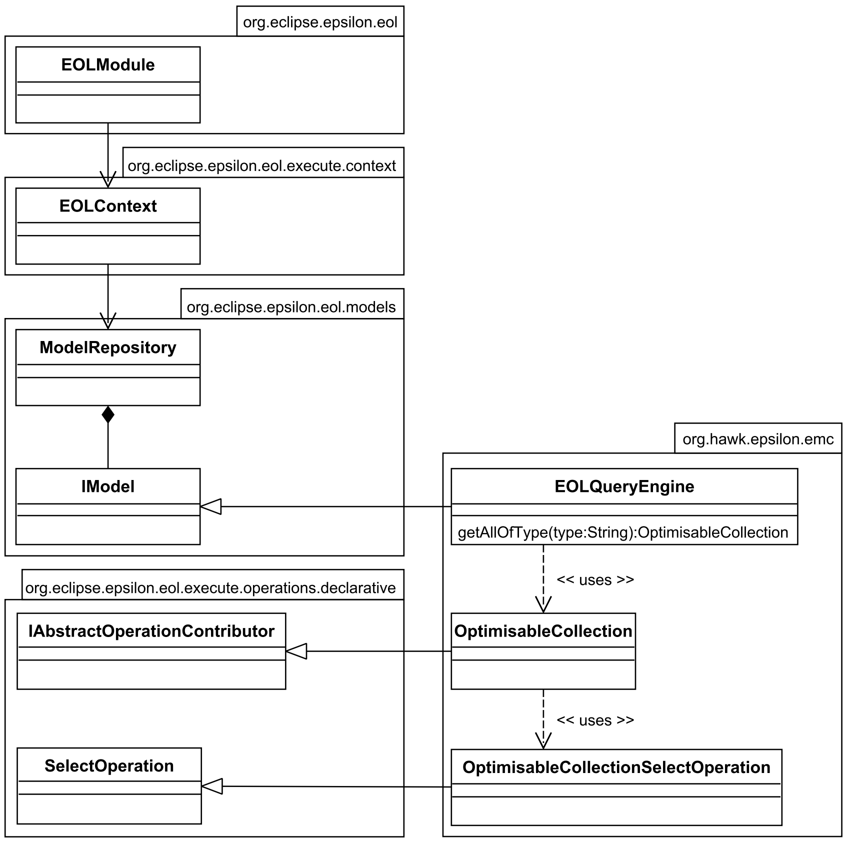

org.eclipse.epsilon.eol
EOLModule
org.eclipse.epsilon.eol.execute.context
EOLContext
org.eclipse.epsilon.eol.models
ModelRepository
IModel
org.hawk.epsilon.emc
EOLQueryEngine
getAllOfType(type:String):OptimisableCollection
<< uses >>
OptimisableCollection
<< uses >>
OptimisableCollectionSelectOperation
org.eclipse.epsilon.eol.execute.operations.declarative
IAbstractOperationContributor
SelectOperation

Figure 4.8.: Important Hawk EMC layer classes and their EOL parents

For example, say one has an object 'A' with a containment reference called 'contain' to an object 'B'. Then, by typing "B.revRefNav_contain" in EOL, we get as a result object A. To increase usability of this feature the keyword "eContainer" is also reserved, so that references of this name return the container object as well (if one exists).

Scoped Queries Hawk supports the use of scoped queries with respect to the file of origin of the model elements involved. As discussed in Appendix C and demonstrated in Figure C.9, should the modeler want to limit the query to elements found in a specific file or set of files, this can be done by providing a relevant file inclusion pattern to Hawk. This allows Hawk to limit its search to elements contained in only a specific subset of its contents. The syntax used is that of a regular expression inclusion pattern for the files the developer wishes to include (or empty for performing a global query).

This allows queries to be performed, as if the rest of the contents of the contents of the model index was not present, for example to obtain results from:

- A specific or a set of specific files, say "model1.xmi, model2.xmi".
- One type of file, such as "*.xmi" or "class_diagram_*.uml".
- A specific directory in the monitored version control system, say "/london_bridge/schematics/*.xmi".

This provides flexibility in the way we can query Hawk, providing a way to expose the file path itself as a way to manipulate retrieving information from Hawk. Similarly, repository locations can also be scoped in the same way to the one presented above, allowing for results originating in one or more repositories to be returned.

It is worth noting that for EOL, Hawk offers two alternative approaches to scoping queries: The first one limits the initial search terms of the query to contain only elements originating in repositories/files within the defined scope; this does not necessarily limit the traversal paths the query takes to get to the results. More specifically, for the *EOLQueryEngine* implementation, the filtering to scoped elements is performed for each global API call in *IModel* (such as *allContents()*, *getAllOfType(...)* or *getAllOfKind(...)*); as EOL allows for various logical operations to be performed (such as *collect* – that may result in navigating to elements originating in other files) as well as permitting imperative code to be run, this approach does not guarantee that the query will only traverse scoped elements during its execution. This approach allows a modeler to query Hawk scoping only the initial terms of the query, with the overhead and its subsequent impact on query performance being very small as the scoping is only done initially (and not at every single element or feature access).

The second approach limits the entire traversal path of the query to the scope provided. This introduces the overhead of having to check that every feature call to a target model element is restricted to the scope but guarantees that the query is truly scoped within the parameters provided from start to finish.

For example, assume that instead of being in a single file, the BPMN loan example is split into two files, as shown in Figure 4.9 (the dotted lines represent a proxy reference to a model element in another file – in this case to the three *SequenceFlow* model elements in file "loan_end.xmi"). Here, running an EOL query for finding out how many outgoing *SequenceFlows* there are in the *BaseElement*s of the loan model:

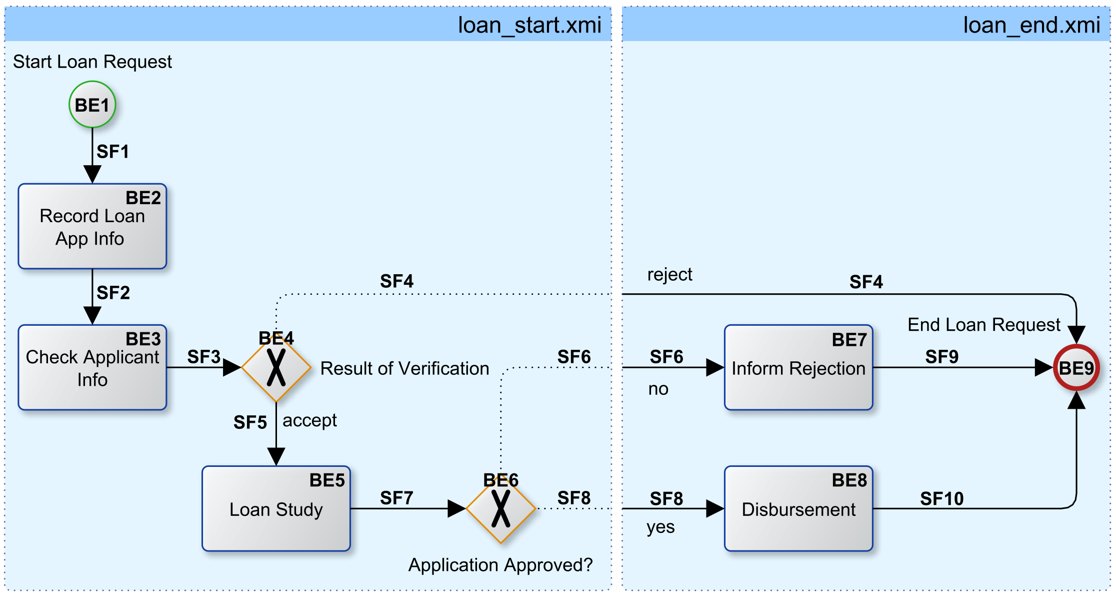


Figure 4.9.: Fragmented BPMN loan model

```
return BaseElement.all.outFlows.flatten.size();
```

Scoped under file "loan_start.xmi", will produce different results depending on whether scoped querying is set to initial term or to full traversal scoping[23] (returning 8 for initial term and 5 for full traversal, detailed below):

- Whether full traversal scoping is enabled or not, the initial terms of the query (*BaseElement.all*) will match the 6 *BaseElement*s in file "loan_start.xmi" (labeled BE1–BE6 in Figure 4.9).
- If full traversal scoping is not enabled, the next part of the query (*.outflows*) will match all target elements of the *.outflows* reference of the abovementioned *BaseElement*s (i.e. *SequenceFlow*s labeled SF1–SF8 in Figure 4.9). As such the query will return 8 (after flattening and counting the results).
- If full traversal scoping is enabled, the next part of the query (*.outflows*) will not match *SequenceFlows* labeled SF4, SF6 and SF8 as they are proxies to model

[23] initial term scoping will only limit model elements found in the scoped files for the first collection call (*.all* / *.allOfType* / *.allOfKind*) of the query, full traversal scoping will only include model elements in the scoped files regardless of when they are accessed in the query (omitting any other model element).

elements in file "loan_end.xmi" and hence are outside the scope provided. As such the query will return 5 (after flattening and counting the results).

4.3.7. System Lifecycle

For Hawk to be able to operate over a continuous period of time, its lifecycle needs to be managed. Hawk provides the following functionality during its normal execution:

- ***Initialization and Addition/Removal of Model Indexes***.
  - Upon initialization of Hawk, any previously created (saved) model indexes are loaded and synchronized with any changes made to their relevant repositories.
  - A model index can be added at any time, and linked to a set of version control systems. This will create the model index and populate it with any relevant model files found in the VCS(s) linked to it.
  - A model index can be removed; there are two types of removal: temporary and permanent. Temporary removal only disables updates to the model index from the VCS (equivalent to disconnecting a folder from a VCS) and can be used if the model index may be of use again in the future. A permanent deletion removes the model index itself from the hard disk as well as any metadata kept about it.
- ***Scheduling / Maintenance operations***.
  - Hawk will periodically synchronize any model indexes with their VCS(s). Timers are used to initiate synchronization requests for each model index and the algorithm used to determine the frequency of these updates is configurable. The default uses the following algorithm (time is in seconds):

    ```
    if ($changes == false$)
        if ($timeToUpdate < 512$)
            $timeToUpdate = timeToUpdate \times 2$
    else
        $timeToUpdate = 1$
    ```

    Where *changes* is a boolean returned by the synchronize method informing the server if there are any changes to the versioned files with respect to the versions in the model index. As shown, the time between checks increases as long as there are no changes and resets to 1 when there is a change.

  - If changes are found in files of interest (i.e. models Hawk can parse), the contents of these files are retrieved and parsed, and the model index is updated accordingly. Further details of the different events that can occur during an update are provided below.
  - If an update results in a model being inconsistent (for example by having references to unresolvable proxies introduced to it due to the deletion of a model fragment file in the VCS), the model index is updated, keeping any unresolved proxies flagged and attempting to resolve them in future updates.

- ***Other runtime operations.***
  - Metamodels can be added to a Hawk model index, which will allow more types of models to be indexed by Hawk. Metamodels are provided as one or more files.
  - Derived attributes can be added to a Hawk model index and any existing attribute can be marked as (database) indexed for enhancing query performance on operations needing to traverse them.
  - Any running Hawk model index can be queried. Depending on the query engine used, the results may be returned to the engine as objects the engine itself understands, or simply as console output.

- ***Shutdown operations.***
  - Under normal shutdown, Hawk finishes any currently running tasks (such as pending updates), saves the meta information of the model indexes currently running (such as the paths to their relevant VCS), and then terminates.
  - The responsibility for coping with abnormal shutdowns (for example power failure in the middle of an update in one of the model indexes in the indexer) is delegated to the back-end used to persist the model index (such as the NoSQL database). The indexer itself will not lose any critical information in case of an abnormal shutdown as it does not keep any volatile critical data in memory; any actions being performed during the failure will be re-initialized

upon the next startup of the system (as the model index will not be updated so will provide the same metadata as previously to the indexer).

4.3.7.1. Synchronization Procedure

If a set of files have changed in a VCS linked to a model index in Hawk (updated/added/removed/renamed/moved), it will perform Algorithm 4 in order to ensure synchronization, where:

- *existsInteresting()*: Returns true if at least one of the changed files is interesting to Hawk. Being interested in a file denotes that this file should be parsed and indexed.
- *modelFiles()*: This method returns a set of file-revision tuples of model files from the changed file set.
- *added(X)*: Returns true iff file X was added to the VCS.
- *removed(X)*: Returns true iff file X was removed from the VCS.
- *updated(X)*: Returns true iff file X was updated in the VCS.
- *renamed(X)*: Returns true iff file X was renamed in the VCS.
- *moved(X)*: Returns true iff file X was moved in the VCS from one folder to another.
- *resolveModelProxies()*: This method resolves any unresolved cross-model references that have been kept as proxies when parsing the files (as only one file is open at a time for performance reasons).
- *scheduleNextUpdate()*: Schedules the next update for this repository, based on the heuristic discussed above.

Relevant addition/update and deletion procedures mentioned in the algorithm, as well as metamodel-level operations, are discussed below:

Metamodel addition The procedure for parsing a metamodel file into a resource is dependent on the file type and is described by its relevant driver. An example using XMI as input is seen in Appendix C. As mentioned above, regardless of input, an *IHawkMetaModelResource* is created from the file and is ready to be parsed for database insertion.

Algorithm 4: Synchronization procedure

```
1  if existsInteresting() then
2      for modelFile m : modelFiles() do
3          if added(m) then
4              Parse the model file into a resource. Insert it into the index (see
               Section 4.3.5.2).
5          else if removed(m) then
6              Delete the model from the index.
7          else if updated(m) then
8              Update the model in the index (refer to Section 4.3.5.2).
9          else if ( renamed(m) or moved(m) ) then
10             Update the model in the index (see Section 4.3.5.2).
11         end
12         resolveModelProxies()
13     end
14 end
15 scheduleNextUpdate()
```

Similarly, creation of the database-specific persistence varies with the back-end used (described by the relevant *IGraphDatabase* driver). Examples using Neo4J and OrientDB as stores can be found in [34].

Metamodel removal Metamodels can be removed at any time (identified using their unique namespace URI) and doing so will also remove any metamodels depending on them, as well as models conforming to these metamodels (and subsequently from any metamodels depending on the removed ones) from Hawk. When this operation is used, any repositories containing models affected by this removal are flagged in Hawk so that they are not ignored should a future insertion of the same metamodel occur (whilst the repository itself has not changed).

Metamodel evolution Hawk works under the assumption that metamodels are not meant to change often and hence does not support advanced metamodel evolution procedures. If a new version of a metamodel needs to be used it can either be inserted with its own unique namespace, in parallel to the old version, in order to maintain the latest version of the model using the old version as well as the new one; or the old metamodel can be removed (and consequently any models conforming to it) and the new metamodel can then be inserted as normal.

Model file addition The procedure for parsing a model file into a resource is dependent on the file type and is described by its relevant driver (Section 4.3.4). Should the metamodel(s) of this model not exist in the store, the model insertion is aborted, pending a future insertion of its relevant metamodel(s); when the relevant metamodel is inserted the model will also be inserted in the next synchronization cycle of Hawk.

Model file update If a model file is updated to a new version, an *IHawkModelResource* is created from this file (through the relevant driver) and any changes (deltas) between it and the current version in the database are found and propagated to the store and the new version is now recorded. Details of this procedure can be found in Section 4.3.5.2.

Model file rename/move If a model file is renamed or moved in the version control system, it is considered as removed and re-added as all complex version control operations are broken down to a sequence of simple operations (add/update/remove). The reason for choosing this "simplified" approach is due to the inherent complexity of move operations on possibly inter-connected model fragments, as cross-file references will be affected (broken if all relevant fragments are not also updated after the move). For example if file "a.xmi" is a model fragment referencing an element in file "b.xmi" and file b.xmi is renamed to "c.xmi", then (as file "a.xmi" has not been altered) the reference is broken as it points to a non-existing file ("b.xmi") and hence needs to be removed from Hawk.

The above procedure ensures any changes to files will be propagated to the model index and that each file is only read once, and never more than 1 file is in memory at a time. This solution is seen as a performant and low resource consuming option for synchronization.

4.3.8. Advanced Features and Optimizations

After the creation of the core implementation for storing models in Hawk, presented in this chapter, further functionality was added to improve both the capabilities and performance of Hawk.

4.3.8.1. Derived Attributes

Regardless of the use of native or back-end independent querying, in order to respond to a model validation query requesting whether all diverging parallel gateways have more than one outgoing sequence flow and exactly one incoming sequence flow and that all converging parallel gateways have more than one incoming sequence flow and exactly one outgoing sequence flow, the following steps would have to occur:

1. The starting point of the query would have to be found. In this case, the collection of all instances of *ParallelGateway* in the model would have to be retrieved.
2. For each parallel gateway node, its *diverging* attribute would have to be retrieved as well as its number of incoming and outgoing sequence flows.
3. Depending on the value of the *diverging* attribute the numbers of sequence flows the parallel gateway has will have to be appropriately analyzed in order to determine whether the validation passes for this specific parallel gateway. If all the gateways meet the requirements *True* can be returned.

Step 1 is easy to perform in Hawk as a (database) index of *Metamodels* is kept which can be used to rapidly provide a starting point for a query which requires elements of a specific type (such as *ParallelGateway* instances for example). If a query uses the whole model index as a starting point then there is no optimization to be performed as the entire model index would have to be traversed in order to find the Node representing the *ParallelGateway* type. Step 2 where we can begin optimizing to improve the execution time of queries which have to perform some non-trivial calculation or navigation on the model.

An effective way to increase query efficiency is to pre-compute and cache – at model indexing time – information that can be used to speed up particular queries of interest to the modeler.

Such attributes are computed using expressions formed in the expression language of a known *Query Engine*. A query engine, as discussed in Section 4.3.6, allows for expression languages (such as OCL or Epsilon's EOL [16]) to be used as a query mechanism for a Hawk model index. Such derived attributes are hence pre-computed and cached at model indexing time and need to be maintained as the model index evolves.

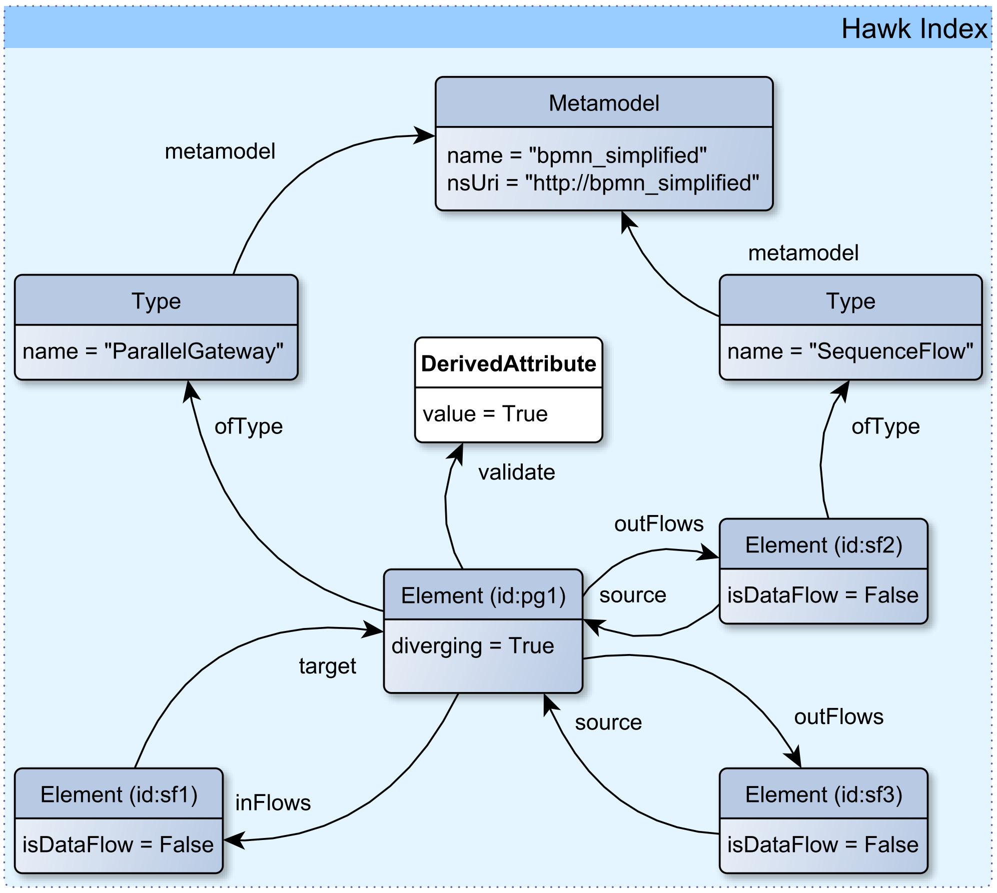


Figure 4.10.: Pre-computing whether parallel gateways are validated

A simple example is shown in Figure 4.10; here, the result of the validation of each parallel gateway is pre-computed (using the EOL program presented in Listing 4.1) and this information is stored in a new *DerivedAttribute* node[24] with the attribute name as the relationship linking it to its parent *Element* node. This derived attribute is handled seamlessly with regards to querying, hence an EOL query used to validate parallel gate-

[24] a new node is used for overcoming a limitation found during incremental updating of derived attributes; further information on this can be found in Section 4.3.8.2

ways would change from the one presented above to: *not ParallelGateway.all.exists(pg | pg.validate==False)* (in both cases returning the Boolean value of the validation).

Listing 4.1: EOL program used to calculate whether a parallel gateway is validated

```
var diverging = self.diverging;
var incomingSFs = self.inFlows.size();
var outgoingSFs = self.outFlows.size();
return ( diverging and incomingSFs == 1 and outgoingSFs > 1 )
   or ( ( not diverging ) and incomingSFs > 1 and
   outgoingSFs == 1 ) ;
```

Expressions of arbitrary complexity are expected to be used in practice so that pre-caching the results of such expressions is actually worthwhile[25]; other examples are presented in the Evaluation Section 5.3.3.

4.3.8.2. Derived Attributes: Incremental Updating

A naive approach for maintaining such attributes would involve having to fully re-compute each one, every time any change happens to the model index. This is due to the fact that any such attribute can potentially depend upon any model element in the model index, thus any change can potentially affect any derived attribute. Such an approach would be extremely inefficient and resource consuming.

As such, an incremental approach for updating derived attributes in Hawk has been used. In this approach, which is an adaptation of the incremental OCL evaluation approach discussed in [72], only attributes affected by a change made to the model index are re-computed when an update happens. In order to know which elements affect which derived attributes, the scope of a derived attribute needs to be calculated. The scope of a derived attribute comprises the current model elements (and/or features) in the model index this attribute needs to access in order to be calculated. When a derived attribute is added/updated in the model index, the query engine used to calculate this attribute also publishes an *AccessListener* to Hawk, providing the collection of *Accesses* this attribute performed. By recording these accesses (element and/or feature accesses), Hawk updates only the derived attributes which access an element altered during an

[25] in order to reduce the inherent complexity of derived attributes depending on other derived attributes, this is not allowed

incremental update. As the incremental update changes the minimal number of elements during model evolution, the updating of derived attributes can be seen to be as efficient as possible with respect to the magnitude of the change.

In more detail, every time an update process happens in Hawk, it records the changes it has made to the model index. A *change* can be one of the following:

- A model element has been created / deleted
- A feature (attribute or reference) of a model element has been altered (set/unset if single valued or updated if multi-valued)

⋆ Note: complex changes (like *move*) are broken down to these simple changes.

Furthermore, every time a derived attribute is added or updated, it records the accesses it requires in order to be computed. An *access* can be one of:

- Access to an attribute or reference value of a model element
- Access to the collection of model elements of a specific *type* or *kind*

By having recorded the above-mentioned changes and accesses, Hawk can calculate which derived attributes need to be re-computed during a model update using Algorithm 5. As the derived attribute is a node itself, it can be directly referenced and updated if necessary; if the derived attribute was located inside its parent *Element* node, that node would have to be referenced instead and hence all derived attributes in it would have to be updated, as there would not be a way to distinguish which ones need updating and which ones do not.

In the loan example, for the derived attribute *isSlow* of node *t1 (name : Record Loan App Info)*, the access would read as follows: The derived attribute *isSlow* (of node *t1*) needs to access node *t1* for its feature *executionTime*; it also needs to access nodes *t2, t3, t4 and t5* for their feature *executionTime*. Hence any time the feature *executionTime* changes for any of the above-mentioned nodes, the derived attribute *isSlow* will have to be recomputed (and only then). As demonstrated by [73], this approach works for expressions of arbitrary complexity as long as they are deterministic (they do not introduce any randomness using random number generators, hash-maps or other genuinely unordered collections). As EOL defaults to using Sequences for collections and does not inherently use random number generators, as long as the expressions provided do not specifically introduce non-determinism, this approach is sound [73].

Algorithm 5: Derived attribute incremental update algorithm

```
1  let nodesToBeUpdated be the set containing the derived attribute nodes which
     will have to be updated – initially empty
2  foreach change in the collection of changes do
3      if the change is a model element addition/deletion then
4          add any derived attribute which accesses this element (or any of its
             structural features) to nodesToBeUpdated
5      else if the change is an attribute/reference value alteration then
6          add any derived attribute which accesses this structural feature to
             nodesToBeUpdated
7      end
8  end
9  foreach node in nodesToBeUpdated do
10     re-compute the value of the (derived attribute) node
11     update the accesses to the new elements/features this node now requires
12 end
```

4.3.8.3. Database Indexing

Another effective way to increase query efficiency is to create custom database indexes of interesting attributes – at model indexing time – so that they can be used to speed up particular queries of interest. Using the bpmn loan example, we can index the "executionTime" attribute of tasks, as shown in Figure 4.11.

By effectively caching this information into a database index, a query requesting all tasks taking longer than an hour can be optimized so that it does not have to iterate though all the tasks in the model index, but instead it can directly compare the integer 3600 (as execution time is in seconds) to the keys of the index named "`http://bpmn_simplified#Task#executionTime`", which contains the values of the attribute "executionTime" of the instances of type "Task" in the metamodel with unique identifier "http://bpmn_simplified".

Any model attribute can be indexed in Hawk, a process performed during model

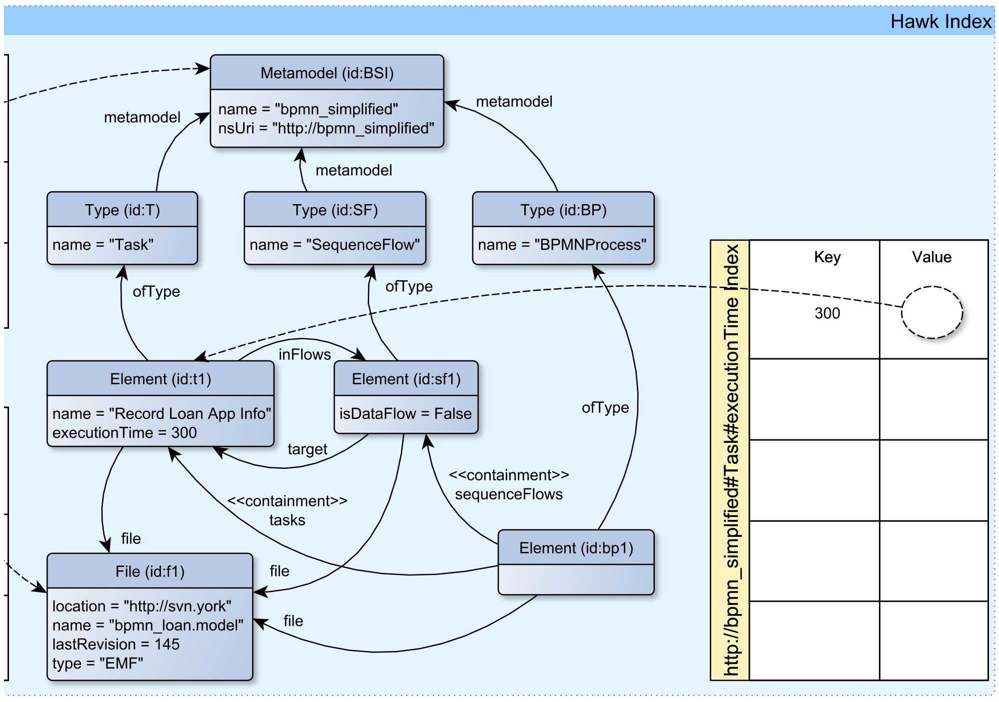


Figure 4.11.: Example of database indexing in Hawk

insertion into the Hawk model index. When Hawk updates any changes made to any models it is indexing, such attributes are updated in the database indexes automatically. Section 5.3.3 evaluates this feature.

4.3.8.4. Querying an optimized Hawk Model Index

In order to be able to use indexed attributes to speed up queries, any query engine (such as the Epsilon Query Engine we have implemented) needs to be aware of possible attribute indexes and how to handle them. The first step is to make the engine aware that in certain situations it can avoid requesting a collection of results by going to the model index and iterating through its contents, but can circumvent this procedure and use the custom (database) indexes found in the model index to retrieve the exact results directly. In Epsilon, this is done by creating a collection implementing the *IAbstractOperationContributor* interface, so that it can use these database indexes to optimize the performance of filtering operations performed on it.

This is implemented by introducing an *OptimisableCollection*, which extends the Java *HashSet* class (which in turn extends *Collection*). This custom collection keeps various meta-items of the context in which it was created, so that should a relevant operation be called on it, it can possibly use an optimized approach. More specifically, it keeps a pointer to the *IModel* which created it, one to a *Node* in the database which denotes the *type* of the elements in this collection, and a custom *SelectOperation* which it uses. A *select* operation in Epsilon takes a collection of model elements and returns a subset of this collection that satisfies a boolean condition (various other logical operations such as *exists*, *reject* etc. use this operation to be computed).

It is notable that such *OptimisableCollection*s are only created when a *.getAllOfType* or *.getAllOfKind* operation is called in an EOL program and hence its contents are all of a common (super-)*type*. Furthermore since this collection is created by a *.getAll** operation, we know that it contains a complete set of instances of the *type* and as such traversing it may be optimizable, should any of the attributes of the *type* be database indexed (a detailed discussion of how a query on a collection of elements is optimizable is detailed below).

Now, should a *select* operation be called by Epsilon upon an *OptimisableCollection*, Hawk's custom *SelectOperation* triggers (instead of the default Epsilon one) and will attempt to optimize the selection of elements if possible:

1. If the select condition contains an expression of the form "x.attr $\star$ value" (where $\star$ denotes a comparison operator[26] – thus is potentially optimizable by use of custom attribute indexes), instead of iterating through the collection and comparing the 'value' of attribute 'attr' with the value of each element, it first checks whether an index of attribute 'attr' exists for elements of that type. If such an index exists, it is used to directly get the sub-collection required without having to perform a costly traversal of all the persisted elements (hence the expression is optimizable).

2. If the select condition contains an expression containing one of the operators: *and, or, xor, implies & not*, it will be decomposed every time such an operator is found, using the following strategy:

   a) If the select condition contains an expression of the form "a **and** b" where 'a' & 'b' are arbitrary sub-expressions, it will first check whether 'a' or 'b' are optimizable (i.e. they satisfy step 1). If both are optimizable, it will

[26] ie: <element collection>.select(x | x.attr = value), <element collection>.select(x | x.attr > value), <element collection>.select(x | x.attr < value), <element collection>.select(x | x.attr >= value), <element collection>.select(x | x.attr <= value)

return a collection containing the set intersection of the results satisfying ‘a’ and those satisfying ‘b’. If only one of the two is optimizable, it will create a collection of the relevant results (of the optimizable sub-expression) and attempt to decompose the other sub-expression (using step 2), defaulting to passing the computed collection (returned by the abovementioned optimized sub-expression execution) to the default EOL driver for executing the non-optimizable sub-expression on this collection (effectively performing efficient partial filtering as it will pass a potentially much smaller collection to the Epsilon driver). If none of the two expressions are optimizable it will pass each sub-expression (‘a’ & ‘b’) to be further decomposed (using step 2) and then will take their computed results and perform a set intersection on them. For example:

- if the expression is: *Task.all.select(t | t.executionTime > 100 and t.execu tionTime < 200)* and the attribute index of ‘executionTime’ is present, both expressions will be optimized and their results will be set-intersected.
- if the expression is: *Task.all.select(t | t.executionTime > 0 and t.out Flows.size > 1)* and the index of ‘executionTime’ is present, the left hand side (t.executionTime > 0) will be optimized and the right hand side will be passed to the default Epsilon execution engine for evaluation (as it cannot be further decomposed), with the result calculated from the left hand side as the element collection provided to the engine – instead of the original, larger collection of all *Task*s). The result provided by the Epsilon execution engine will be returned.
- if the expression is: *Task.all.select(t | t.executionTime > 0 and t.execution Time < 100 and t.outFlows.size > 1)* with the index of ‘executionTime’ being present, the sub-expression (t.executionTime > 0) will be optimized and the remaining sub-expression will be decomposed (providing it the result of the optimized execution as its element collection). Then the (t.executionTime < 100) sub-expression will be optimized and the remaining sub-expression will be passed to the default Epsilon execution engine for evaluation (on the new, even smaller element collection, calculated using the previous optimized executions of both sub-expressions above), with the results of the first two (optimized) executions being

set-unioned and then the resulting collection set-unioned with the third (non-optimized) execution results.

b) If the select condition contains an expression of the form "a **or** b" where 'a' & 'b' are arbitrary sub-expressions, it will first check whether 'a' & 'b' are both optimizable. If both are optimizable, it will return a collection containing the set union of the results satisfying 'a' and those satisfying 'b'. If one or none of the two expressions are optimizable it will attempt to decompose the sub-expressions 'a' & 'b' (using step 2) and then will take their results and perform a set union on them. For example:

   - if the expression is: *Task.all.select(t | t.executionTime < 100 or t.execu tionTime > 200)* and the attribute index of 'executionTime' is present, both expressions will be optimized and their results will be set-unioned.
   - if the expression is: *Task.all.select(t | t.executionTime < 100 or t.execu tionTime > 200 and t.outFlows.size > 1)* with the index of 'execution-Time' being present, the sub-expression (t.executionTime > 0) will be optimized and the remaining sub-expression will be decomposed (providing it the result of the optimized execution as its element collection). Then the (t.executionTime < 100) sub-expression will be optimized and the remaining sub-expression will be passed to the default Epsilon execution engine for evaluation (on the new, even smaller element collection, calculated using the previous optimized executions of both sub-expressions above), with the results of the first two (optimized) executions being unioned and then the resulting collection set-intersected with the third (non-optimized) execution results.

c) If the select condition contains an expression of the form "a **xor** b" where 'a' & 'b' are arbitrary sub-expressions, it will first check whether 'a' & 'b' are both optimizable. If both are optimizable, it will return a collection containing members of the collection satisfying 'a' or members satisfying 'b' but not ones satisfying both. If one or none of the two expressions are optimizable it attempt to decompose the sub-expressions 'a' & 'b' (using step 2) and then will take their results and perform the abovementioned exclusive or comparison on them.

d) If the select condition contains an expression of the form "a **implies** b" where 'a' & 'b' are arbitrary sub-expressions, it will first check whether 'a' & 'b' are both optimizable. If both are optimizable, it will return the original collection with members of it which satisfy 'a' removed (from the original source collection) and members satisfying 'b' re-added (if not already present – effectively performing "not(a) or b"). If 'a' is optimizable it will firstly get the collection satisfying 'a'. It will then use this sub-collection as the source collection the filter of 'b' runs on. It will then create a collection containing elements of the sub-collection of 'a' minus those of the sub collection of 'b' and return the original collection minus this one (effectively performing "not(a and not(b))"); similarly if 'b' is optimizable. If none of the two expressions are optimizable it will attempt to decompose the sub-expressions 'a' & 'b' (using step 2) and then will take their results and perform the abovementioned implication comparison on them.

e) If the select condition contains an expression of the form "**not**(a)" where 'a' is an arbitrary sub-expression, it will first check whether 'a' is optimizable. If it is, it will return the result of the original source collection minus the elements satisfying 'a'. If it is not, it will attempt to decompose the sub-expression 'a' (using step 2) and then will take the result and perform the abovementioned negation operation on it.

3. Should the select condition contain any other expression (or sub-expression that cannot be further decomposed using step 2) it delegates to the default Epsilon *SelectOperation execute()* method, to perform the job (providing it the relevant source collection, possibly shrunk by performing partial optimized executions as detailed above).

4.3.9. Summary

This section detailed the design of the model indexing system (Hawk). It presented the model index structure and explained the importance of each of its key elements (such as *File nodes, Element nodes*, etc.). It then discussed the two mapping layers the system offers and their importance in offering an extensible system capable of incorporating new back-ends and model persistence technologies on demand, in the form of pluggable drivers. Information about the various components of the system was then provided: for the version control and modeling components the focus was on their extensibility, whist for the querying and updating components the focus was on performance. Finally, this section demonstrated how the use of derived attributes and database indexing can offer potential performance improvements for querying whilst attempting to minimize their overhead on the system. The following Section 4.4 offers a concise summary of the implementation details of the Hawk model indexing prototype developed as part of this work, whilst Chapter 5 evaluates the various features and optimizations presented here.

4.4. Implementation

This section details the implementation details of the current Hawk prototype; a user guide can be found in Appendix C.

4.4.1. Eclipse Plugins

Even though Hawk can run as a standalone Java application if need be, it offers Eclipse IDE integration for providing an administrator user interface and automating relevant lifecycle operations.

The structure of Hawk's primary plugins is as follows:

- *org.hawk.core plugin* This plugin (from now on referred to as the "core" plugin) contains all the interfaces of Hawk as well as the necessary implementation classes for running the main Hawk controller *IModelIndexer*. It only depends on the *org.hawk.core.dependencies* plugin. It defines all the extension points to used by other plugins in order to be automatically discovered upon initializing Hawk:

  org.hawk.core.ModelExtensionPoint extension point Allows extensions defining new types of model resource factories to be added to Hawk.

  org.hawk.core.MetaModelExtensionPoint extension point Allows extensions defining new types of metamodel resource factories to be added to Hawk.

  org.hawk.core.QueryExtensionPoint extension point Allows extensions defining new types of query engines to be added to Hawk.

  org.hawk.core.ModelUpdaterExtensionPoint extension point Allows extensions defining new types of model updater to be added to Hawk.

  org.hawk.core.MetaModelUpdaterExtensionPoint extension point Allows extensions defining new types of metamodel updater to be added to Hawk.

org.hawk.core.BackEndExtensionPoint extension point Allows extensions defining new types of back-end stores (*IGraphDatabases*) to be used by Hawk.

org.hawk.core.VCSExtensionPoint extension point Allows extensions defining new version control managers be added to Hawk.

- *org.hawk.core.dependencies plugin* This plugin contains all the (publicly available – license compliant) dependencies of the core plugin. Currently the only dependency is *xstream-1.4.8.jar*, used to serialize and deserialize Hawk metadata used for persisting its model indexes between runs.

- *org.hawk.emf plugin* This plugin contains the necessary implementations for reading EMF XMI-based metamodels and models into memory. It depends on the core plugin as well as the *org.eclipse.emf.ecore* and *org.eclipse.emf.ecore.xmi* plugins. These plugins can be obtained either automatically in any Eclipse modeling tools distribution or manually from the EMF Eclipse update site. It implements extensions for Hawk's *ModelExtensionPoint* and *MetaModelExtensionPoint* extension points.

- *org.hawk.bpmn plugin* This plugin contains the necessary implementations for reading BPMN EMF XMI-based metamodels and models into memory. It depends on the core plugin as well as the *org.eclipse.emf.ecore*, *org.eclipse.emf.ecore.xmi* and *org.eclipse.bpmn2* plugins. These plugins (except the BPMN one) can be obtained either automatically in any Eclipse modeling tools distribution or manually from the EMF Eclipse update site. The BPMN plugin can be obtained from the relevant Eclipse update site[27]. It implements extensions for Hawk's *ModelExtensionPoint* and *MetaModelExtensionPoint* extension points.

- *org.hawk.graph plugin* This plugin contains the necessary implementations for updating metamodels or models in Hawk. It depends on the core plugin. It implements extensions for Hawk's *MetaModelUpdaterExtensionPoint* and *ModelUpdaterExtensionPoint* extension points.

- *org.hawk.svn plugin* This plugin contains the necessary implementations for connecting to an SVN version control system in Hawk. It depends on the core plugin as well as *org.tmatesoft.svnkit* and *org.tmatesoft.sqljet* plugins. These plugins can

[27] `http://download.eclipse.org/bpmn2-modeler/updates`

be found in the Eclipse update site of the official svnkit website. It implements an extension for Hawk's *VCSExtensionPoint* extension point.

- *org.hawk.localfolder plugin* This plugin contains the necessary implementations for using a local folder as a pseudo-version control system in Hawk. It depends on the core plugin. It implements an extension for Hawk's *VCSExtensionPoint* extension point.

- *org.hawk.git plugin* This plugin contains the necessary implementations for using a local git repository as a version control system in Hawk. It depends on the core plugin. It implements an extension for Hawk's *VCSExtensionPoint* extension point.

- *org.hawk.neo4j-v2 plugin* This plugin contains the necessary implementations for connecting with a Neo4J NoSQL database. It depends on the core plugin as well as the *org.hawk.neo4j-v2.dependencies* plugin. It implements an extension for Hawk's *BackEndExtensionPoint* extension point.

- *org.hawk.neo4j-v2.dependencies plugin* This plugin contains the necessary Neo4J files for running a Neo4J database. As Neo4J has a non-compatible license with the Eclipse public license of Hawk, the relevant *.jar*s are not included in the plugin, only references to which ones are needed. A public open-source release of these Neo4J files can be found in the official Neo4J website under community downloads.

- *org.hawk.epsilon plugin* This plugin contains the necessary implementations for using the Epsilon model suite to perform queries on Hawk. It depends on the core plugin as well as the *org.eclipse.epsilon.eol.engine* plugin. This plugin can be obtained from the public distribution of the Epsilon toolset in Eclipse. It implements an extension for Hawk's *QueryExtensionPoint* extension point.

- *org.hawk.ui.emc.dt2 plugin* This plugin contains the necessary implementations for adding Hawk to the repositories available in the Epsilon platform, in order to integrate with the current workflow of Epsilon. It depends on the core plugin, the epsilon plugin (of Hawk), the UI plugin (of Hawk) as well as on *org.eclipse.epsilon.common.dt* (can be found in the same place as the other Epsilon plugins needed for the Hawk Epsilon plugin).

- *org.hawk.osgiserver plugin* This plugin contains the necessary implementations for running Hawk server in a headless OSGI environment. It depends on the core plugin, as well as various standard Eclipse plugins that expose OSGI functionality.
- *org.hawk.ui2 plugin* This plugin (UI plugin) contains the necessary implementations for offering a user interface for Hawk integrated into Eclipse (by extending the osgiserver plugin). It depends on the core plugin, the osgiserver plugin, as well as various standard Eclipse UI plugins.

Furthermore, various other plugins were developed in the scope of the MONDO project (in collaboration with Dr. Seyyed Shah and Dr. Antonio Garcia Dominguez)

- *org.hawk.ifc plugin* This plugin contains the necessary implementations for reading Building Information Modeling (BIM) Industry Foundation Classes (IFC)-based metamodels and models into memory. It depends on the core plugin as well as the *org.eclipse.emf.ecore* and *org.eclipse.emf.ecore.xmi* plugins. It implements extensions for Hawk's *ModelExtensionPoint* and *MetaModelExtensionPoint* extension points.
- *org.hawk.modelio plugin* This plugin contains the necessary implementations for reading Modelio-based metamodels and models into memory. It depends on the core plugin, the *org.eclipse.emf.ecore* and *org.eclipse.emf.ecore.xmi* plugins, as well as various Modelio plugins. The Modelio plugins can be found in the public open-source distribution of the Modelio tool found online. It implements extensions for Hawk's *ModelExtensionPoint* and *MetaModelExtensionPoint* extension points.
- *org.hawk.emfresource plugin* This plugin contains the necessary implementations for exposing a Hawk model index as a standard EMF resource. It depends on the core and graph plugins as well as the *org.eclipse.emf.ecore* plugin.
- *org.hawk.orientdb plugin* This plugin contains the necessary implementations for connecting with an OrientDB NoSQL database. It depends on the core plugin. It implements an extension for Hawk's *BackEndExtensionPoint* extension point.
- *org.hawk.workspace plugin* This plugin contains the necessary implementations for connecting to an active Eclipse Workspace as a version control system in Hawk. It depends on the core plugin as well as various standard Eclipse plugins. It implements an extension for Hawk's *VCSExtensionPoint* extension point.

- *org.hawk.ui.emfresource plugin* This plugin contains the necessary UI implementations for exposing a Hawk EMF resource and for using an optimized version of the Epsilon Exeed tree editor with Hawk. IT depends on the core, graph, osgiserver, UI and emfresource plugins as well as the *org.eclipse.emf.ecore* and the *org.eclipse.epsilon.dt.exeed* plugins.

All relevant API operations are exposed through a GUI in the form of an Eclipse view, a detailed presentation of which can be found in Appendix C.

5. Evaluation

This chapter evaluates the research hypothesis by presenting the evaluation strategy, evaluation benchmarks and evaluation results obtained during this work. The evaluation strategy Section 5.1 presents which aspects of Hawk can be evaluated, detailing the use-cases covered in each case. It discusses how each scenario offers unique and critical insight into the relevant functional and non-functional aspects of Hawk that are to be evaluated. The evaluation benchmarks Section 5.2 outlines the various benchmarks used in the evaluation. It presents each one and summarizes its relevance with respect to validating the research hypothesis. The evaluation results Section 5.3 uses the various benchmarks presented to test the various functional and non functional requirements of Hawk, with the tool integration Section 5.4 performing a qualitative evaluation of Hawk's architecture. Finally Section 5.6 summarizing the findings and how they validate the research hypothesis.

5.1. Evaluation Strategy

Evaluating the various aspects of a component-based system like Hawk firstly requires splitting them into their distinct categories. This section presents them and discusses the relevance of each one. Two primary categories, that of correctness and that of performance, are identified, each broken down into relevant scenarios clarifying the specific aspect of Hawk being looked at.

5.1.1. Correctness

Evaluating the correctness of the algorithms used in Hawk can be broken down into two aspects: the correctness of the contents of the model index (reflecting the correctness of the various algorithms used to insert and update models into Hawk) and the correctness of the results returned by querying this index.

5.1.1.1. Index Content Correctness

As Hawk is a model index that stores copies of models, an important aspect to evaluate is the correctness of this index with respect to the model files it is monitoring. More specifically there are three scenarios to consider:

- A new model file is found and inserted into Hawk. In this scenario we need to ensure that Hawk now contains the contents of the new file, that the previous contents of Hawk are unaltered, that any proxy references pointing to the new file are resolved and that any derived attributes that need updating are updated correctly.
- An existing model file is updated and the changes need to be propagated into Hawk. In this scenario we need to ensure that the updated version of the model is propagated to Hawk so that the index now matches the new version, that the remaining (unaffected) contents of the index are unaltered, that any proxy references pointing to the changed file are resolved and that any derived attributes that need updating are updated correctly.
- A model file is deleted and needs to be removed from Hawk. In this scenario we need to ensure that any elements of the deleted file are removed from hawk, that the remaining contents of Hawk are unaltered, that any references to the deleted file from other files are kept as proxies and that any derived attributes that need updating are updated correctly.

By handling these three cases we are able to ensure consistency in the model index under normal execution as any compound change such as files being moved can be broken down to a sequence of the aforementioned changes. By a consistent state of the model index we mean the contents of the index for that version of the model reflect the contents of the original model, so that regardless of whether a user retrieves information from Hawk or the original model, the same results are returned.

To ensure consistency under failure we need to ensure that changes are transactional and that any failure is rolled back to a previous stable state, so that hawk can retry the change the next time it is able to.

In order to hook into these changes we use the change notification framework provided by Hawk (Section 4.3.2.2) and run the Hawk validation listener (detailed below) after Hawk attempts to synchronize with its monitored repositories.

Readers can refer to sections 5.3.4.2 and 5.4.4.2 for empirical data collected on tool correctness.

5.1.1.2. Hawk Validation Listener

Using Hawk's update notification framework (presented in Section 4.3.2.2) a validation tool has been created that allows for checking Hawk's consistency; this tool also outputs various interesting metrics regarding each of Hawk's synchronizations performed (one such analysis is performed for each repository Hawk is currently watching over). The tool has two modes, one which only records metrics for each update process and one which also does a full two-way analysis of the Hawk index and the changed resources in order to validate Hawk's consistency. The first mode has little to no impact on performance but the second mode has a large overhead as it not only performs a costly comparison (presented below) but it also requires that Hawk keeps open ALL resources used during a synchronize so that it can be given them to use for its validation (normally Hawk can dispose of any resource used and only have one open at a time, in order to use less memory).

Metrics provided are the following:

- total number of model files present in the current commit.
- number of changed model files Hawk needs to update.
- number of deleted model files Hawk needs to remove.
- number of changed model files successfully loaded into resources using their relevant model factory.
- number of model elements changed during this update (added or updated into Hawk).
- number of model elements deleted during this update (either due to the model file being deleted or the element being deleted in the latest version of the model file).
- time taken for the synchronize to complete (not counting the time taken for this tool to perform its analysis/validation).

Furthermore, should validation mode be enabled, this tool also performs a comprehensive two-way comparison between the current version of Hawk and any model resources used in this synchronize. More specifically, this process performs Algorithm 6, which checks that for each model resource Hawk contains exactly the elements found in the resource (and no other elements) and that for each such element its attributes and references match the ones of the resource itself. If any inconsistency is found (either in the model itself or in Hawk's indexing of the model) it is output by the tool for informative and debug purposes.

5.1.1.3. Query Correctness

In order to ensure correct results to queries performed on Hawk (Section 2.2.3), it is not sufficient to have confidence in the contents of the model index alone, there is the need to ensure that the query engines used to query Hawk are also correct. The index is said to offer correct querying results when for any query, regardless on whether it is executed on the model index or the original model itself, the same results are returned. In practice, as testing all possible queries is not possible, empirical data can be gathered using a collection of queries whose results are known beforehand, and comparing the results obtained by querying hawk to these results. Refer to sections 5.3.1.2 and 5.4.4.2 for more details.

5.1.2. Performance

As one of Hawk's non-functional requirements is performance when dealing with large-scale models, evaluating this aspect is essential. This evaluation can be broken down into two main areas: query performance and update performance.

5.1.2.1. Query Performance

Hawk aims at enabling performant global queries on models indexed by it. As such, regardless of how large or how many models are stored in Hawk it should be able to seamlessly handle queries provided to it in an efficient manner. As described in Section 4.3.1, as the contents of Hawk's model indexes are stored as homogeneous property graphs performant global querying can be achieved by retrieving the necessary information from the required elements in the graph. Empirical data gathered on querying large models can be found in Section 5.3.1.2 and improvements observed when using derived and indexed attributes can be found in Section 5.3.3.2.

Algorithm 6: Validation algorithm

```
1  let totalGraphSize be the number of model elements in Hawk that are contained
     in one of the model files changed in this commit
2  let totalResourceSize be the number of model elements in all of the loaded
     resources of model files changed in this commit
3  foreach changed model file in this commit do
4      retrieve the model resource created by Hawk for this model file
5      add the size of the resource to totalResourceSize
6      foreach graph node in Hawk linked to the current model file do
7          add one to totalGraphSize
8          if the node cannot be mapped to a model element in the resource then
9              inform that validation has failed alongside information about this
                 node which is not in the model resource
10         else
11             compare the attribute values of the node with the values of the model
                 element in the resource, informing of any inconsistencies and failing
                 the validation if there are any
12             compare the reference values of the node with the values of the
                 element, informing of any inconsistencies and failing the validation if
                 there are any
13         end
14     end
15     if there are any left over model elements not mapped to a node then
16         inform that validation has failed alongside information about the model
             elements not found in Hawk
17     end
18 end
19 if totalGraphSize not equal to totalResourceSize then
20     inform that validation has failed as Hawk and the loaded resources do not
         contain an identical number of model elements
21 end
```

5.1.2.2. Update Performance

Hawk needs to have the ability to handle evolving models stored in version control systems (Section 4.1). As such it needs the ability to efficiently update its contents every time any monitored model is changed. Furthermore it needs to efficiently re-compute its derived attributes so that their values reflect the new state of the index.

Updating few large models. The first benchmark performed (Section 5.3.1) covers updating a small set of large models (in the order of millions of model elements) and attempted to push Hawk to its limits with regards to the number of changes in each update.

Updating many small models. The next benchmark (Section 5.3.4) covers updating a large set of smaller models and aimed to demonstrate Hawks efficiency in handling large amounts of small changes to an index comprised hundreds of models.

5.1.3. Tool Integration

One of the envisioned benefits of Hawk is that it allows for easy integration with current tools as it provides an orthogonal technology to current model storage and through its simple generic API. As part of the integration efforts in the MONDO project (in collaboration with Dr. Antonio Garcia Dominguez and Gábor Szárnyas), several tools have been integrated with Hawk. Details can be found in section 5.4.

5.1.4. Architecture Evaluation

As Hawk aims to provide an extensible heterogeneous model indexing framework, evaluating the extent to which its architecture can be used with various modeling and data persistence technologies is needed. Details on this can be found in Section 5.5.

5.2. Evaluation Benchmarks

This section presents the sources for the benchmarks used to evaluate Hawk. In each case the reason for selection is discussed alongside details about the various artefacts used. The first case-study focuses on monolithic models of increasing sizes (reaching the order of millions of model elements) and offers a complex query that can be used to evaluate the performance of querying the model index. The second focuses on a large collection of (hundreds) of smaller models, which have been evolving, providing hundreds of different versions of this collection; this benchmark lends itself to evaluating Hawk's update procedures in terms of correctness and performance.

5.2.1. Grabats 2009 Case-Study

To obtain meaningful evaluation results when using large models, large-scale models extracted by reverse engineering existing Java code are used. In particular, the updated version of the JDTAST metamodel used in the *SharenGo Java Legacy Reverse-Engineering* MoDisco [74] use case, presented in the Grabats 2009 contest [75] described below, as well as the five models also provided in the contest.

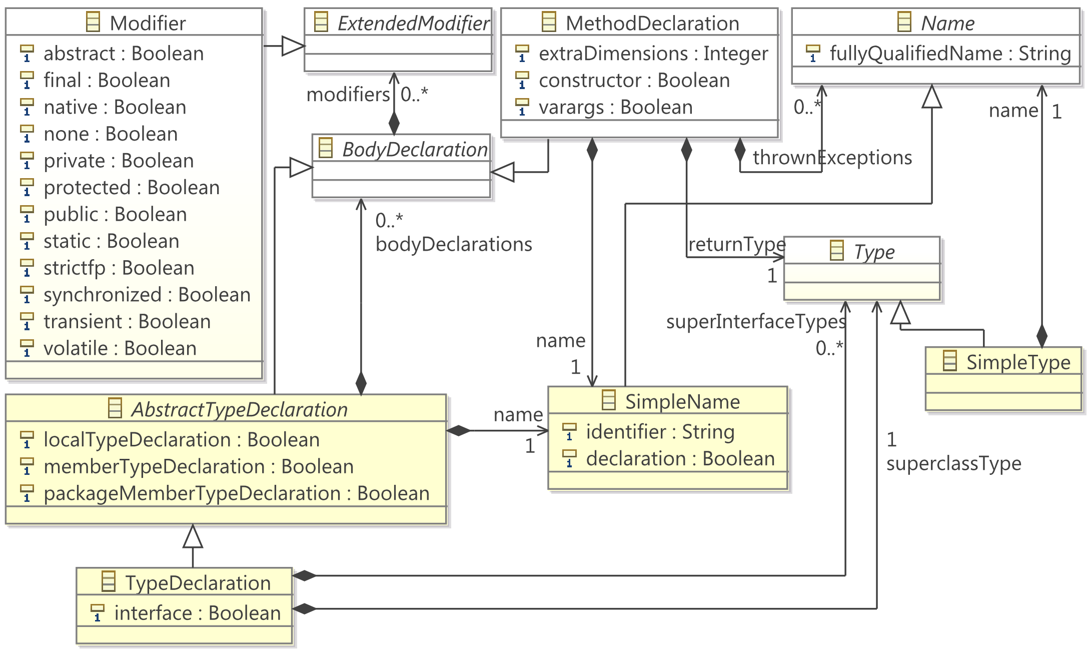


Figure 5.1.: Small subset of the Java JDTAST metamodel

A subset of the Java JDTAST metamodel is presented in Figure 5.1. In this figure, there are *TypeDeclarations* that are used to represent Java classes and interfaces, *MethodDeclarations* that are used to define Java methods (in classes or interfaces, for example) and *Modifiers* that are used to represent Java modifiers (like static or synchronized) for Java classes or Java methods.

The Grabats 2009 contest comprised several tasks, including the case study used in this work, for benchmarking different model querying and pattern detection technologies. More specifically, task 1 of this case study is performed, using all of the case studies' models, set0 – set4 (which represent progressively larger models, from one with 70,447 model elements (set0) to one with 4,961,779 model elements (set4)), all of which conform to the JDTAST metamodel.

These models are injected into the persistence technologies used in the benchmark (insertion benchmark) and then queried using the Grabats 2009 task 1 query (query benchmark) [76]. This query requests all instances of *TypeDeclaration* elements which declare at least one *MethodDeclaration* that has static and public modifiers and the declared type being its returning type (i.e. singleton candidates).

5.2.1.1. Grabats Query

A model index like Hawk can be queried in a variety of ways, ranging from native Java, technology-specific query languages such as Cypher (for a Neo4J backend), or general-purpose languages such as Epsilon's EOL.

Below we see the Grabats query written in these three forms:

Listing 5.1: Code excerpt for the Grabats query implemented in Java for Neo4J

```
{
...
for (Relationship outEdge : typeDeclaration.getRelationships(
    Direction.OUTGOING, DynamicRelationshipType.withName("
    bodyDeclarations"))) {
  Node methodDeclaration = outEdge.getEndNode();
  if (new MetamodelUtils().isOfType(methodDeclaration,
      new MetamodelUtils().eClassNSURI(methodDeclarationClass))) {
    boolean isPublic;
    boolean isStatic;
    String currMethodName;
    for (Relationship methodDeclarationOutEdge :
        methodDeclaration.getRelationships(Direction.OUTGOING,
        DynamicRelationshipType.withName("name"))) {
      Node name = methodDeclarationOutEdge.getEndNode();
```

```
        currMethodName = name.getProperty("fullyQualifiedName").
            toString(); }
    for (Relationship methodDeclarationOutEdge :
        methodDeclaration.getRelationships(Direction.OUTGOING,
        DynamicRelationshipType.withName("modifiers"))) {
...
}
```

Listing 5.2: Code excerpt for the Grabats query implemented in Cypher for Neo4J

```
ExecutionEngine engine = new ExecutionEngine(graph);
ExecutionResult result = engine.execute("
  START dom=node:METAMODELINDEX('id:org.amma.dsl.jdt.dom')
  MATCH dom<-[]-(td{id:'TypeDeclaration'})<-[:typeOf]-(node)
  MATCH node-[:bodyDeclarations]->(methodnode)-[:modifiers]->(
      modifiernode{public:true})
  MATCH methodnode-[:modifiers]->({static:true})
  MATCH node-[:name]->(nodename)
  MATCH methodnode-[:returnType]->()-[:name]->(returntypename)
  WHERE nodename.fullyQualifiedName=returntypename.
      fullyQualifiedName
  MATCH methodnode-[:name]->(methodnodename)
  RETURN DISTINCT nodename.fullyQualifiedName,methodnodename.
      fullyQualifiedName
");
```

Listing 5.3: The Grabats 2009 query expressed in EOL

```
TypeDeclaration.all.select(
  td|td.bodyDeclarations.exists(
    md:MethodDeclaration|
      md.modifiers.exists(mod:Modifier|mod.public==true) and
      md.modifiers.exists(mod:Modifier|mod.static==true) and
      md.returnType.isTypeOf(SimpleType) and
      md.returnType.name.fullyQualifiedName == td.name.
          fullyQualifiedName
  )
);
```

In Section 5.3.1, we use the Hawk Epsilon driver to evaluate the impact of using EOL as a higher-level query language in terms of performance and in order to evaluate Hawk's effectiveness in handling this class of queries on large models.

5.2.2. The BPMN MIWG Test Suite Repository

The BPMN Model Interchange Working Group (MIWG) at the OMG has created a repository holding data about BPMN-based tools[1]. They have provided a set of eleven reference BPMN models, both in XML and graphical form, and invite any tool using BPMN models to use them in order to test its compatibility.

They record the results each tool obtains for up to four test cases (as many as the tool in question supports):

- Import. The tool uses the XML representation of the reference models in order to import them. The graphical representation of the model that is generated by the tool is then compared to the reference image to determine whether the tool provides a similar enough figure.
- Export. The tool is used to draw graphical representations of the reference models. The tool then saves these representations and exports them as XMLs. The resulting XMLs are then compared to the reference model XMLs.
- Roundtrip. The tool is used to import the reference XML models. The tool then saves the graphical representations generated and proceeds to export the XML representations of the models it initially imported. These XMLs are then compared to the original reference model XMLs.
- Cross-test. The abovementioned roundtrip technique is used with the XML representations of models exported by other tools contributing to this test suite instead of using the original reference models. The resulting XMLs are then compared to the reference model XMLs.

[1] `https://github.com/bpmn-miwg/bpmn-miwg-test-suite`

Currently (October 2015), 28 tools contribute data to this repository, resulting in hundreds of variants of the original BPMN models being stored in it, alongside hundreds of revisions representing the evolution of this test suite over time.

The combination of a large set of versions with a large collection of models lends itself nicely for evaluating Hawk's ability to rapidly synchronize with many changed models. Section 5.3.4 discusses the results obtained when indexing the various versions of the models with Hawk.

5.3. Evaluation Results

This section presents an analysis of the empirical data obtained running the evaluation benchmarks described above. Hawk's model insertion, model updating and query execution are investigated, focusing on its performance when compared to relevant state-of-the-art tools. As this work spans a substantial time-frame, the execution environment used may vary from case to case, and so is explicitly stated.

5.3.1. Benchmarking of Model Insertion and Querying Using Native Java and EOL

In this section, XMI, Teneo/Hibernate using a MySQL server, CDO (using its default H2 SQL database as well as with a MySQL server) and two prototype Hawk drivers (using Neo4J and OrientDB) implemented in this work are compared to assess their performance and efficiency in terms of memory use. It is worth noting that an attempt for running Morsa (Section 2.2.1.3) in the same environment did not succeed due to lack of documentation. This work on benchmarking model insertion and querying has been published in [34].

Execution Environment Performance figures that have been measured on a PC with Intel(R) Core(TM) i5-2300 CPU @ 2.80GHz, with 8GB of physical memory, and running the Windows 7 (64 bits) operating system are presented. The Java Virtual Machine (JVM) version 1.6.0_25-b06 has been restarted for each measure as well as for each of the 20 repetitions of each measure.

Table 5.1 shows the configurations that have been used for the JVM and for the relevant databases aiming to optimize execution time and were obtained empirically.

Table 5.1.: Configuration options for benchmarks

Config	Persistence Mechanism					
	XMI	Teneo/Hibernate	CDO[H2]	CDO[MySQL]	Hawk[Neo4J]	Hawk[OrientDB]
JVM	-Xmx6G	-Xmx6G	-Xmx5G	-Xmx5G	-Xmx6G	-Xmx5G
Database	n/a	default	default	default	2.2G MMIO	1.5G MMIO

MMIO stands for memory-mapped file IO, provided by most operating systems such as Windows[2] or Linux[3], a feature leveraged by many NoSQL databases, that allows them to rapidly access file-based storage mirrored in RAM in order to gain performance .

5.3.1.1. Model Insertion

Tables 5.2 and 5.3 show the results for the insertion of an XMI model into the databases. We assume availability of XMI model files so models written to an XMI file are omitted.

Regarding insertion time, Teneo/Hibernate did not successfully insert set2 – set4 and CDO did not successfully insert set3 – set4 (neither with H2 nor with MySQL), as even with maximum memory allocated to both client and server in both cases, they threw an exception, so values are omitted. For small model sizes, in the order of tens of megabytes (set0, set1), CDO performs the best but for larger ones, in the order of hundreds of megabytes (set2 – set4), Hawk[Neo4J] and Hawk[OrientDB] are not only able to store them successfully, but for set2 do so faster than CDO.

5.3.1.2. Query Execution Time and Memory Footprint

Table 5.4 shows the results for performing the Grabats query (Section 5.2.1.1) on the databases. As previously mentioned, the Grabats query finds all occurrences of *Type-Declaration* elements that declare at least one public static method with the declared type as its returning type.

[2] `https://msdn.microsoft.com/en-us/library/dd997372(v=vs.110).aspx`
[3] `http://pubs.opengroup.org/onlinepubs/9699919799/functions/mmap.html`

Table 5.2.: Model insertion (persistent to database) size results

Model	**Size (in MB)**					
	XMI	Teneo/Hibernate	CDO[H2]	CDO[MySQL]	Hawk[Neo4J]	Hawk[OrientDB]
Set0	8.75	38.6	26.0	34.8	29.4	53.6
Set1	26.59	83.1	67.0	75.7	85.9	134.0
Set2	270.12	-	539	551	794	1197
Set3	597.67	-	-	-	1750	2591
Set4	645.53	-	-	-	1890	2789

Table 5.3.: Model insertion (persistent to database) execution time results

Model	**Time taken (in seconds)**					
	XMI	Teneo/Hibernate	CDO[H2]	CDO[MySQL]	Hawk[Neo4J]	Hawk[OrientDB]
Set0	n/a	58.7	11.8	26.2	12.4	19.6
Set1	n/a	218.2	19.2	66.7	32.5	57.1
Set2	n/a	-	778.5	647.5	499.1	590.8
Set3	n/a	-	-	-	2210	2245
Set4	n/a	-	-	-	2432	2397

Table 5.4.: Grabats Query results **(in seconds and MB)**

Model	**Metric**	**Persistence Mechanism**							
		XMI	Teneo/Hib	CDO[H2]	CDO[MySQL]	Hawk[Neo4J]	Hawk[OrientDB]	Hawk[Neo4J](EMC)	Hawk[OrientDB](EMC)
Set0	Time	1.20	4.53	0.60	0.61	0.11	0.43	0.35	0.99
	Mem (Max)	42	248	14	12	15	10	12	15
	Mem (Avg)	19	117	12	9	11	10	11	12
Set1	Time	2.28	7.34	1.12	1.06	0.62	1.18	0.61	2.21
	Mem (Max)	111	323	17	20	18	27	12	23
	Mem (Avg)	48	176	13	17	13	17	11	14
Set2	Time	16.51	-	12.94	12.20	3.10	9.83	6.02	15.63
	Mem (Max)	813	-	98	120	401	742	520	910
	Mem (Avg)	432	-	32	70	195	255	280	390
Set3	Time	84.91	-	-	-	6.71	24.41	12.71	38.92
	Mem (Max)	1750	-	-	-	960	2229	1320	2400
	Mem (Avg)	844	-	-	-	620	881	520	750
Set4	Time	145.67	-	-	-	7.16	29.65	14.99	41.37
	Mem (Max)	1850	-	-	-	1070	2463	1410	2540
	Mem (Avg)	939	-	-	-	866	1314	810	870

As Teneo/Hibernate did not insert set2 – set4 and CDO did not insert set3 – set4 (neither with H2 nor MySQL), query values are omitted for these test cases. As can be observed, Hawk$^{[\text{Neo4J}]}$ demonstrates the best performance in terms of execution time and Hawk$^{[\text{OrientDB}]}$ is faster than XMI but also uses a comparable memory footprint. CDO has the lowest memory consumption for the queries it can run (we are not considering memory use of set0 and set1 as it is extremely low and the variance caused by the computer itself is significant) but is also slower to execute than Hawk$^{[\text{Neo4J}]}$ and comparable to Hawk$^{[\text{OrientDB}]}$.

Using this empirical data we can deduce that even though Hawk$^{[\text{OrientDB}]}$ is competitive and can be an improvement to XMI even for the largest model sizes in this benchmark, due to the fact that it is built atop a document store causes its performance to be lower than that of Hawk$^{[\text{Neo4J}]}$, which uses a pure graph-based database.

The last two columns of Table 5.4 show the results for performing the Grabats query using the Epsilon EMC layer to query the databases. The EMC layer allows Hawk to connect with the Epsilon platform in order to be queried using EOL as an expression language (Section 4.3.6.2). As can be expected this layer adds an overhead both in memory and execution time, but the results are still greatly superior to XMI persistence.

Figure 5.3 compares the total time taken for Ecore's XMI loader and our prototypes to answer the query, starting from a model provided in an XMI file. The querying time (at 0 times performed) is the time it takes to insert the model to the store as we assume the availability only of the XMI files.

The total time is calculated assuming that the persistence mechanism is disconnected from the query API each time but the persistence (of Hawk model indexes) is not deleted, and can be used to visualize after how many such runs a Hawk-based solution would be beneficial to deploy. It is worth noting that the query execution time for XMI, not counting the loading of the resource is comparable to Hawk$^{[\text{Neo4J}]}$ query execution times (seen in Table 5.4), so if a model only needs to be analyzed very few distinct times, with multiple queries executed, XMI is still the fastest approach, assuming that the client can handle the immense memory consumption it requires.

Regarding native querying, for set2, Hawk$^{[\text{Neo4J}]}$ is preferable to XMI after around 35 repeats while Hawk$^{[\text{OrientDB}]}$ after around 90. For set3, Hawk$^{[\text{Neo4J}]}$ is preferable after around 28 while Hawk$^{[\text{OrientDB}]}$ after 37 repeats. For set4, Hawk$^{[\text{Neo4J}]}$ is preferable after around 18 while Hawk$^{[\text{OrientDB}]}$ after 21 repeats.

We can observe that for any model size both Hawk solutions are beneficial after some threshold, the larger the model size the earlier we can use Hawk solution to persist it and that Hawk[Neo4J] is always more performant than Hawk[OrientDB].

Regarding EMC querying we observe that for set2, Hawk[OrientDB] has similar gradient to XMI, with the lines only intersecting at around 2500 repeats and with Hawk[Neo4J] still outperforming XMI at around 49 repeats. For set3 and set4, we see similar results to native querying, with Hawk[Neo4J] surpassing XMI at 30 and 19 repeats respectively and Hawk[OrientDB] at 53 and 23 respectively.

These results seem to show that the overhead of using Epsilon with Hawk[OrientDB] is sufficient enough to cause it to only be negligibly more efficient than XMI for relatively small model sizes (set0 – set2); when working with large enough model sizes (set3 – set4) though, Hawk[OrientDB]'s EMC performance starts to reflect that of its native querying with respect to XMI. Regarding Hawk[Neo4J], Epsilon's overhead only minorly affects its overall performance causing to be quickly surpass that of XMI for any model size.

5.3.1.3. Disc Space

As expected, Hawk[Neo4J] and Hawk[OrientDB] require more disk space than XMI. Figure 5.2 shows the ratios of relative disk space needed to store the different models (set0 – set4) for the different technologies, using the results in Table 5.2.

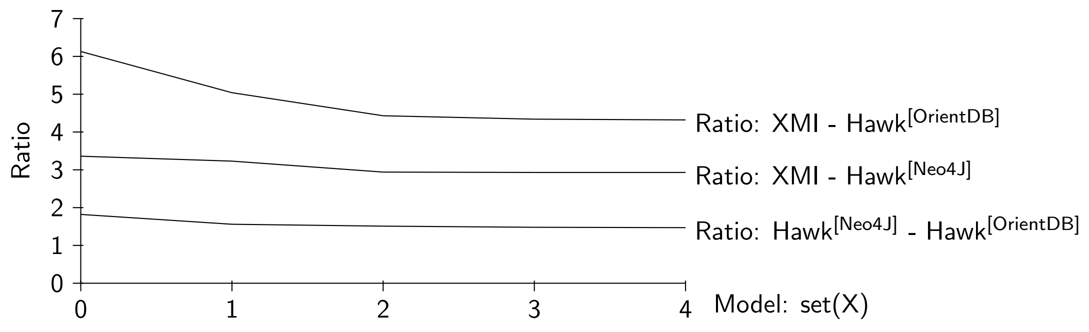
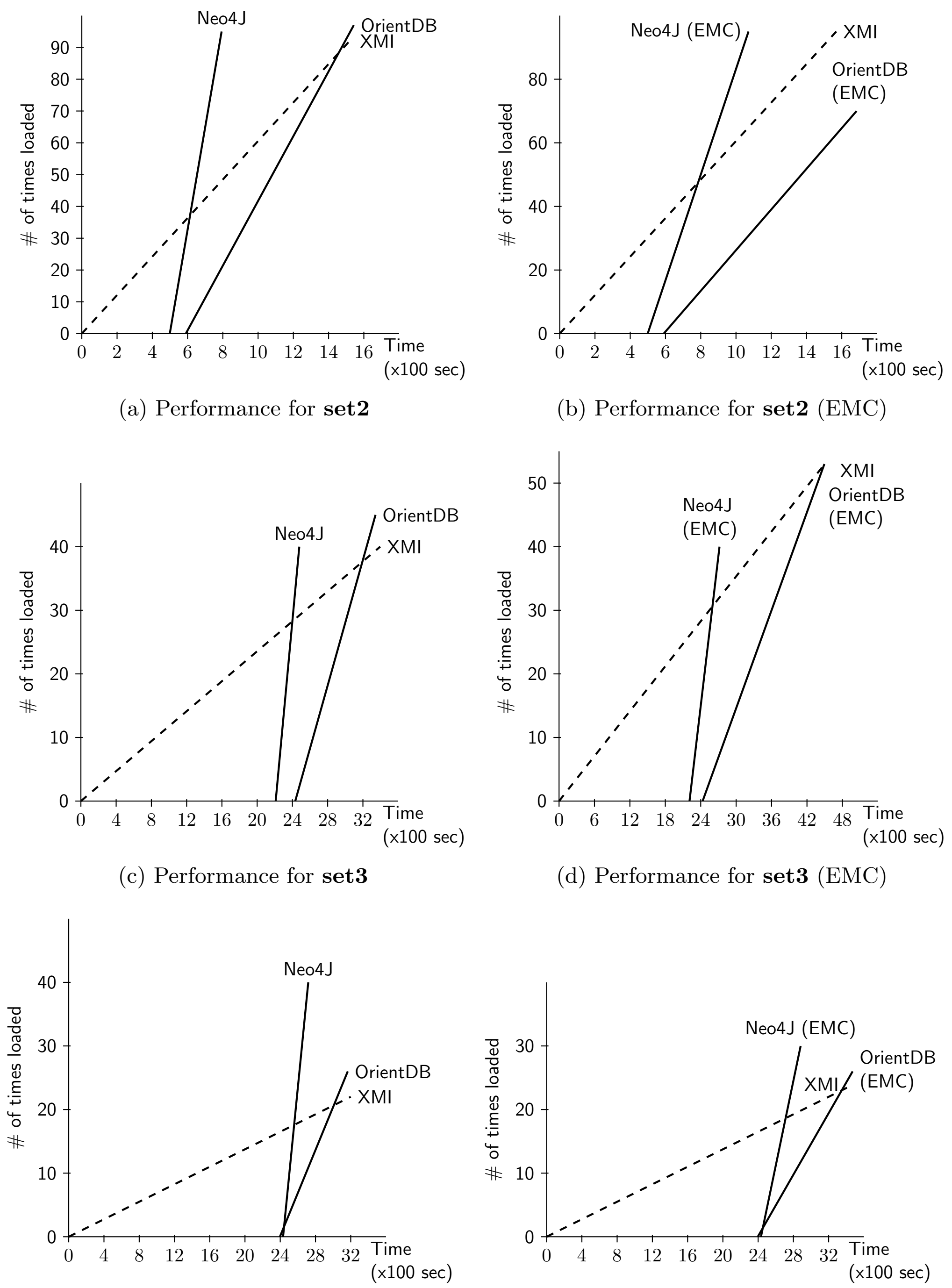


Figure 5.2.: Ratios of relative disk space used for the different persistence mechanisms

All three ratios, for large enough model sizes, tend to a constant. This constant is estimated to be 4.3 for XMI – Hawk[OrientDB], 2.9 for XMI – Hawk[Neo4J] and 1.45 for Hawk[Neo4J] – Hawk[OrientDB]. For smaller model sizes variables such as database-specific overhead seem to influence the ratios substantially (hence the larger ratios with respect to XMI for smaller models). Hence, for large enough models, we can expect an Hawk$^{[\text{OrientDB}]}$ store to be around 4.3x as large as its XMI file and a Hawk$^{[\text{Neo4J}]}$ store around 2.9x as large. Furthermore the results seem to show that Hawk$^{[\text{Neo4J}]}$ is more efficient in storing the data relative to Hawk$^{[\text{OrientDB}]}$, which can be expected as it handles references in a more lightweight fashion, as explained in Section 4.3.1.1. The ratio between Hawk$^{[\text{Neo4J}]}$ and Hawk$^{[\text{OrientDB}]}$ seems to indicate that for both databases their relative overheads discussed above are similar, but Hawk$^{[\text{OrientDB}]}$ is less efficient in that regard (with a 19.2% delta in the ratio between Hawk$^{[\text{Neo4J}]}$ and Hawk$^{[\text{OrientDB}]}$ at set0 and the one at set4).



(a) Performance for **set2**

(b) Performance for **set2** (EMC)

(c) Performance for **set3**

(d) Performance for **set3** (EMC)

(e) Performance for **set4**

(f) Performance for **set4** (EMC)

Figure 5.3.: Performance comparison for executing the Grabats Query from XMI through Hawk's chosen persistence mechanism using native and EMC querying

5.3.2. Benchmarking of Incremental Updating in Hawk

In this section, models from the Grabats benchmark are used to conduct performance tests for updating a Hawk model index. These models are mutated in order to simulate changes that are picked up by Hawk. This work has been published in [66].

Execution Environment Performance figures that have been measured on a PC with Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz, with 32GB of physical memory, a Solid State Drive (SSD) hard disk, and running the Windows 7 (64 bits) operating system are presented. The Java Virtual Machine (JVM) version 1.8.0_20-b26 has been restarted for each measure as well as for each repetition of each measure. In each case, 20GB of RAM has been allocated to the JVM (which includes any virtual memory used by the embedded Neo4J database server running the tests).

Model Manipulation In order to perform model manipulation operations, we used Epsilon's EOL. We decided to perform five model mutations (changes), which are representative of modifications performed in Java code. These mutations are performed by five EOL operations (shown in Appendix B, and available online[4]). The first operation covers model element deletion, by removing a random *TypeDeclaration* model element and all its children; the second operation covers model element creation, by inserting a new *TypeDeclaration* element as well as a new *SimpleName* element connected to it; the third operation covers complex model manipulation by adding a new *MethodDeclaration* element to a random *TypeDeclaration*, setting a new *Modifier* to this *MethodDeclaration*, and then setting the *returnType* of the *TypeDeclaration* to a newly created *SimpleType* linked with a random *SimpleName*; the fourth operation covers attribute change by altering the value a random attribute of a random *Modifier* of a random *MethodDeclaration*; the final one also covers attribute change by altering the attribute *fullyQualifiedName* of the *SimpleName* of a random *TypeDeclaration*. By using these operations in an EOL script we can change the model it is run on using operations often used in manipulation of Java code, such as deleting a Java class (as indicative of a real-world scenario where a developer is updating their Java model), as well as introducing randomness (for example, by randomizing which Java class is deleted each time) in order to try limit any bias in pre-selecting which elements are manipulated in each case.

[4] `https://github.com/kb634/mondo-hawk/blob/master/model_manipulations.eol`

5.3.2.1. Model Update Execution Time

Table 5.5 shows the average time taken to complete an update for the models produced by performing the model mutations presented above, on the original Grabats models. M(INS) represents the initial insert of the original Grabats models into an empty Hawk model index (using the naive insert process) and M(0%)–M(50%) represent the update time (from the original model) to one containing 0% to 50% content mutations. These mutations contain an equal degree of each mutation operation found in Section 5.3.2 so that the total change to the model ends up being N% of the original model contents. As such, each of the five mutation operations performs changes equal to $\frac{N}{5}$% of the original model elements; since some changes are addition/removal operations on model elements, the size of the resulting model is not the same as that of the original.

Table 5.5.: Update execution time results

	Execution Time (in seconds)									
Mutation	**Set0**		**Set1**		**Set2**		**Set3**		**Set4**	
	Naive	Inc.	Naive	Inc.	Naive	Inc.	Naive	Inc.	Naive	Inc.
M(INS)	9.96	n/a	18.69	n/a	118.19	n/a	291.06	n/a	346.46	n/a
M(0%)	16.61	2.70	45.72	6.07	-	63.96	-	162.52	-	224.85
M(10%)	16.82	3.94	47.71	10.45	-	94.59	-	247.94	-	292.86
M(20%)	17.76	4.71	48.22	11.53	-	115.86	-	364.94	-	417.50
M(30%)	18.93	5.66	50.60	15.04	-	145.56	-	440.78	-	622.51
M(40%)	21.84	7.04	54.73	18.79	-	165.48	-	781.35	-	-
M(50%)	22.09	7.97	60.21	20.92	-	193.41	-	-	-	-

For each case both the incremental and naive updates were tested and compared with one another. The naive update follows the process described in the prequel for naive insertion, after having had the currently indexed elements deleted from the model index. As the naive update process failed to terminate for the larger sets (Set2–Set4), figures for these models are not presented for the naive update process. The reason for this failure is that the Neo4J back-end runs out of memory when trying to delete the entire contents of the model index. This is an unforeseen limitation in the Neo4J database, as we require of it to perform a single transaction to delete the entire contents (as it does not support nested transactions but only flattened nested transactions, which only commit when the top-level transaction is closed) in order to maintain consistency between model versions. We also note that the incremental update fails to complete for 50% of Set3, 40% of Set4 and 50% of Set4. This is due to the fact that the magnitude of the change is so large that not enough memory is available for Neo4J to be able to fit this change in a transaction. The aim is to test the limits of Hawk, as such a system typically aims at collecting a large amount of fragmented models and not large monolithic ones; in the former case memory would not be an issue as it can be flushed after each file is updated. Furthermore, a 40% or 50% change on a model with millions of elements is not an expected use-case and again is presented to test the limits of the system.

These results suggest that the incremental update process is substantially faster than the naive approach, while also not compromising availability of the model index[5]. This can be largely attributed to there being no support for "mass deletes" in the model index, which ends up taking the majority of time needed for a naive update. The actual time taken for the incremental updates is promising as it scales linearly with the magnitude of the change in the model, giving us improvements of up to 78.10% decrease in execution time for a 10% model change and up to 65.25% for a 50% model change, averaging a 70.7% decrease in execution time over all of the comparable results[6].

5.3.2.2. Derived Attribute Update Execution Time

Results for the execution time of altering derived attributes are not presented as they would have to be compared to a baseline. Such a baseline would have been to use a naive approach whereby all derived attributes in the model index would have to be updated any time any model element or feature gets added/updated/removed. As Hawk is a model index working with large collections of models, likely with multiple derived attributes, the small overhead caused by the incremental process (storing and updating property accesses) was deemed negligible when compared to that of having to fully re-compute every single derived attribute any time anything in the index changes. As such, a naive approach was never implemented, so a meaningful comparison cannot be made.

[5] as it does not block any incoming queries which may need to be performed

[6] the 10 results from set0 and set1 that both naive and incremental approaches completed, disregarding the 0% change values as they are presented as a baseline

5.3.2.3. Threats to Validity

There are five observed threats to the validity of this approach:

- The model mutations performed may have influenced the results. We tried to limit this by performing multiple mutations in each case, all of which contain a random factor in them. An example involving real-world changes on a large collection of models can be found in Section 5.3.4.

- The percentage change of each model may not be indicative of real model change. We tried to limit this by exploring a large variety of changes ranging from zero to fifty percent of the original model. An example involving real-world changes on a large collection of models can be found in Section 5.3.4.

- The model sizes used for empirical evaluation may not be indicative. Hawk aims at handling large collections of (possibly fragmented) models (as defined in Section 3.4), thus we anticipate that the size of each fragment will not be orders of magnitude greater than the test models. An example using a large collection of evolving small models can be found in Section 5.3.4.

- This version of Hawk used an integer representation for the signatures, which has a chance for collisions;

  this chance tends to 1 in 4.29 billion for non-trivial Strings. In all of the empirical tests performed no clashes have been observed, which gives us some confidence that the approach should be used for performance reasons. It is worth noting that in later versions of Hawk a SHA-1 signature is being used to further decrease the chance for collisions (as of October 2015, no actual collisions are publicly known for any process using SHA-1).

- The last one is regarding the correctness of the incremental algorithm. While this is not formally proven, empirical tests comparing the model index state after an incremental update with that of the original naive update, previously used in Hawk (for the same changes), provided the same results for all of the mutated models where both the incremental and naive updates completed. Sections 5.3.4.2 and 5.4.4.2 present various further empirical tests performed to evaluate the correctness of this algorithm.

5.3.3. Benchmarking of Derived and Indexed Attributes in Hawk

As Hawk supports the creation of derived attributes as well as the indexing of attribute values, this Section published in [67] evaluates the performance benefit of using such features for query execution.

Execution Environment Performance figures that have been measured on a PC with Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz, with 32GB of physical memory, a Solid State Drive (SSD) hard disk, and running the Windows 7 (64 bits) operating system are presented. Java Virtual Machine (JVM) version 1.8.0_20-b26 has been used and both the database connection to Neo4J and the Epsilon driver have been restarted for each measure as well as for each repetition of each measure. In each case, 20GB of RAM has been allocated to the JVM (which includes any virtual memory used by the embedded Neo4J database server running the tests).

5.3.3.1. Derived Attribute Definition

In order to effectively use (database) indexed attributes to optimize the Grabats query execution, due to the nature of the Grabats metamodel, we have to also use the derived attributes feature hawk offers. This functionality allows for any applicable EOL expression to be used to evaluate the values of derived attributes, during the model indexing process. The derived attributes used are the following:

- isPublic (in MethodDeclaration), which denotes whether the *MethodDeclaration* is public (has a modifier with attribute public = true), defined by:

  ```
  self.modifiers.exists( m:Modifier | m.public == true )
  ```

- isStatic (in MethodDeclaration), which denotes whether the *MethodDeclaration* is static (has a modifier with attribute static = true), defined by:

  ```
  self.modifiers.exists( m:Modifier | m.static == true )
  ```

- isSameReturnType (in MethodDeclaration), which denotes that the return type of the *MethodDeclaration* is the same type as its containing *TypeDeclaration* (both have the same name), defined by:

```
self.returnType.isTypeOf(SimpleType) and
self.eContainer.isTypeOf(TypeDeclaration) and
self.returnType.name.fullyQualifiedName == self.eContainer.name.
   fullyQualifiedName
```

- singleton (in TypeDeclaration), which denotes that the *TypeDeclaration* fulfills all of the criteria posed by the Grabats query (has at least one method which is public and static and has this type as its return), defined by:

```
self.bodyDeclarations.exists(
  md:MethodDeclaration |
    md.modifiers.exists( mod:Modifier | mod.public == true ) and
    md.modifiers.exists( mod:Modifier | mod.static == true ) and
    md.returnType.isTypeOf(SimpleType) and
    md.returnType.name.fullyQualifiedName == self.name.
         fullyQualifiedName
)
```

5.3.3.2. Query Definition and Execution Time

Table 5.6 shows the results for performing the first Grabats 2009 query on the various persisted models. It is worth noting that the impact of keeping these derived and indexed attributes in Hawk (in terms of additional time needed during model insertion into Hawk) was negligible (less than 1% of total insertion time in each case) in this case-study.

For these tests five queries have been written in EOL (Q1 – Q5):

- *Q1* reads:

```
TypeDeclaration.all.select(
  td|td.bodyDeclarations.exists(
    md:MethodDeclaration|
      md.modifiers.exists(mod:Modifier|mod.public==true) and
      md.modifiers.exists(mod:Modifier|mod.static==true) and
      md.returnType.isTypeOf(SimpleType) and
    md.returnType.name.fullyQualifiedName == td.name.
      fullyQualifiedName
  )
)
```

Table 5.6.: Grabats Query execution time results

	Execution Time (in seconds)				
Model	Original	DerivedOnly		Derived&Indexed	
	Q1	Q2	Q3	Q4	Q5
Set0	0.040	0.031	0.021	0.062	0.020
Set1	0.081	0.051	0.030	0.072	0.042
Set2	1.850	1.282	0.262	0.302	0.072
Set3	3.664	2.688	0.732	0.852	0.080
Set4	4.124	2.847	0.772	0.902	0.081

This query (Q1) is the basic Grabats query using the original metamodel to insert the relevant models into Hawk. As such it only uses attributes found in the unaltered JDTAST metamodel and is used as a baseline for comparison.

- *Q2* reads:

```
TypeDeclaration.all.select(
  td|td.bodyDeclarations.exists(
    md:MethodDeclaration |
      md.isPublic == true and md.isStatic == true and
      md.isSameReturnType == true
  )
)
```

This query (Q2) assumes that relevant derived attributes have been created by Hawk during insertion. As such, it uses attributes found in the original JDTAST metamodel as well as the derived attributes 'isPublic', 'isStatic' and 'isSameReturnType'.

- *Q3* reads:

```
TypeDeclaration.all.select( td | td.singleton == true )
```

  This query (Q3) assumes that relevant derived attributes have been created by Hawk during insertion. As such, it uses attributes found in the unaltered JDTAST metamodel as well as the derived attribute 'singleton'.

- *Q4* reads:

```
MethodDeclaration.all.select(
  md |
    md.isPublic == true and md.isStatic == true and
    md.isSameReturnType == true
)
.collect( td | td.eContainer )
```

  This query (Q4) assumes that relevant derived attributes have been created by Hawk during insertion and is a re-written form of Q2 which takes advantage of (database) indexing of the attributes 'isPublic', 'isStatic' and 'isSameReturnType' in order to optimize performance. Using the eContainer call (in the spirit of EMF's homonymous method) we can get the *TypeDeclaration*s that the *MethodDeclaration*s are contained in and thusly report the same output as Q2.

- *Q5* Is the same as Q3 as it can take advantage of attribute indexing as-is (should the relevant custom indexes exist in the store).

Similar to the discussion in [67] (which compares the use of derived attributes in an older version of Hawk), Q2 and Q3 (which only use the derived attribute feature of Hawk) perform a lot better than the original Q1, with observed improvements of 22.5% – 37.0% when comparing Q1 and Q2 and 47.5% – 85.8% comparing Q1 and Q3.

Comparing Q2 with Q4 (as they have access to the same derived attributes, but in Q4 they are also (database) indexed), we note that for the small model sizes the overhead of using database indexing results in similar execution times to that without it but for the larger models we note a large improvement (of 68.3% – 76.4%).

Comparing Q3 with Q5 (as they have access to the same derived attributes, but in Q5 they are also (database) indexed), we can also see that for the small model sizes the overhead of using database indexing results in similar execution times, but for the larger models we note a large improvement (of 72.5% – 89.5%).

These results support the idea that for sufficiently large model sizes the targeted use of custom indexes for attributes can greatly improve query performance of certain types of queries, while not compromising the querying of smaller models.

5.3.4. Benchmarking of Continuous Model Updates in Hawk

The BPMN MIWG repository presented in Section 5.2.2 is used to evaluate the performance and correctness of Hawk, when dealing with a large collection of small, evolving models. This section presents the methodology used and the results obtained during this experiment.

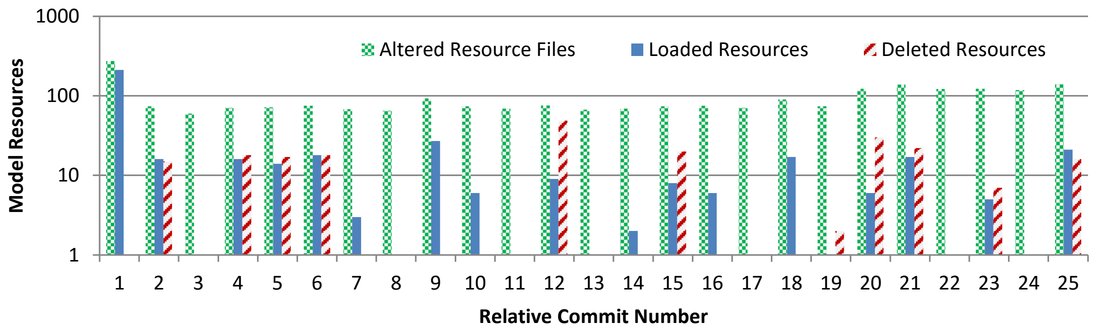
Execution Environment Performance figures that have been measured on a PC with Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz, with 32GB of physical memory, a Solid State Drive (SSD) hard disk, and running the Windows 7 (64 bits) operating system are presented. Java Virtual Machine (JVM) version 1.8.0_65-b17 has been used and the database has been re-created from scratch for each repetition of each measure. In each case, 3GB of RAM has been allocated to the JVM (which includes any virtual memory used by the embedded Neo4J database server running the tests, but NOT any virtual memory used by OrientDB as this memory does not use the Java Heap).

Methodology As the BPMN MWIG repository contains hundreds of commits (490 as of November 2015), a reasonable heuristic had to be created for deciding which commits to use in order to emulate a real-life model evolution scenario. The following data was gathered when analyzing the various commits:

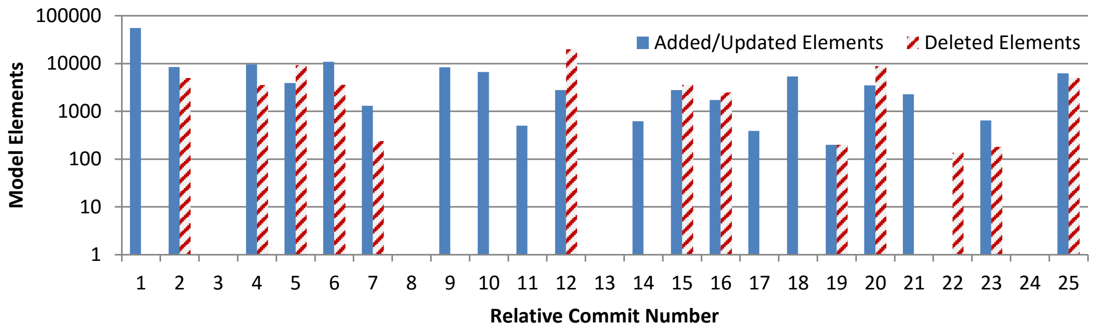
- There were 485 commits which resulted in at least one BPMN file to be changed
- There were 457 commits with over 10 BPMN files changed
- The were 397 commits with over 50 BPMN files changed
- There were 122 commits with over 250 BPMN files changed

Using this data, commits that resulted in the change of over 250 BPMN files were considered, and out of those commits only one in every 5 was selected, resulting in 25 commits being flagged.

Using these flagged 25 commits, Hawk was initially provided with the first (chronologically) commit and then each successive commit to synchronize with. Figure 5.4 shows the changed (file-based) model resources/elements resulting from each successive commit, with respect to the previous one (the initial commit is with respect to an initial empty store). Note that in various commits the loaded and deleted resources do not add up to the total altered resources (Figure 5.4a), as some model files were either malformed or contained metamodels not present in Hawk, and hence the EMF-based Hawk BPMN model factory was not able to load the relevant resource into memory.



(a) Number of Resources Loaded in Each Commit



(b) Total Number of Elements Affected by Each Commit

Figure 5.4.: BPMN benchmark model change results

5.3.4.1. Update Performance Results

Figure 5.5 displays the execution time of each synchronization process for each commit. As expected the first commit takes the most time as it has over 200 files (and successfully loaded resources) to fully insert into Hawk; the execution time of subsequent commits is substantially lower as they only need to handle 17 model files on average (either changed (and successfully loaded into a resource) or removed), and in some cases can perform an incremental update (as 37 of the 194 subsequently loaded resources could be incrementally updated). Compared to one another both back-end technologies perform very similarly, with a difference of ~30% in overall time taken to complete the benchmark.

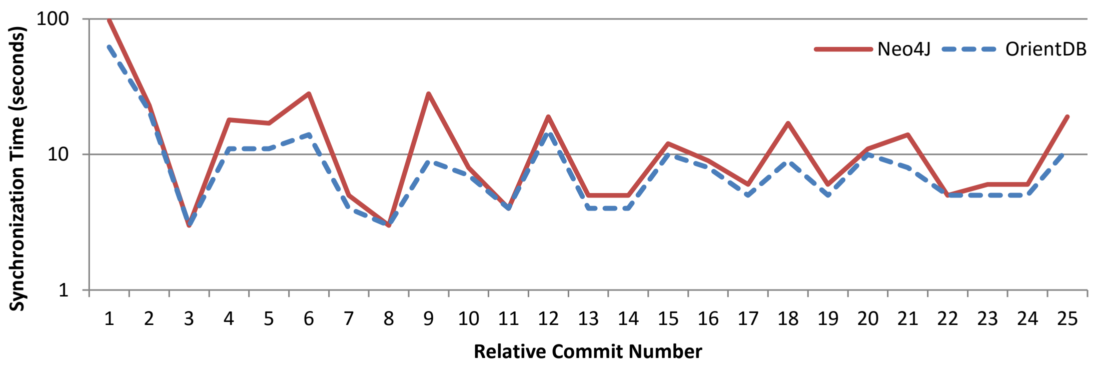


Figure 5.5.: BPMN benchmark execution time (Hawk synchronizing with each commit)

Figures 5.6 and 5.7 display the memory use of (a representative run of) Hawk throughout this benchmark[7] (using Neo4J and OrientDB respectively). As Java contains three areas of memory (eden, survivor and old gen)[8] they are all presented here, but for the sake of simplicity we will only consider the sum of all of them at each given time period (i.e. the total memory use of the program at any given time). Line *L1* denotes the time when the synchronization with the first commit finishes and line *L2* denotes the time when the synchronization with the last commit finishes. Looking at the time-line between the start of the benchmark and *L1*, Hawk's memory use is consistently low in both cases, as Hawk only loads one resource at a time to insert it into its back-end (so the Java garbage collection can keep discarding the old objects used for managing previous files). In the case of Hawk[Neo4J], the memory used is largely for the embedded MMIO

[7] using the YourKit Java profiler: `https://www.yourkit.com/`

[8] `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`

of Neo4J (hence it rapidly reaches a peak (of around 250MB) and remains there for the duration of the update), while in Hawk[OrientDB] (whereby the MMIO is not embedded into the Java Heap but directly uses system RAM), the memory use rises consistently during the update as OrientDB keeps strong references to all of its newly formed elements; Neo4J can be seen as more efficient in this case as it does not need this memory but can quickly dispose of newly formed objects to disk, seen by the intermittent peaks in memory use during the update process.

Looking at the time-line between *L1* and *L2*, Hawk's memory periodically spikes as numerous batch inserts, deletions or incremental updates are called (incremental updates require data from disk to be retrieved so that the current version of the model indexed can be compared to the changed version of the new commit). In the case of Hawk[Neo4J], the memory used keeps resetting down to near its MMIO baseline every time the Java garbage collector deems necessary, while in the Hawk[OrientDB] case it keeps resetting down to nearly 0 as the MMIO is not embedded into the Java Heap.

Finally, the time-line between *L2* and the end of the recording of the benchmark shows a constant memory footprint when no changes are detected by Hawk. The memory keeping the Neo4J caches of recently accessed nodes remains constant and so does the memory used by OrientDB keeping soft references to its MMIO that is outside the Java Heap.

5.3.4.2. Update Validation Results

During the execution of this benchmark the Hawk validation listener (Section 5.1.1.2) was used to analyze the contents of the model index after each synchronize and compare them with any changed model files involved in that commit. For each of the 25 commits the validator reported that Hawk had a consistent state with the model files, which provides empirical evidence that Hawk's batch injection and incremental update processes are both correct.

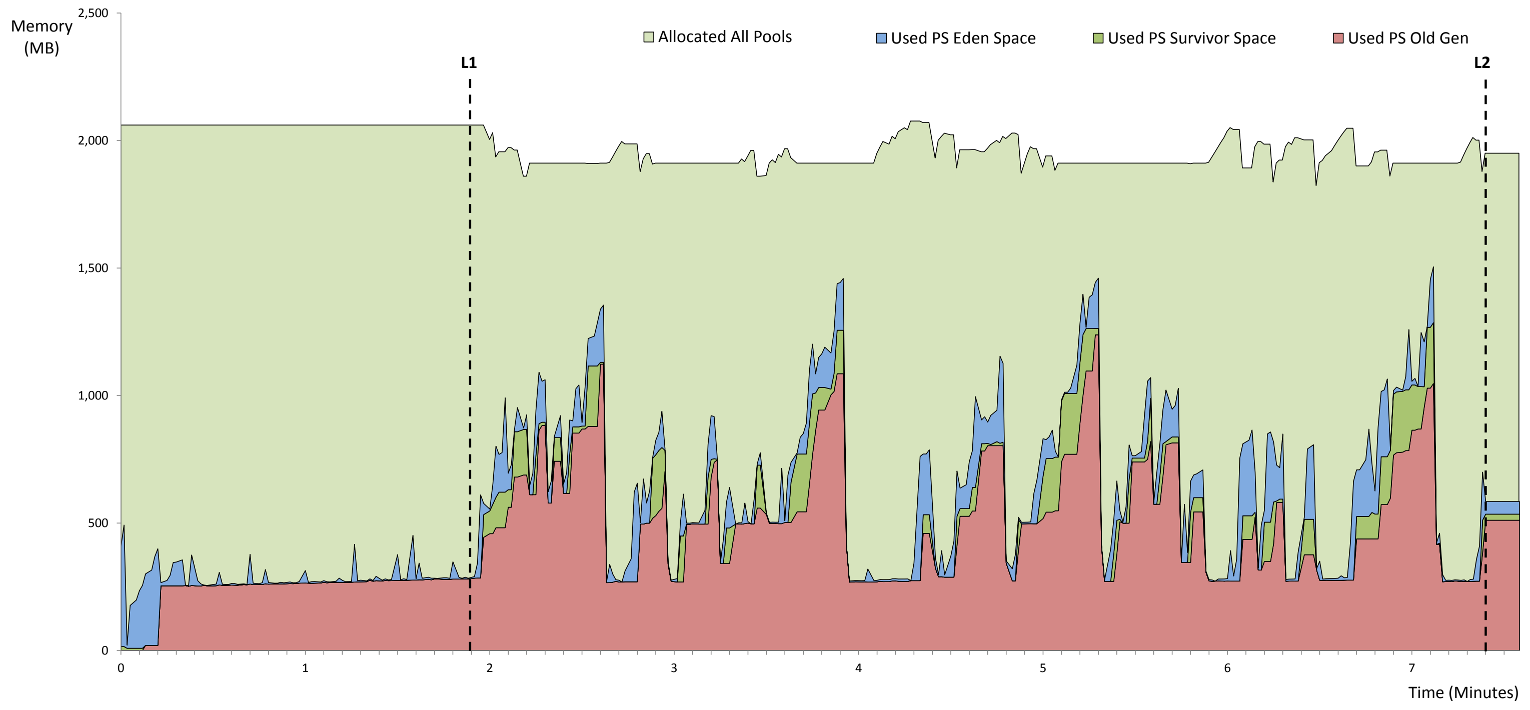


Figure 5.6.: Memory graph of full BPMN benchmark execution (Hawk[Neo4J])

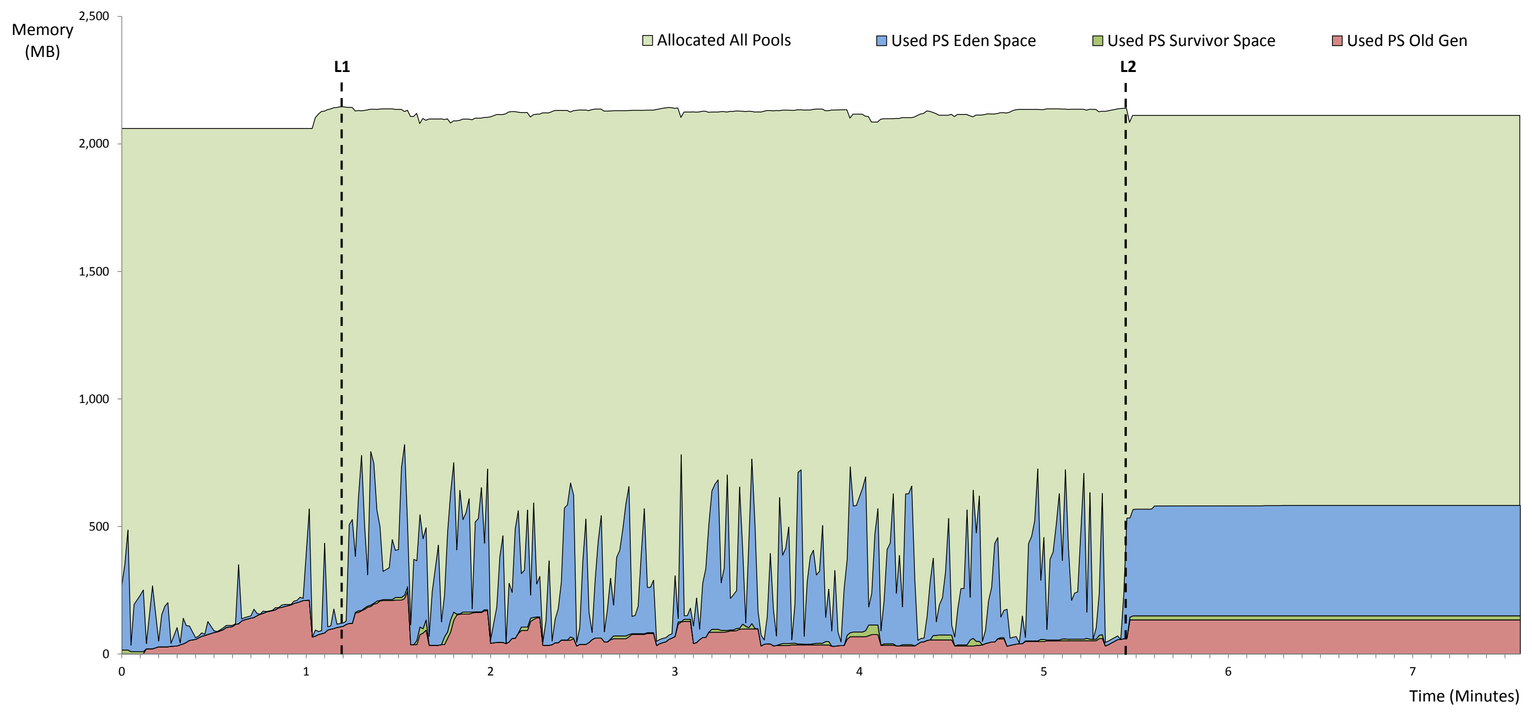


Figure 5.7.: Memory graph of full BPMN benchmark execution (Hawk$^{[\text{OrientDB}]}$)

5.4. Hawk Tool Integration

Hawk has been integrated with several external MDE tools (in collaboration with various MONDO project partners – detailed in each section). This section provides insight into the integration and the reasoning behind it.

5.4.1. Epsilon Integration

The first tool integrated with Hawk was Epsilon, in order to provide it with a query engine (as discussed in Section 4.3.6.2). This allowed for empirical tests for comparing the performance of querying Hawk using a native Java approach with respect to using the Epsilon Object Language as a query expression language (Section 5.3.1.2). Even though this effort was not done collaboratively, it is mentioned here for completeness.

5.4.2. Exposing Hawk as an EMF Resource

As Hawk deals primarily with EMF-based models provided by its various model resource factories, and to allow easy integration with any EMF-compatible tool and language, Hawk can be exposed as a read-only EMF resource. When compared to a usual EMF resource, Hawk provides various useful optimizations to improve performance:

Lazy Loading An important ability of a Hawk EMF resource is to load its contents in a lazy manner. This functionality results in partial traversal of the Hawk index, based on various options such as lazy loading only of root elements, lazy loading only of proxies to model elements until their features are requested and lazy loading only of attributes of elements until the targets of references are needed.

Editor Integration A Hawk EMF resource can be lazily loaded in the EMF-based Exeed tree-based editor of Epsilon in order to be incrementally visualized, without having to front-load the entire model beforehand. This allows for large models that would normally take a long time to load, or would not load at all (as they would require more memory than that available to the tool in order to be loaded from disk), to be partially loaded using the various lazy loading strategies mentioned above.

Convenience Methods Hawks EMF resource also provides various convenience methods that can be used to improve performance of using such resources for various model management operations:

- fetchNodes(eClass,fetchAttributes), which retrieves all model elements of type *eClass* and returns it as an *EList* of *EObjects*. Optionally it can also fetch all the attribute values of these elements. This method is similar to the *Type.getAllOfType()* method Epsilon provides.

- fetchValuesByEClassifier(dataType), which lists all values of *EStructuralFeature*s of type *dataType*.

- fetchTypesWithEClassifier(dataType), which maps all *EClass*es with a *EStructuralFeature* of type *dataType* to a List of their relevant *EStructuralFeature*s.

- fetchValuesByEStructuralFeature(feature), which maps all *EObject*s with a specific *EStructuralFeature* to their value of this *EStructuralFeature*.

This work was done in collaboration with Dr. Antonio Garcia Dominguez.

5.4.3. Remote Query API (using Apache Thrift)

In order to connect to Hawk remotely, a remote API has been created and implemented using Apache Thrift[9]. This feature allows for Hawk to connect to remote instances (of Hawk model indexers) and to perform the same operations as on local instances.

Notifications provided by remote instances are handled using Apache ActiveMQ Artemis[10]. This remote queue allows for handling of disconnects and compressing large change notifications to reduce network overhead.

Results of queries are serialized using Thrift and can be de-serialized on the client side into various forms such as EMF resources. This remote Hawk EMF resource provides similar functionality to the local one described in Section 5.4.2, allowing for lazy loading and resolution in order to reduce network overhead. This architecture can be seen in Figure 5.8.

This work was done in collaboration with Dr. Antonio Garcia Dominguez.

[9] `https://thrift.apache.org/`
[10] `https://activemq.apache.org/artemis/`

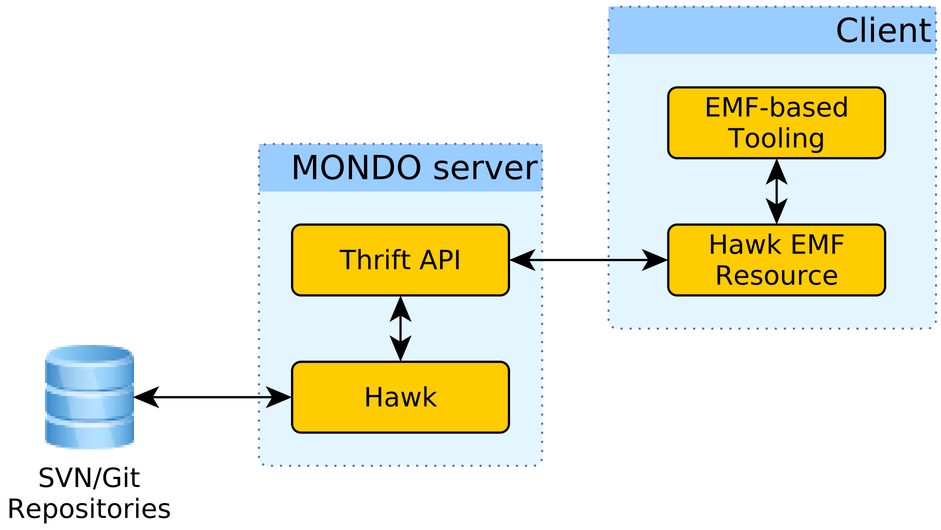


Figure 5.8.: Architecture of Hawk's remote EMF API

5.4.4. EMF IncQuery Integration

As part of the efforts for the integration deliverable of the MONDO project, Hawk has been integrated with IncQuery. This section briefly introduces this tool and describes the evaluation opportunities it opened for Hawk.

5.4.4.1. EMF IncQuery

EMF IncQuery[11] offers an incremental graph query engine on EMF-based models. Using its declarative query language, users are able to formulate graph pattern matching queries [77].

The tool uses an adaptation of the RETE algorithm [78] in order to incrementally calculate results of queries based on previous results and the collection of changes since these results; it caches current results in memory and keeps them up to date with any changes made to the model.

IncQuery also offers a *BaseIndexer* API[12] that can be used to optimize the connection of any EMF-based persistence layer so that the engine can take full advantage of the underlying store's capabilities. As such, implementing this API for Hawk (through Hawk's

[11] `https://www.eclipse.org/incquery/`
[12] `https://wiki.eclipse.org/EMFIncQuery/UserDocumentation/API/BaseIndexer`

EMF resource implementation (Section 5.4.2)) has allowed Hawk model indexes to be exposed to IncQuery and used for answering queries performed through its language.

5.4.4.2. The Train Benchmark

As part of the evaluation of IncQuery, a custom benchmark has been created. This benchmark's main goal is to measure the execution time of graph-based querying, with emphasis on incremental re-evaluation of queries as models evolve[13]. It has been used in various publications including [79–81]. It uses the metamodel shown in Figure 5.9[13] and systematically generates instance models of incrementally larger sizes, using various randomization points such as exact number of elements and cardinalities.

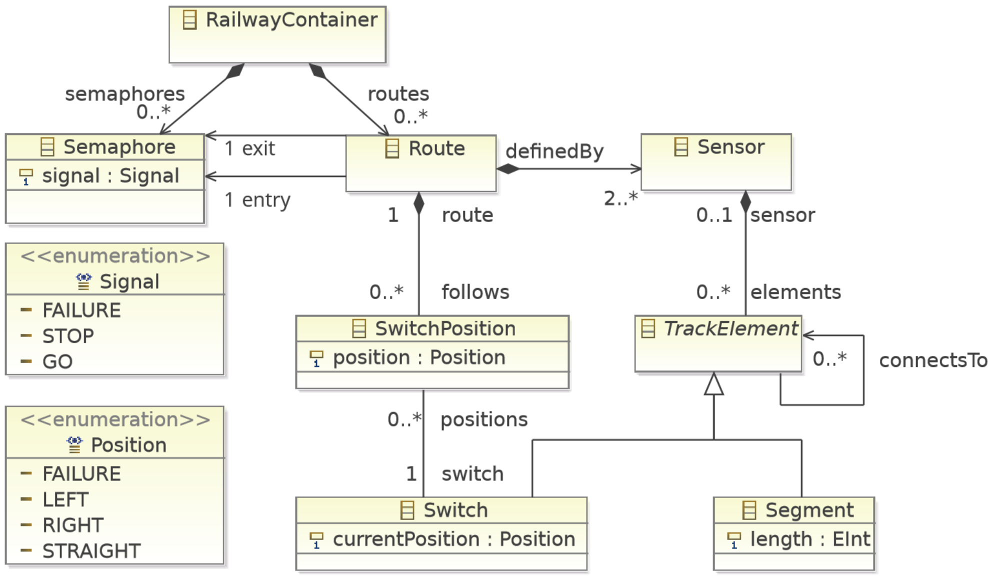


Figure 5.9.: The train benchmark metamodel

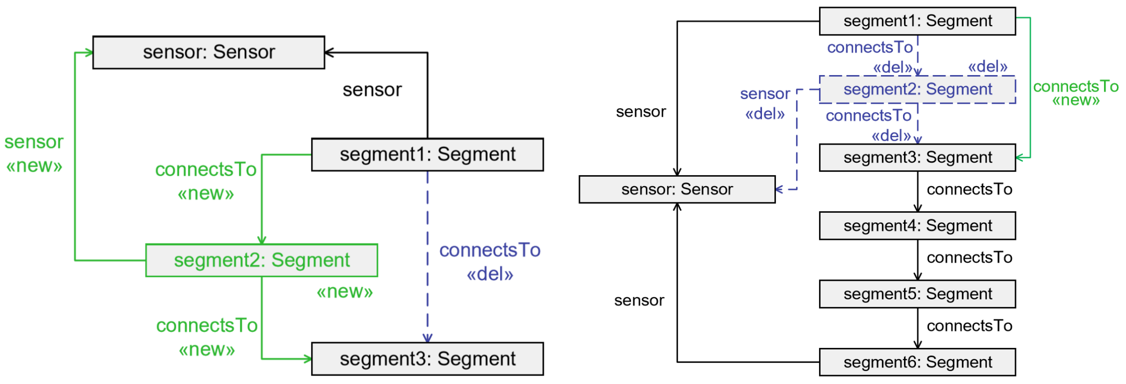
These models are then used to simulate a real-world scenario where faults emerge in the railway system and they are repaired by engineers. This sequence of fault injection and repair comprises model evolution (in the form of in-place model transformations) and the tools used are expected to provide consistent results in each case (as well as being performant).

[13] `https://www.sharelatex.com/github/repos/FTSRG/trainbenchmark-docs/builds/latest/output.pdf`

Hawk Correctness With respect to the Hawk integration, performance considerations have not yet been made, but the experiments performed do provide evidence of the correctness of Hawk's incremental update process as well as that of the EMF resource API it offers.

The train benchmark offers a collection of 24 JUnit[14] test cases each of which tests and evaluates the performance and correctness of different operations performed on a specific train model with respect to a set of well-formedness constraints (for example the ConnectedSegments test case injects faulty sensors with more than five segments (Figure 5.10a) and then repairs some of these sensors (Figure 5.10b)). As Hawk has passed all of these test cases, further empirical evidence of Hawk's correctness is obtained both for the contents of the model indexer after initial and incremental updates are performed, as well as for the results obtained when using Epsilon as a query engine.

This work was done in collaboration with Dr. Antonio Garcia Dominguez and Gábor Szárnyas.



(a) Inject transformation (b) Repair transformation

Figure 5.10.: ConnectedSegments inject and repair transformations

5.5. Additional Drivers for Hawk

Even though Hawk's primary drivers work with SVN, EMF, Neo4J and Epsilon; various other drivers have been developed that can support Hawk's extensibility argument:

[14]`http://junit.org/`

5.5.1. Alternate Version Control Managers

- As presented in Section 4.3.3.2, Hawk supports indexing models found on the local machine through the *LocalFolder* driver.

- As presented in Section 4.3.3.4, Hawk can index models located in a local eclipse workspace using the *Workspace* driver.

5.5.2. Alternate Model Factories

- As presented in Section 4.3.4.2, Hawk is able to index IFC models stored in XML or STEP formats through its *IFC*ResourceFactory* drivers.

- As presented in Section 4.3.4.2, Hawk is able to index Modelio UML models stored in Modelio's XML format using its *Modelio*ResourceFactory* drivers.

5.5.3. Alternate Persistence Technologies

- A driver for OrientDB has been developed (and kept up to date in collaboration with one of the MONDO project partners – Dr. Antonio Garcia Dominguez), and can be used as an alternative back-end for Hawk. Section 5.3.1 compares the performance of Neo4J and OrientDB for indexing and querying a set of large models and Section 5.3.4 presents a similar comparison for indexing a large collection of smaller models and their evolution over time.

5.6. Research Hypothesis Evaluation

In this section the empirical results obtained via extensive benchmarking on Hawk were presented; these benchmarks tested the various aspects of the system in terms of correctness and performance. The results provide sufficient confidence that the research hypothesis can be validated, showing that the overhead of computing model-element-level queries over large (collections of) models stored in a file-based VCS can be significantly reduced using a non-invasive model-indexing system orthogonal to the specific VCS or model representation format. The research objectives presented in Section 3.3 have been met:

(i) The prototype being evaluated (Hawk) provides sufficient validation of the research hypothesis as the evaluation results obtained as part of this research support the argument that a model indexing system can help reduce the query overhead on large (collections of) models.

(ii) An appropriate back-end has been selected, showing promising results in terms of model insertion, model update and query execution performance, when compared to various alternatives evaluated as part of this work, as well as to current state-of-the-art model persistence approaches.

(iii) Hawk offers support for three modeling technologies (EMF, BIM IFC, Modelio UML) and four model persistence formats (EMF XMI, IFC STEP, IFC XMI, Modelio XML).

(iv) Hawk offers support for SVN and Git repositories as well as local folders and Eclipse workspaces.

(v) Results on query execution show Hawk's ability to offer a performant querying environment for both large models and large collections of model fragments.

(vi) Results on index content (insertion and updating) performance and correctness offer sufficient evidence of meeting Hawk's requirements of correctness and incrementality.

(vii) Results show Hawk competing with current state-of-the-art modeling tools, offering better results in most cases, for the queries in the benchmarks performed as part of this work.

These results encourage further development of this approach in the future, discussed in Section 6.4.

6. Conclusions

This chapter summarizes the findings of this thesis, presents the various contributions it has made to the field of software engineering and discusses various paths for extending this work, which can be taken in the future.

6.1. Summary

In this work we presented Hawk, a model indexing framework aimed at tackling some of the scalability issues identified in the field of MDE. Chapter 2 introduced the reader to the field of MDE and to file-based version control; it reviewed many of the current state-of-the-art tools and technologies used in MDE today, and discussed each of their capabilities; finally it delved into model querying, one of the important aspects of MDE commonly bottle-necking the process when using very large models (or large collections of inter-connected models). Chapter 3 synthesized the findings of the review, presented the research hypothesis this work is based upon and introduced the reader to the concept of model indexing; it discussed the research objectives and identified the scope of the research. Chapter 4 delved into the inner workings of the proposed framework by detailing the architectural, design and implementation decisions made, while presenting the various key processes, procedures and algorithms used. Finally, Chapter 5 detailed a thorough empirical evaluation of the various aspects of Hawk that provided confidence that its intended capabilities (both functional and non-functional are able to be met).

6.2. Contributions of the Thesis Research

This section presents the various contributions to the field of software engineering and in particular to the work on scalability on MDE this work has offered. It is split into two subsections, one for novel tools/techniques developed as part of this work and one for other important side-products the research has produced.

6.2.1. Novel Tools and Techniques

As part of this work, several novel approaches have been proposed for tackling the various scalability aspects presented.

6.2.1.1. Heterogeneous Model Indexing Platform

The primary contribution of this research is in the form of Hawk, a scalable heterogeneous model indexing framework. It provides validation of the research hypothesis of this work and offers an alternative to work alongside a widely-used paradigm in MDE:

Scalability. Hawk offers a scalable solution to indexing large collections of models found in VCS by using graph-based NoSQL stores to persist the data as a property graph. Empirical results presented in this work have provided confidence that the approach taken is effective both in terms of persisting and in terms of querying models of increasing sizes and numbers.

Modeling Technology Heterogeneity. Hawk supports indexing of EMF-based models, as well as offering the capability for adding drivers for conceptually any modeling technology to be used for providing models to Hawk. As examples of alternate technologies Hawk can handle, prototype drivers for Modelio UML and IFC BIM models have been created, in collaboration with the MONDO research partners.

Extensibility. Hawk offers extensibility mechanisms for adding new drivers for alternate VCS in which models can be stored in, as well as alternate back-end stores and model updaters. Multiple drivers have been created to validate this claim, including Local-Folder and Workspace drivers as well as one for OrientDB. Furthermore, Hawk has been integrated with various other modeling tools such as Epsilon (for querying), IncQuery (for validation) and EMF (for allowing any EMF-based tool to work with Hawk, for example the EMF tree editor), further validating this architecture.

6.2.1.2. Incremental Updating of Model Indexes

Another novel contribution is that regarding the use of incremental updates in model indexes. This work has presented an approach using model element signatures for efficiently calculating the changed elements between two versions of a model and hence updating the model index only with the elements which have actually changed. This automated process allows for a semantic model-level incremental update to be performed on model indexes, for which the empirical data gathered and presented here shows promising results.

6.2.1.3. Incremental Updating of Derived Attributes

A further contribution of this work is the novel approach used for incrementally recalculating only the relevant derived attributes after each model index update. By only having to re-execute the expressions of derived attributes known to have been affected by a change, a large overhead of redundantly having to re-evaluate unchanged attributes is eliminated.

6.2.2. Notable side-products

While this work produced various novel tools, algorithms and techniques, a large part of it offered interesting insights and quantitative results which, even though not novel themselves, offer useful insight into the various scalability concerns raised in this research.

6.2.2.1. Evaluation of Model Persistence Technologies

This research, alongside the work presented in [41] (which was in collaboration with one of the MONDO research partners), provides a thorough evaluation of various novel persistence technologies in the context of MDE; it offers insight into the functionality and limitations of the various state-of-the-art tools reviewed.

6.2.2.2. Use of Derived and Indexed Attributes in Model Indexes

This research has presented the use of derived and indexed attributes to improve the performance of certain classes of queries on model indexes.

Database Indexed Attributes. Indexed attributes allow for efficient equality and comparison operations to be performed on relevant attributes of all elements of a type without having to go to the persisted model index; with empirical data showing a large performance gains when such attributes are used.

Derived Attributes. Derived attributes can be computed using one of Hawk's query engines and stored in their relevant model element. Such attributes allow not only for the ability to quickly retrieve the solution to a complex expression on demand but can provide significant performance benefits when used in certain classes of query, as supported by the empirical results obtained as part of this research.

6.2.2.3. Scalable Model Querying

The model indexing framework presented in this work lists as one of its primary requirements having the ability to efficiently answer global queries on large collections of models it indexes. In this work, an OCL-like query language is used to investigate the performance of querying such a model index as well as through its various optimizations (using indexed and derived attributes). Empirical results presented here support this claim as the observed execution times for a complex query performed on models of increasing sizes are very promising.

6.3. Applications

Every use-case of MDE is different, whether regarding the domain in question, the environment surrounding the specific effort or the concrete set of scenarios tackled. As such, it would be remiss to say that a tool, whether Hawk or other MDE-based framework would fit every (or even most) situations; each tool has its own unique characteristics and offers a different approach to managing models.

In particular, positioning Hawk in today's MDE landscape would leverage its non-invasive, orthogonal approach to one of the prominent current practices used today, namely storing file-based (possibly fragmented) models in a VCS. As such, modeling tools which save models as files (like EMF), can use a Hawk model index to perform efficient global queries, without having to obtain or load the models locally. Hawk's extensible infrastructure can be used for providing new drivers on demand, if the modeling technology is not currently supported by Hawk.

For example, consider a company managing a large model of their evolving web-site, stored as multiple XMI files on an SVN server, and generating the code used for running it using model transformations (e.g. in the spirit of Jekyll[1]). They can use Hawk as the

[1] `https://jekyllrb.com/`

source for re-executing the transformation every time the model changes. This would leverage the efficient querying of Hawk to provide the transformation engine the data it needs in a much more efficient manner than having to load all of the (remotely stored) XMI files into memory and then accessing the required elements.

Alternatively, in an offline collaborative modeling scenario where a large fragmented model representing the building details of a football stadium is being manipulated by five teams of developers and stored as a collection of IFC STEP files in a remote VCS repository. In this example, each team would only work with a specific subset of the model and hence would only check out and load those model fragments it uses. Now should any team need to perform a query that may require access to the remainder of the model (such as one asking how much is the total projected cost of the stadium in the current model version) they can use Hawk to do so without having to manipulate the model, or having to check out or load any additional files.

A tool such as CDO would be beneficial when the user wishes to actually persist their models in a model repository and hence leverage the benefits it provides, but is not limited to using XML-based or other file-based MDE tooling. For example in an online collaborative modeling scenario where a large model representing the design of a power plant is being currently developed by five teams of developers, CDO can offer a centralized location where all teams can concurrently update and query the model as it is being developed.

A tool such as NeoEMF can provide a database persistence format for EMF models and would be useful when the users do not need a repository but want to store large models in an efficiently queriable format which offers lazy loading of elements. For example a company owning a large model representing its internal business processes can use NeoEMF to store it, so that whenever it has to query it for information, it can do so without having to load the entire model into memory, with the limitation that version control would have to be handled separately.

Tools such as Modelio, MagicDraw and MetaEdit+ offer extensive graphical UI functionality for their respective domains and facilitate the creation of UML or domain-specific models while also offering various alternatives for how these models can be persisted or exported. For example a business creating a new UML model of their envisioned structure can use these tools to create it in a graphical and domain-specific manner, hence allowing non-experts such as managers to understand it and contribute their expertise to its development.

Finally, a tool directly using EMF and its default XMI serialization can be beneficial when a model, for example of a legacy system which needs to be analyzed (and is already stored as an XMI file), is used as a one-time source of data (or used very sparsely), for a single query or transformation. In this case, the overhead of migrating this model into another model persistence format, a model repository or a model index would surpass the actual time needed to load this model into memory and run the relevant query or transformation on it. Furthermore, XMI persistence of a small model (in the order of thousands of model elements or smaller) will cause a manageable overhead when loaded into memory (both in terms of memory and time, when considering contemporary hardware) such that having to migrate it to another system may introduce additional needs (such as training to use the new system, or financial overheads) that can be avoided without much actual performance loss in practice.

Any effort spent on deciding which technology fits the users specific scenario is well spent, as it will inevitably benefit them in the long term, when contrasted to attempting to make a specific technology fit these needs. Table 6.1 summarizes the approaches, categorizing the tools reviewed in Section 2.2 (as a summary of Table 2.1).

Table 6.1.: Categorization of state-of-the-art model persistence and versioning tools

Category		**Tools**
Model Persistence Format	**File-based**	EMF
	Database	Teneo, Morsa, MongoEMF, NeoEMF
Model Repository		ModelCVS, CDO, EMFStore, Modelio MagicDraw, MetaEdit+
Model Index		Hawk

6.4. Future Work

Evaluating this model indexing approach against current state-of-the-art modeling tools and approaches has provided us with sufficient confidence that the research hypothesis is validated, giving incentive for it to be extended in the future.

Firstly, it is worth noting that a model index, even though currently contains a full copy of the model contents found on the relevant version control system, does not have to. In principle, if some contents of the model are not deemed useful they can be omitted in order to gain an improvement in insertion, update and possibly query time (this can be done by creating custom updaters, and using them instead of – or in conjunction with – the default one used to create the representation described in Section 4.3.1). Furthermore, meta information regarding aspects outside the model can be stored in the model index, either formulated in the programing language of the system (such as Java for Hawk), or using the derived attribute functionality seen in Section 4.3.8.1; for example metadata regarding the commits made for each model file (such as author, time-stamp or file size) can be stored alongside information extracted from the models.

A second aspect of this work that can be further investigated is optimizing various components of the system. The VCS managers currently work with base operations such as add, delete, update but could be able to handle complex events such as move in a much more efficient manner (for example in Hawk a file move event would result in the indexed file being removed and re-added whereas a better approach would be to simply update the file path in the system). The metamodel managers currently assume immutable metamodels hence if a new version is needed it has to either be added after removing the old version (alongside the entire contents of indexed files depending on it) or as a new metamodel with a new unique namespace, but a better approach would be to support metamodel evolution and only require re-indexing of dependent models when breaking metamodel changes are discovered.

A third aspect of interest is offering a distributed back-end for the system. Currently, both back-ends work on single-node machines and a better approach would be to offer horizontal scalability (for example in a network cluster or in the cloud) so that the order of magnitude of models the system is capable of efficiently handling is increased. This can be achieved either by adding support for distributed management of the current technologies used, or by investigating the relative effectiveness of a distributed layout of the various technologies reviewed in this work.

A final aspect that can be looked at is further integration of this approach with current state-of-the-art tools and practices. For example a tool like OpenBIM could in principle directly use a model indexer to respond to queries or to efficiently create partial visualizations of subsystems (views of the system). Currently, the integration would have to be through EMF or Epsilon, but other query engines could be added to provide either a more efficient way to connect to specific tools, or additional functionality.

Appendices

A. Details on Hawk Interfaces

This appendix presents the Core interfaces Hawk exposes; implementations can be either connected together to form a running Hawk server programmatically, or through the Hawk user interface (shown in Appendix C, which uses the plugin extension point mechanism provided by the Eclipse IDE, discussed in Section 4.4.1).

IModelIndexer This interface provides the API for the Core component of Hawk. It offers the required methods for setting up a Hawk model indexing server and running any necessary operations on it (startup/maintenance/querying etc.); it acts as the central point of control for the system. Key methods are described below:

- *init()* This method is used to initialize Hawk. It should be called after all the relevant factories have been created and implementations need to create a synchronization schedule (which needs to run in a separate NON-daemon thread, ensuring the JVM stays active) as well as scheduling the initial insertion of any static metamodels of known metamodel factories into Hawk (if they are not already present).

- *shutdown()* This method is used to gracefully terminate Hawk. No other methods should be called after this other than *init()* (in the case Hawk needs to be restarted). Implementations need to offer a serialization strategy for saving Hawk's metadata to the platform used to host Hawk, need to call the shutdown methods of all *VcsManagers* as well as the *IGraphDatabase* Hawk is currently using, and then cancel the running synchronization strategy used by Hawk (in order to allow for a potential termination of the JVM if required).

- *addVCSManager(vcs)* This method is used to add an implementation of an *IVcsManager* to Hawk. This will allow Hawk to monitor such version control systems.

- *addMetaModelResourceFactory(metaModelFactory)* This method is used to add an implementation of an *IMetaModelResourceFactory* to Hawk. This will allow Hawk to be able to parse metamodels of this type.

- *addModelResourceFactory(modelFactory)* This method is used to add an implementation of an *IModelResourceFactory* to Hawk. This will allow Hawk to be able to parse models of this type.

- *setDB(db)* This method is used to set the back-end of Hawk to the selected implementation of an *IGraphDatabase*.

- *addQueryEngine(q)* This method is used to add an implementation of an *IQueryEngine* to Hawk. This allows Hawk to use this engine for querying and creating derived attributes (more information can be found in Section 4.3.8.1).

- *addModelUpdater(updater)* This method is used to add an implementation of an *IModelUpdater* to Hawk. A model updater is responsible for handling the propagation of model changes (from monitored VCSs) to Hawk.

- *setMetaModelUpdater(metaModelUpdater)* This method is used to set the implementation of the *IMetaModelUpdater* in Hawk. The metamodel updater is responsible for inserting metamodels into Hawk.

- *registerMetamodel(f)* This method registers a metamodel originating in a File (*f*), with Hawk. Registering a metamodel comprises using the relevant *IMetamodelResourceFactory* to parse it into an *IHawkMetamodelResource* and then using the *IMetaModelUpdater*(s) to insert it into Hawk.

- *synchronize()* This method is called according to the synchronization schedule of Hawk. It finds out which files of interest to Hawk have changed, uses the relevant *IModelResourceFactories* to parse them into *IHawkModelResource*s and propagates the resulting resources to Hawk's *IModelUpdater*s.

IVcsManager This interface exposes a version control system to Hawk. It is used to find out which files in it have changed between two revisions and subsequently fetch these files for updating them in Hawk. Key methods are described below:

- *getDelta(startRevision)* This method takes an initial revision and returns the files which have changed in the repository since that revision.

- *importFiles(path,temp)* This method imports one or more files from the VCS using *path* as the path to the file(s) and *temp* as the local directory where to import them to. It is worth noting that calling this method should only request files of interest to Hawk, making it the responsibility of the *IModelIndexer* not to request unnecessary files (such as files Hawk cannot parse or unchanged files).

IMetaModelResourceFactory This interface exposes a modeling language (metamodel) persistence format to Hawk. This allows for any metamodel written in that language (and persisted in that form), to be stored in Hawk. Key methods are described below:

- *canParse(f)* This method returns whether a File (*f*) can be parsed by this factory into an *IHawkMetaModelResource*. Implementations should aim to provide a lightweight approach for this method, such as by using the file extension type.

- *parse(f)* This method returns the *IHawkMetaModelResource* created by parsing the File (*f*). More details on *IHawkMetaModelResource*s can be found is Section 4.3.2.1.

- *getStaticMetamodels()* This method returns any static metamodel(s) provided by the modeling language, as *IHawkMetaModelResource*(s). Such metamodels are intended to be automatically added to any Hawk that knows about this *IMetaModelResourceFactory*, upon initialization.

- *parseFromString(String name, String contents)* This method returns a *IHawkMetaModelResource* created from a String containing the entire metamodel content. Certain modeling technologies (such as EMF) only allow models to be parsed into memory when their metamodel is also in memory; this limitation means that during the execution of Hawk, all metamodels of such technologies need to be loaded in memory in their native format. For this requirement to be met, Hawk needs to persist the original metamodel in its original format upon registration, and every time Hawk starts up it needs to provide this persisted metamodel to its relevant factory and hence have it in memory for as long as Hawk runs. This method allows for the factory to receive the persisted metamodel (in the form of a String) and create the relevant resource (which, if required, will be registered to the relevant modeling technology for use). The content String is persisted in Hawk as an attribute found in every metamodel node in the model index, which happens during metamodel insertion.

IModelResourceFactory This interface exposes a model persistence format to Hawk. This allows for any model written in that language (and persisted in that form) to be stored in Hawk. Key methods are described below:

- *canParse(f)* This method returns whether a File (*f*) can be parsed by this factory into an *IHawkModelResource*. Implementations should aim to provide a lightweight approach for this method, such as by using the file extension type.
- *parse(f)* This method returns the *IHawkModelResource* created by parsing the File (*f*). More details on *IHawkModelResource*s can be found is Section 4.3.2.1.
- *getModelExtensions()* This method returns a collection of Strings representing which file extensions this factory is able to parse. This method is used for discarding non-model files and should be as complete as possible with any model extension possibly recognized by the factory. For the actual model parsing into a resource the *canParse(f)* method should be called before any costly parsing begins (if any more checks need to be performed).

IMetaModelUpdater This interface provides Hawk with a strategy for inserting *IHawkMetaModelResource*s (created by *IMetaModelResourceFactories*) into its current back-end *IGraphDatabase*. Key methods are described below:

- *insertMetamodels(metaModelResources,hawk)* This method takes a collection of *IHawkMetaModelResource*s and uses the *IGraphDatabase* API to insert them into Hawk.
- *addDerivedAttribute(...)* This method adds a *derived attribute* to the relevant metamodel already present in Hawk. More details on *derived attribute*s can be found in Section 4.3.8.1.
- *addIndexedAttribute(...)* This method adds an *indexed attribute* to the relevant metamodel already present in Hawk. More details on *indexed attribute*s can be found in Section 4.3.8.3.

IModelUpdater This interface provides Hawk with a strategy for inserting *IHawk-ModelResource*s (created by *IModelResourceFactories*) into its current back-end *IGraph-Database*. Key methods are described below:

- *updateStore(modelResources,hawk)* This method takes a collection of *IHawkModel-Resource*s (alongside the versions of their originating file) and uses the *IGraph-Database* API to update Hawk to reflect the latest version of each one. It returns the actual model-level changes Hawk had to perform to synchronize itself with the changes in these *IHawkModelResource*s (compared to the current version of each one in Hawk).

- *updateDerivedAttribute(...)* This method updates a *derived attribute* to the relevant model already present in Hawk. More details on *derived attribute*s can be found in Section 4.3.8.1 and more information of when such attributes are updated can be found in Section 4.3.8.2.

- *updateIndexedAttribute(...)* This method updates an *indexed attribute* to the relevant model already present in Hawk. More details on *indexed attribute*s can be found in Section 4.3.8.3.

IQueryEngine This interface allows Hawk to connect with engines providing languages that can be used to query it. Such engines are of great importance as they offer the main way information stored in Hawk model indexes can be retrieved. Key methods are described below:

- *contextlessQuery(query,graph)* This method takes a query in the form of a String and returns the results of running this query in Hawk's graph back-end. Implementations will need to either use the *IQueryEngine*'s API to implement a connection with Hawk (through its *IGraphDatabase* layer) or will have to parse the query and convert it to a set of calls to Hawk's *IGraphDatabase* layer directly.

- *contextfullQuery(query,graph,context)* Similar to the above, this method takes a query, this time with a context map which can provide parameters to the *IQuery-Engine* on how it should perform the query or on constraints it needs to place on the results provided back from the query.

B. Model Mutation Operations

This appendix presents the five model mutation operations used to change the source models for the experiments performed in Section 5.3.2.

Listing B.1: EOL model mutation operations

```
//deletes a random TypeDeclaration and keeps track of the total
    number of deleted elements (due to containments being deleted) in
    variable i
operation deleteTypeDeclaration() {
var td = TypeDeclaration.all.random();
var it = td.eAllContents;
while(it.hasnext()){
  it.next();
  i=i+1;
  }
i=i+1;
delete td;
}
//creates a new TypeDeclaration and gives it a random name
operation createTypeDeclaration() {
var t = new TypeDeclaration;
var count = Sequence{0..1000}.random();
var name = new SimpleName;
name.fullyQualifiedName = "synthetic_name_"+count;
t.name = name;
```

```
}
//adds a new MethodDeclaration to a random TypeDeclaration, setting
   its returnType to a randomly named type, and setting one random
   Modifier of the MethodDeclaration
operation addMethodDeclaration() {
var t = TypeDeclaration.all.random();
var m = new MethodDeclaration;
var returnType = new SimpleType;
var count = Sequence{0..1000}.random();
var name = new SimpleName;
name.fullyQualifiedName = "synthetic_name_"+count;
returnType.name = name;
m.returnType = returnType;
var mod = new Modifier;
var choice = Sequence{0..5}.random();
switch(choice){
  case 0 : mod.public = true;
  case 1 : mod.protected = true;
  case 2 : mod.private = true;
  case 3 : mod.static = true;
  case 4 : mod.abstract = true;
  case 5 : mod.final = true;
}
m.modifiers.add(mod);
t.bodyDeclarations.add(m);
}
//changes a random Modifier of a random MethodDeclaration from true
   to false (if at least one exists)
operation changeModifier(){
```

```
var m = MethodDeclaration.all.random();
var mods = m.modifiers;
mods = mods.select(mod:Modifier|mod.public = true or mod.
   protected = true or mod.private = true or mod.static = true
   or mod.abstract = true or mod.final = true);
if(mods.size>0){
  var mod = mods.random();
  mod.public = false; mod.protected = false; mod.private = false
     ; mod.static = false; mod.abstract = false; mod.final =
     false;
  }
}
//renames a random TypeDeclaration to a random name
operation renameTypeDeclaration(){
var t = TypeDeclaration.all.random();
var name = t.name;
var count = Sequence{0..1000}.random();
var n = "synthetic_name_"+count; name.fullyQualifiedName = n;
}
```

C. User Guide

Hawk offers an Eclipse-based user interface for running and maintaining model indexes. Figure C.1 shows how a new Hawk view looks like in a new Eclipse running the requires Hawk plugins (installation details and the plugins themselves can be found online[1]).

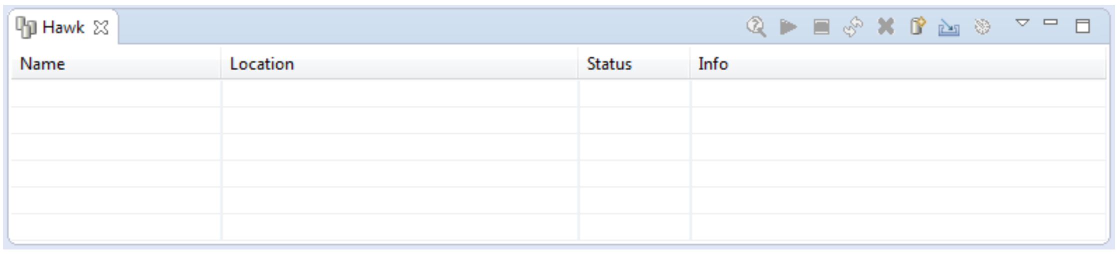
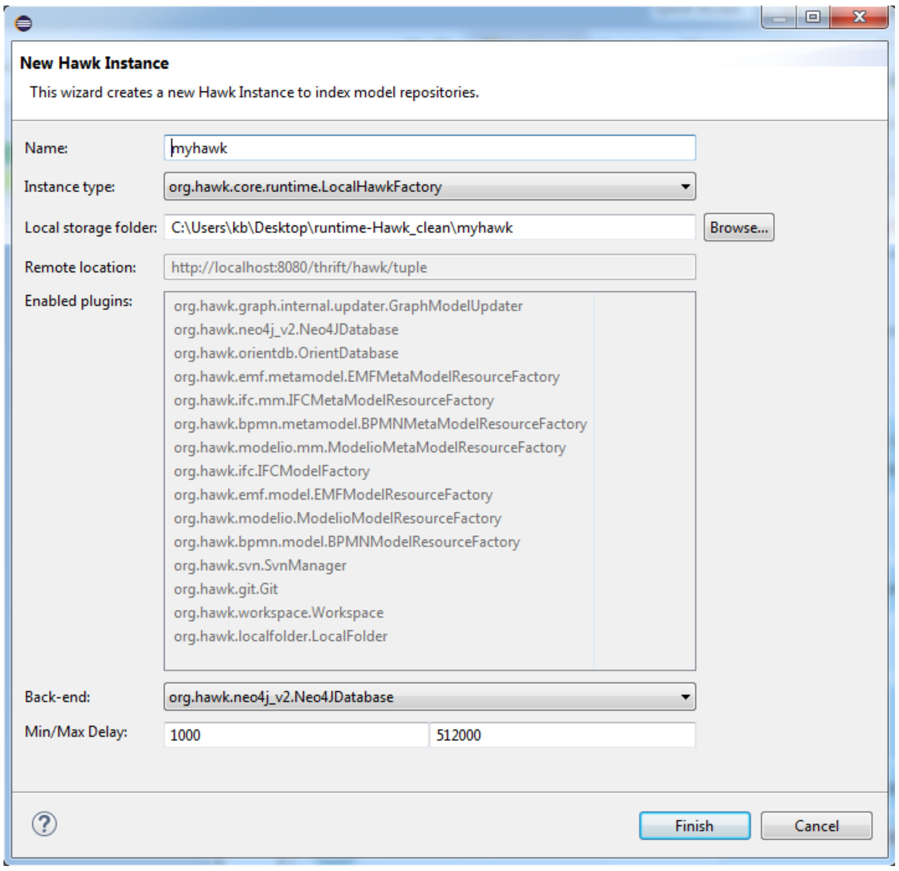


Figure C.1.: Initial Hawk View

A new Hawk instance is added by clicking on the “Add” button found both in the menu when you right click anywhere within the Hawk view and as a button in the button bar on the top right hand side of the view. The window shown in Figure C.2 allows configuring of the new Hawk by choosing its name and location, as well as which back-end store should be used. Advanced options such as setting the periodic check interval of Hawk as well as using remote Hawk instances are also available but are not covered here.

After having a running Hawk instance selected, metamodels can be added to it. Firstly, clicking “Configure” button (which is only enabled if a running Hawk server is currently selected in the view) will open the Hawk configuration dialog. The window in Figure C.3 allows configuring the metamodels in the selected Hawk instance. By clicking the “Add...” button in this window a file chooser is shown, allowing the addition of a (set of) metamodel file(s) to Hawk. In this example the JDTAST.ecore metamodel (used in the evaluation section 5.2.1) is added to Hawk, which will now show in the Metamodels tab, as seen in Figure C.4.

[1] `https://github.com/mondo-project/mondo-hawk`

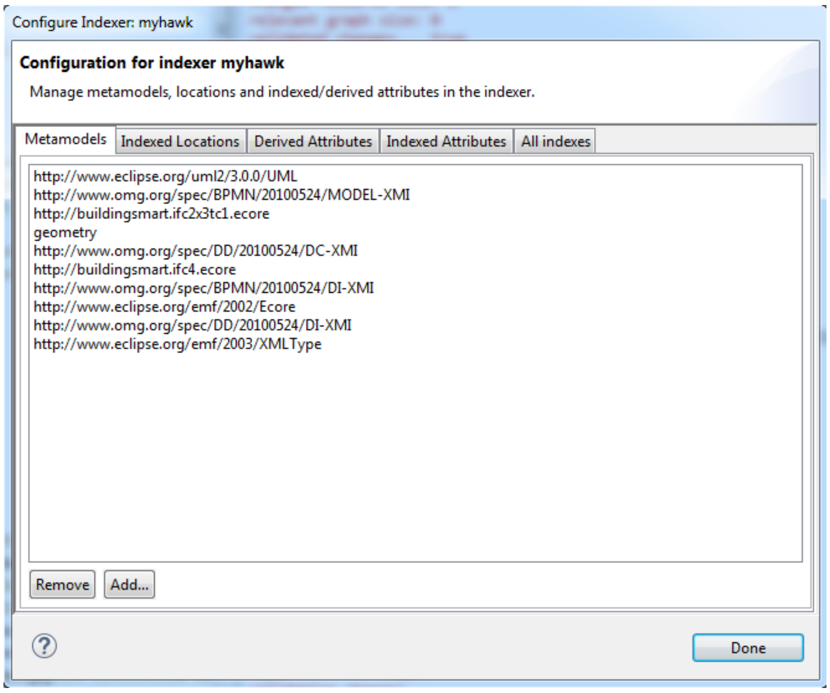

New Hawk Instance
This wizard creates a new Hawk Instance to index model repositories.

Name: myhawk
Instance type: org.hawk.core.runtime.LocalHawkFactory
Local storage folder: C:\Users\kb\Desktop\runtime-Hawk_clean\myhawk
Remote location: http://localhost:8080/thrift/hawk/tuple
Enabled plugins:
org.hawk.graph.internal.updater.GraphModelUpdater
org.hawk.neo4j_v2.Neo4JDatabase
org.hawk.orientdb.OrientDatabase
org.hawk.emf.metamodel.EMFMetaModelResourceFactory
org.hawk.ifc.mm.IFCMetaModelResourceFactory
org.hawk.bpmn.metamodel.BPMNMetaModelResourceFactory
org.hawk.modelio.mm.ModelioMetaModelResourceFactory
org.hawk.ifc.IFCModelFactory
org.hawk.emf.model.EMFModelResourceFactory
org.hawk.modelio.ModelioModelResourceFactory
org.hawk.bpmn.model.BPMNModelResourceFactory
org.hawk.svn.SvnManager
org.hawk.git.Git
org.hawk.workspace.Workspace
org.hawk.localfolder.LocalFolder
Back-end: org.hawk.neo4j_v2.Neo4JDatabase
Min/Max Delay: 1000 512000

Figure C.2.: Creating a new Hawk

Next, Hawk needs to be provided with a path so that it can pick up any models found there to index. In this example, the LocalFolder driver is used for simplicity and the local folder “exampleModels” is indexed as shown in Figure C.5. After clicking the “OK” button in the window the chosen folder is now indexed in Hawk, as in Figure C.6.

Now that a folder is indexed, in this case one named *exampleModels*, Hawk has automatically created the relevant model index, which can now be queried using the EOL query engine driver of Hawk. For running the EOL program called grabats.eol, the usual Epsilon “Run configurations” option in Eclipse can be used (by right-clicking the file), and creating a new EOL Program run configuration, as shown in Figure C.7. This program points to the selected EOL file (in this case a file named *grabats.eol*). By going to the “Models” tab of the configuration, the “Add..” button can be used to add a new Hawk model index to the configuration, as shown in Figure C.8. This opens the configuration window shown in Figure C.9 where model can be named and pointed to

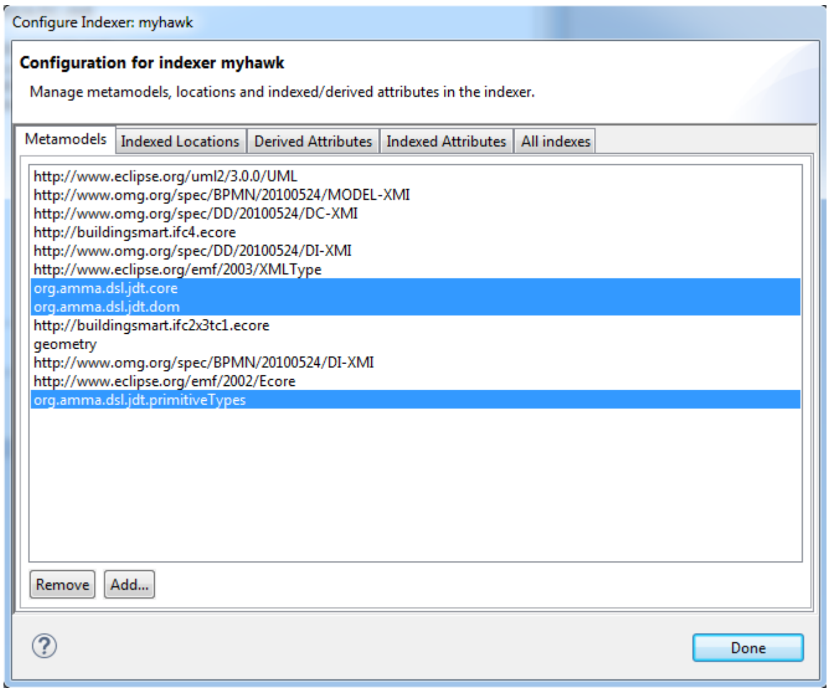


Configuration for indexer myhawk

Manage metamodels, locations and indexed/derived attributes in the indexer.

```
http://www.eclipse.org/uml2/3.0.0/UML
http://www.omg.org/spec/BPMN/20100524/MODEL-XMI
http://buildingsmart.ifc2x3tc1.ecore
geometry
http://www.omg.org/spec/DD/20100524/DC-XMI
http://buildingsmart.ifc4.ecore
http://www.omg.org/spec/BPMN/20100524/DI-XMI
http://www.eclipse.org/emf/2002/Ecore
http://www.omg.org/spec/DD/20100524/DI-XMI
http://www.eclipse.org/emf/2003/XMLType
```

Figure C.3.: Metamodel configuration

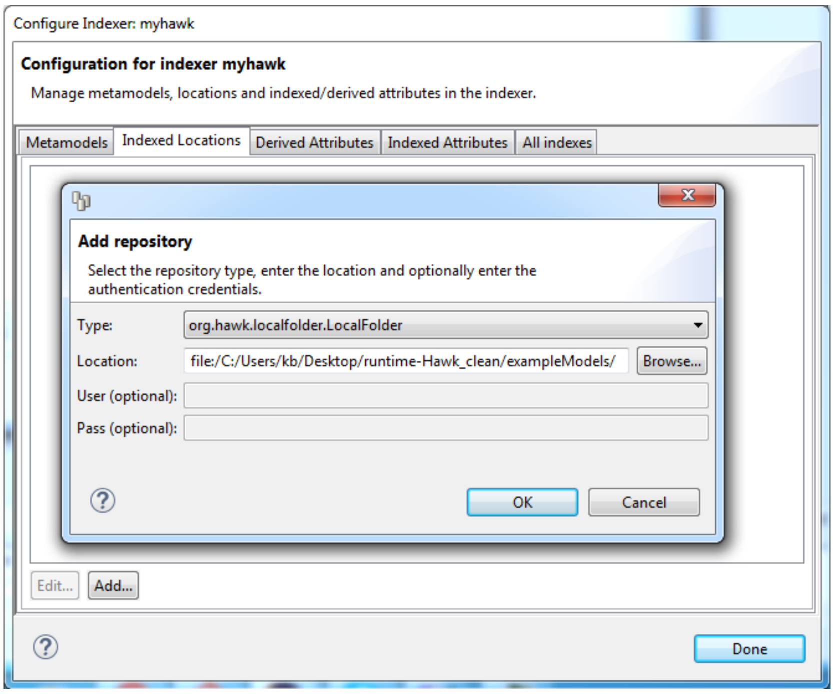


Configuration for indexer myhawk

Manage metamodels, locations and indexed/derived attributes in the indexer.

```
http://www.eclipse.org/uml2/3.0.0/UML
http://www.omg.org/spec/BPMN/20100524/MODEL-XMI
http://www.omg.org/spec/DD/20100524/DC-XMI
http://buildingsmart.ifc4.ecore
http://www.omg.org/spec/DD/20100524/DI-XMI
http://www.eclipse.org/emf/2003/XMLType
org.amma.dsl.jdt.core
org.amma.dsl.jdt.dom
http://buildingsmart.ifc2x3tc1.ecore
geometry
http://www.omg.org/spec/BPMN/20100524/DI-XMI
http://www.eclipse.org/emf/2002/Ecore
org.amma.dsl.jdt.primitiveTypes
```

Figure C.4.: JDTAST Metamodels added to Hawk


Configure Indexer: myhawk

Configuration for indexer myhawk

Manage metamodels, locations and indexed/derived attributes in the indexer.

Add repository

Select the repository type, enter the location and optionally enter the authentication credentials.

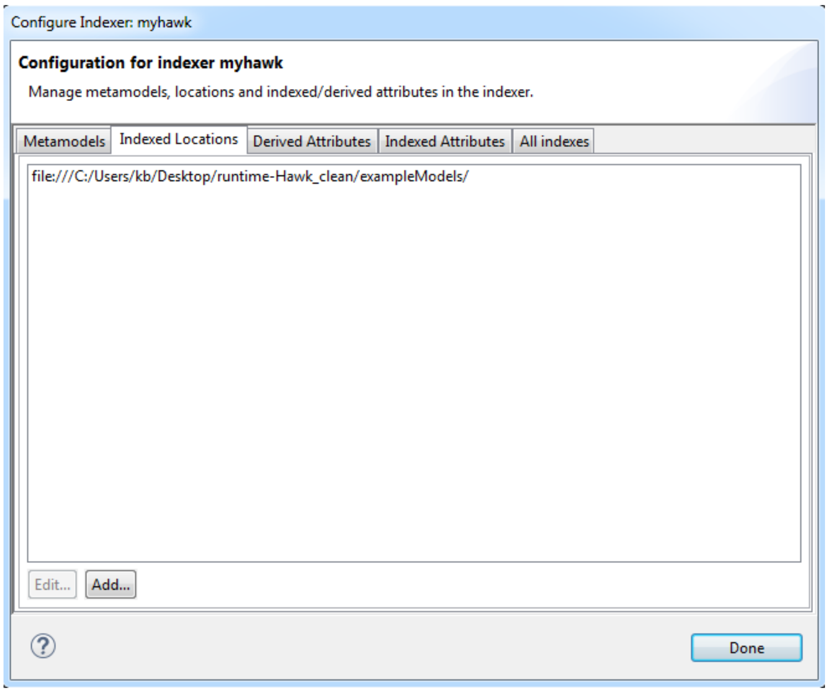
Type: org.hawk.localfolder.LocalFolder

Location: file:/C:/Users/kb/Desktop/runtime-Hawk_clean/exampleModels/

User (optional):

Pass (optional):

Figure C.5.: Indexed folder configuration


Configure Indexer: myhawk

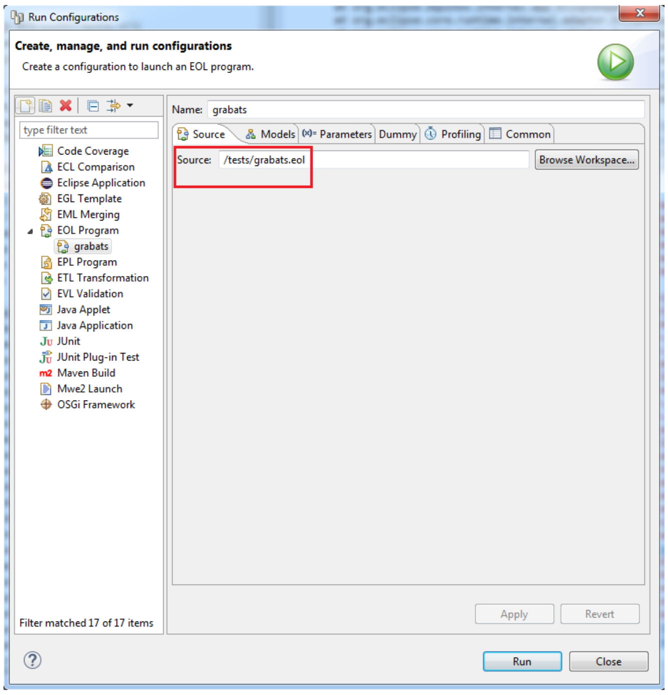
Configuration for indexer myhawk

Manage metamodels, locations and indexed/derived attributes in the indexer.

file:///C:/Users/kb/Desktop/runtime-Hawk_clean/exampleModels/

Figure C.6.: Folder added to Hawk

the specific Hawk model index it needs to be read from. Optionally the query results can be limited to a specific set of files/repositories (as described in Section 4.3.6.2) if desired. Now the query can be run on the Hawk model by pressing the "Run" button, as shown in Figure C.10.



Name: grabats

Source: /tests/grabats.eol

Figure C.7.: Setting the source EOL file

The Hawk model index can be further optimized by adding derived attributes (described in Section 4.3.8.1). This can be done by selecting the "Derived Attributes" tab of the configuration window of Hawk, as shown in Figure C.11. By clicking the "Add.." button in the window the option to create a new derived attribute for Hawk is available, as shown in Figure C.12. After clicking "OK" the new derived attribute that has been added is seen, as shown in Figure C.13.

Finally, (database) indexed attributes can be added, as described in Section 4.3.8.3. This is done by selecting the "Indexed Attributes" tab of the configuration window of Hawk, as shown in Figure C.14. By clicking the "Add.." button in the window the option to create a new indexed attribute for Hawk is available. After clicking "OK" the new indexed attribute that has been added is seen, as shown in Figure C.15.

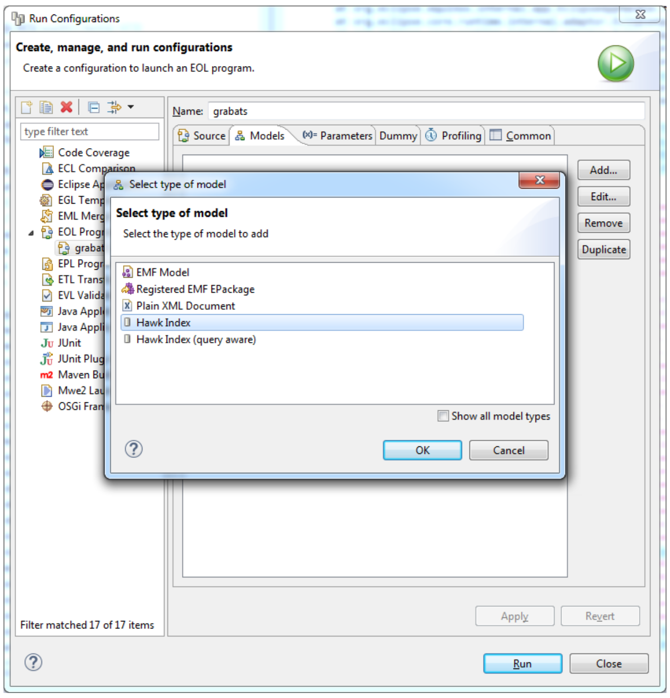
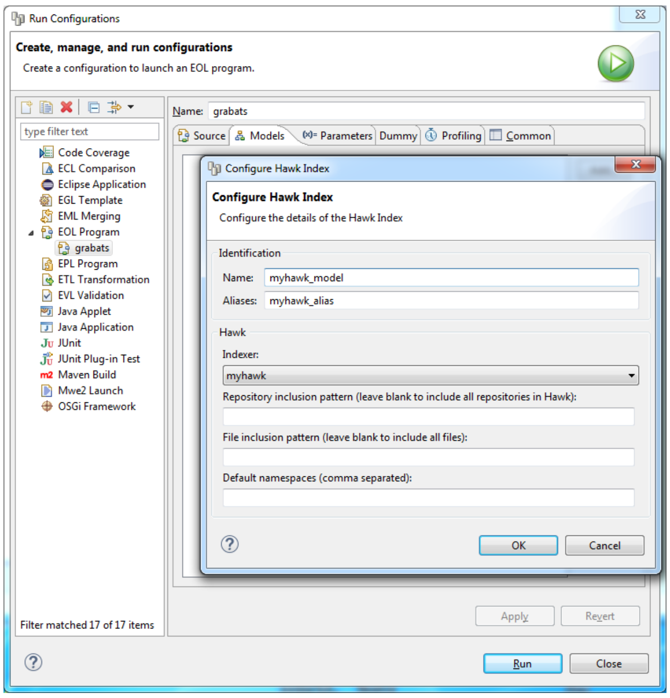


Figure C.8.: Adding a Hawk Index

A complete set of screencasts for running Hawk can be found online[2].

[2] https://www.youtube.com/channel/UCfJydYvvfcEg6o0kaSzC-LQ



Run Configurations

Create, manage, and run configurations

Create a configuration to launch an EOL program.

Name: grabats

Configure Hawk Index

Configure Hawk Index

Configure the details of the Hawk Index

Identification

Name: myhawk_model

Aliases: myhawk_alias

Hawk

Indexer: myhawk

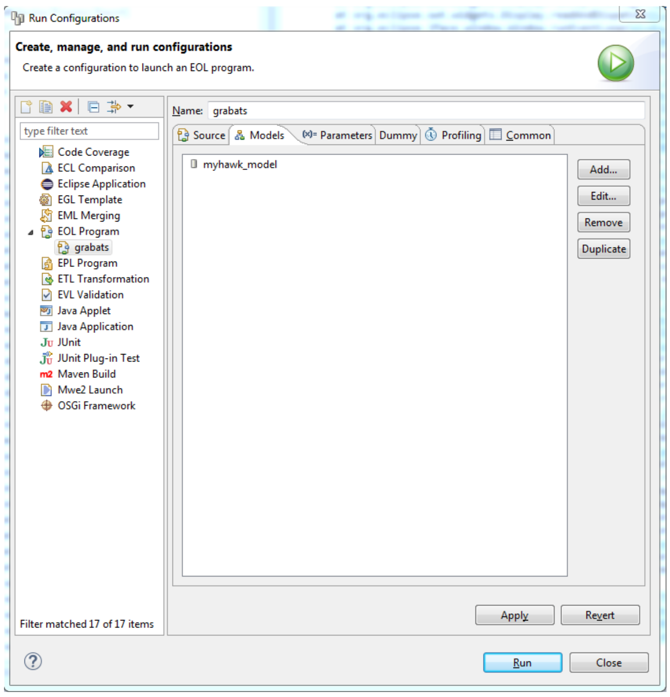
Repository inclusion pattern (leave blank to include all repositories in Hawk):

File inclusion pattern (leave blank to include all files):

Default namespaces (comma separated):

Filter matched 17 of 17 items

Figure C.9.: Configuring the Hawk Index



Run Configurations

Create, manage, and run configurations

Create a configuration to launch an EOL program.

Name: grabats

myhawk_model

Filter matched 17 of 17 items

Figure C.10.: Running the EOL query

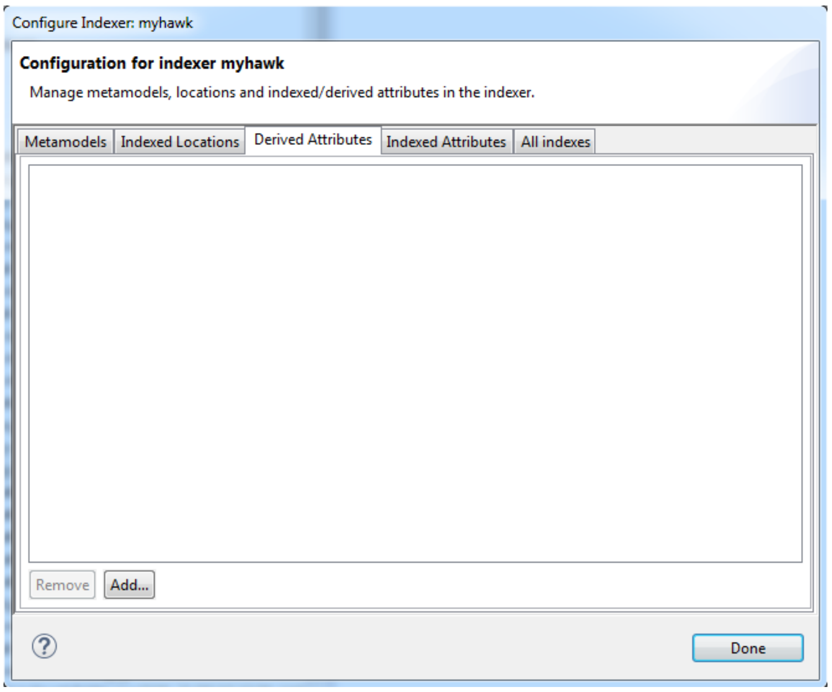
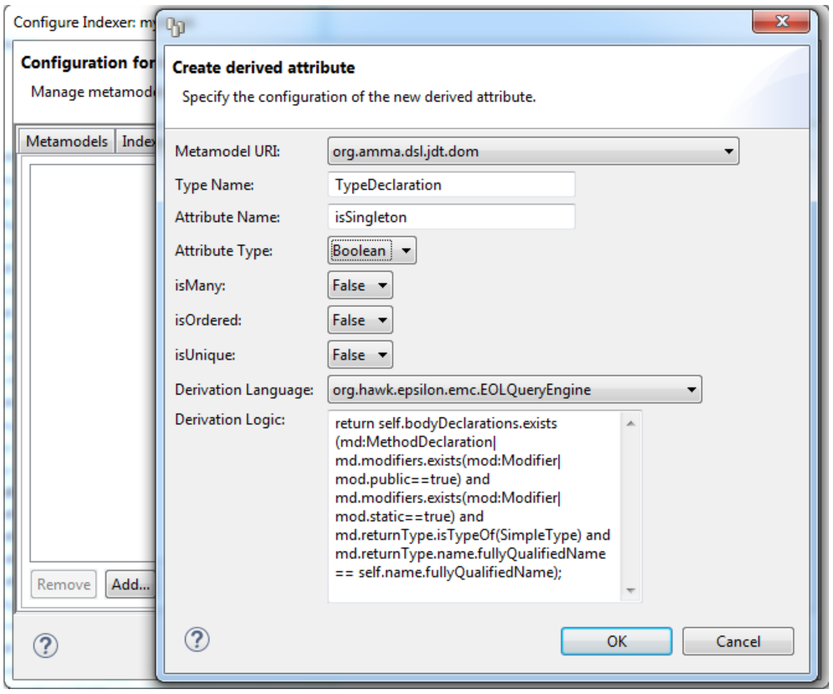

Configure Indexer: myhawk
Configuration for indexer myhawk
Manage metamodels, locations and indexed/derived attributes in the indexer.
Metamodels
Indexed Locations
Derived Attributes
Indexed Attributes
All indexes
Remove
Add...
Done

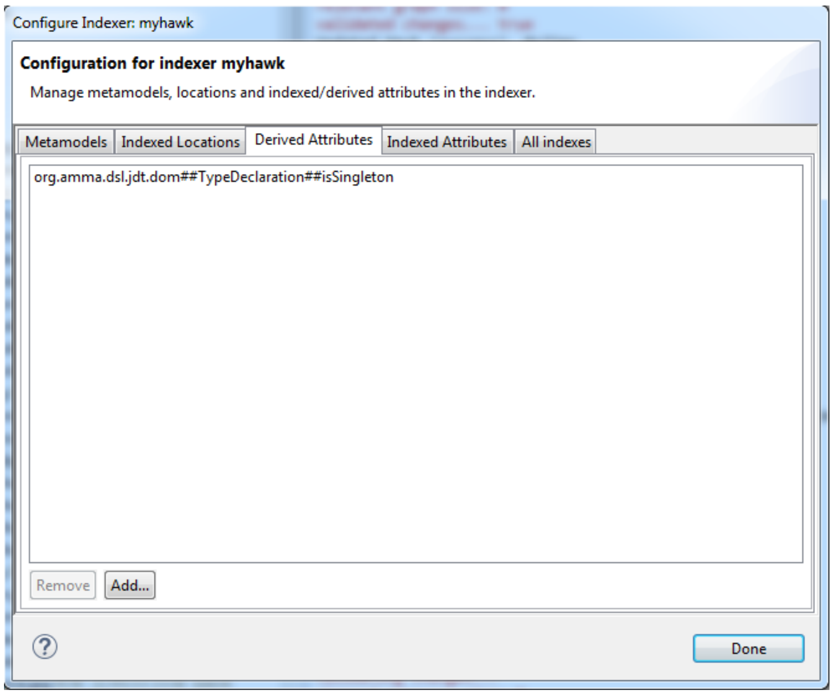
Figure C.11.: Configuring Derived Attributes



Metamodel URI: org.amma.dsl.jdt.dom
Type Name: TypeDeclaration
Attribute Name: isSingleton
Attribute Type: Boolean
isMany: False
isOrdered: False
isUnique: False
Derivation Language: org.hawk.epsilon.emc.EOLQueryEngine
Derivation Logic:

```
return self.bodyDeclarations.exists
(md:MethodDeclaration|
md.modifiers.exists(mod:Modifier|
mod.public==true) and
md.modifiers.exists(mod:Modifier|
mod.static==true) and
md.returnType.isTypeOf(SimpleType) and
md.returnType.name.fullyQualifiedName
== self.name.fullyQualifiedName);
```

Figure C.12.: Adding a new Derived Attribute



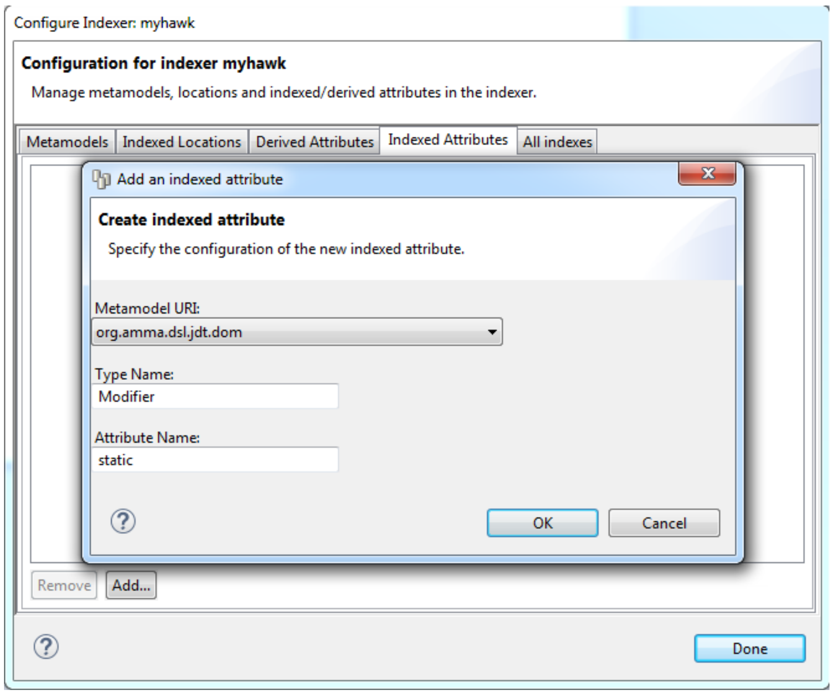
Configure Indexer: myhawk

Configuration for indexer myhawk

Manage metamodels, locations and indexed/derived attributes in the indexer.

org.amma.dsl.jdt.dom##TypeDeclaration##isSingleton

Figure C.13.: Viewing current Derived Attributes



Configure Indexer: myhawk

Configuration for indexer myhawk

Manage metamodels, locations and indexed/derived attributes in the indexer.

Add an indexed attribute

Create indexed attribute

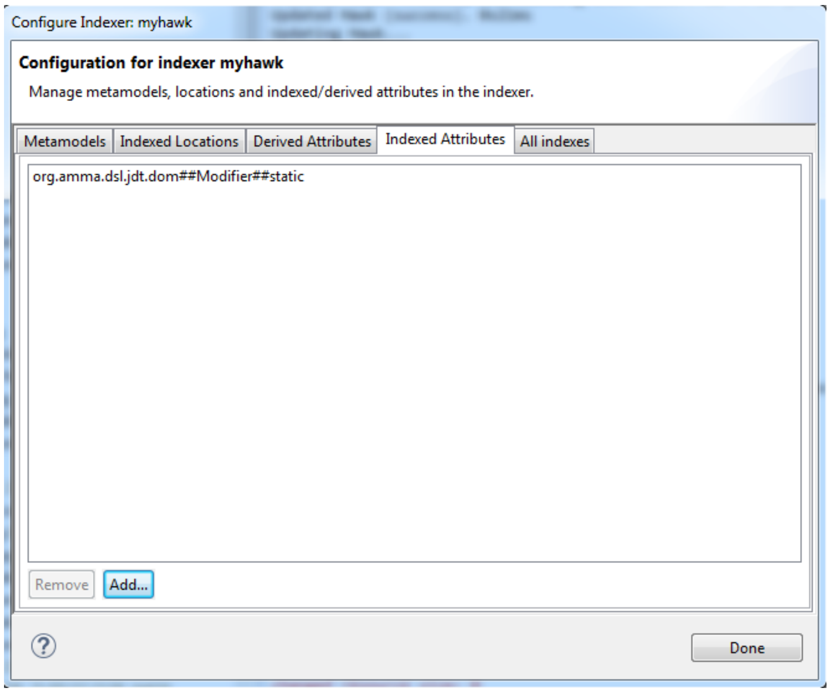
Specify the configuration of the new indexed attribute.

Metamodel URI: org.amma.dsl.jdt.dom

Type Name: Modifier

Attribute Name: static

Figure C.14.: Adding a new Indexed Attribute


Configure Indexer: myhawk

Configuration for indexer myhawk

Manage metamodels, locations and indexed/derived attributes in the indexer.

org.amma.dsl.jdt.dom##Modifier##static

Figure C.15.: Viewing current Indexed Attributes

Bibliography

[1] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The Epsilon Object Language (EOL). In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin / Heidelberg, 2006. 10.1007/11787044_11.

[2] The Unified Modeling Language (UML) [online], 2012. [Accessed 1 June 2012] Available at: `http://www.uml.org/`.

[3] The Business Process Model and Notation (BPMN) [online], 2015. [Accessed 1 September 2015] Available at: `http://www.bpmn.org/`.

[4] The Systems Modeling Language (SysML) [online], 2015. [Accessed 1 September 2015] Available at: `http://sysml.org/`.

[5] Archimate Modeling Language [online], 2015. [Accessed 1 September 2015] Available at: `http://www.opengroup.org/subjectareas/enterprise/archimate`.

[6] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling.* John Wiley & Sons, 2007.

[7] The Web Modeling Language [online], 2012. [Accessed 1 June 2012] Available at: `http://www.webml.org/`.

[8] The Systems Biology Markup Language [online], 2012. [Accessed 1 June 2012] Available at: `http://sbml.org/Main_Page`.

[9] Richard Mark Soley. *Object Management Architecture Guide*, volume OMG Document 92-11-1. Object Management Group, 1992.

[10] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[11] OMG. Meta-Object Facility (MOF) Specification Version 2.4.1 [online], 2012. [Accessed 1 June 2012] Available at: `http://www.omg.org/spec/MOF/2.4.1/PDF`.

[12] Dick Grune and Ceriel J. H. Jacobs. *Parsing techniques: a practical guide*. Springer New York, 2008.

[13] Heiko Kern. The interchange of (meta) models between metaedit+ and eclipse emf using m3-level-based bridges. In *8th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA*, volume 2008, 2008.

[14] Marcelo Paternostro Dave Steinberg Frank Budinsky and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2008.

[15] Jordi Cabot and Martin Gogolla. Object constraint language (ocl): a definitive guide. In *Formal Methods for Model-Driven Engineering*, pages 58–90. Springer, 2012.

[16] Kolovos, D.S., Paige, R.F. and Polack, F.A. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.

[17] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152(0):125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[18] Sreedhar Reddy, R Venkatesh, and Zahid Ansari. A relational approach to model transformation using QVT Relations. *iistunuedu*, 2008.

[19] Radomil Dvorak. Model transformation with operational QVT. In *EclipseCon 08, Santa Clara*, 2008.

[20] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer*

Science, pages 46–60. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-69927-9_4.

[21] Frédéric Jouault 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. 10.1007/11663430_14.

[22] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 719–720, New York, NY, USA, 2006. ACM.

[23] The Epsilon Generation Language [online], 2012. [Accessed 1 June 2012] Available at: `http://www.eclipse.org/epsilon/doc/egl/`.

[24] Xpand - M2T transformation language [online], 2012. [Accessed 1 June 2012] Available at: `http://www.eclipse.org/projects/project.php?id=modeling.m2t.xpand`.

[25] Acceleo - Code generation language [online], 2012. [Accessed 1 June 2012] Available at: `http://www.acceleo.org/pages/welcome/en`.

[26] Java Emitter Template (JET) [online], 2012. [Accessed 1 June 2012] Available at: `https://www.eclipse.org/modeling/m2t/?project=jet`.

[27] Parastoo Mohagheghi, Miguel Fernandez, Juan Martell, Mathias Fritzsche, and Wasif Gilani. MDE Adoption in Industry: Challenges and Success Criteria. In *Models in Software Engineering*, volume 5421 of *Lecture Notes in Computer Science*, pages 54–59. Springer, 2009.

[28] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Scalability: The Holy Grail of Model Driven Engineering. In *Proc. Workshop on Challenges in MDE, collocated with MoDELS '08, Toulouse, France*, 2008.

[29] Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michèle Soria. Uniform Random Generation of Huge Metamodel Instances. In *Proceedings of ECMDA-FA '09*, pages 130–145, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The Epsilon Transformation Language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-69927-9_4.

[31] 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.

[32] The Teneo/Hibernate Relational Database Model Store [online], 2012. [Accessed 1 June 2012] Available at: `http://wiki.eclipse.org/Teneo/Hibernate`.

[33] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. A repository for scalable model management. *Software & Systems Modeling*, pages 1–21, 2013.

[34] Konstantinos Barmpis and Dimitrios S. Kolovos. Evaluation of contemporary graph databases for efficient persistence of large-scale models. *Journal of Object Technology*, 13-3:3:1–26, July 2014.

[35] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. Morsa: a scalable approach for persisting and accessing large models. In *Proceedings of MODELS'11*, pages 77–92, Berlin, Heidelberg, 2011. Springer-Verlag.

[36] MongoDB Developers. MongoDB, Document-Store NoSQL Database [online], 2012. [Accessed 1 June 2012] Available at: `www.mongodb.org/`.

[37] Apache CouchDB Document Store Database [online], 2012. [Accessed 1 June 2012] Available at: `http://couchdb.apache.org/`.

[38] OrientDB Developers. OrientDB, Hybrid Document-Store and Graph NoSQL Database [online], 2012. [Accessed 1 June 2012] Available at: `http://www.orientechnologies.com/`.

[39] Neo4J Developers. Neo4J, Graph NoSQL Database [online], 2012. [Accessed 1 June 2012] Available at: `http://neo4j.org/`.

[40] InfiniteGraph Graph Database [online], 2012. [Accessed 1 June 2012] Available at: `http://objectivity.com/products/infinitegraph/overview`.

[41] Seyyed M. Shah, Ran Wei, Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Konstantinos Barmpis. A framework to benchmark nosql data stores for large-scale model persistence. In *Proc. 15th Conf. on Model-Driven Engineering Lang. and Systems, Models'14*, 2014.

[42] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4emf, a scalable persistence layer for emf models. In Jordi Cabot and Julia Rubin, editors, *Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer International Publishing, 2014.

[43] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proc. 21st ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, 2007.

[44] Abel Gmez, Massimo Tisi, Gerson Suny, and Jordi Cabot. Map-based transparent persistence for very large models. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering*, volume 9033 of *Lecture Notes in Computer Science*, pages 19–34. Springer Berlin Heidelberg, 2015.

[45] Apache Subversion [online], 2012. [Accessed 1 June 2012] Available at: `http://subversion.apache.org/`.

[46] TWiki - the Open Source Enterprise Wiki and Web 2.0 Application Platform [online], 2012. [Accessed 1 June 2012] Available at: `http://twiki.org/`.

[47] PVCS Version Manager [online], 2012. [Accessed 1 June 2012] Available at: `http://pvcs.synergex.com/products/pvcs_version_manager.aspx`.

[48] IBM Rational ClearCase [online], 2012. [Accessed 1 June 2012] Available at: `http://www-01.ibm.com/software/awdtools/clearcase/`.

[49] Brian Berliner and Jeff Polk. Concurrent Versions System (CVS) [online], 2012. [Accessed 1 June 2012] Available at: `www.cvshome.org`.

[50] Git - distributed version control system [online], 2012. [Accessed 1 June 2012] Available at: `http://git-scm.com/`.

[51] Mercurial [online], 2012. [Accessed 1 June 2012] Available at: `http://mercurial.selenic.com/`.

[52] Bazaar [online], 2012. [Accessed 1 June 2012] Available at: `http://bazaar.canonical.com/en/`.

[53] Peer-to-peer Version Control for Distributed Development [online], 2012. [Accessed 1 June 2012] Available at: `http://www.relisoft.com/co_op/index.htm`.

[54] Ben Collins-Sussman, Brian Fitzpatrick, and Michael Pilato. *Version control with subversion.* " O'Reilly Media, Inc.", 2004.

[55] Haifeng Shen and Chengzheng Sun. Operation-based revision control systems. In *Proceedings of the 3rd Annual International Workshop on Collaborative Editing Systems in conjunction with ACM Group Conference*, 2001.

[56] Alessandro Rossini, Adrian Rutle, Yngve Lamo, and Uwe Wolter. A formalisation of the copy-modify-merge approach to version control in MDE. *Journal of Logic and Algebraic Programming*, 79(7):636 – 658, 2010. The 20th Nordic Workshop on Programming Theory (NWPT 2008).

[57] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.

[58] Presentation of EMF Compare Utility [online], EclipseCon 2006, 2006. [Accessed 1 January 2016] Available at: `https://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium10_EMFCompareUtility.pdf`.

[59] Markus Scheidgen and Anatolij Zubow. Map/reduce on emf models. In *Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and CLoud computing*, MDHPCL '12, pages 7:1–7:5, New York, NY, USA, 2012. ACM.

[60] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Towards a semantic infrastructure supporting model-based tool integration. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, GaMMa '06, pages 43–46, New York, NY, USA, 2006. ACM.

[61] The Connected Data Objects model Repository [online], 2012. [Accessed 1 June 2012] Available at: `http://wiki.eclipse.org/CDO`.

[62] Maximilian Koegel and Jonas Helming. Emfstore: a model repository for emf models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 307–308. ACM, 2010.

[63] Javier Espinazo Pagán and Jesús García Molina. Querying large models efficiently. *Inf. Softw. Technol.*, 56(6):586–622, June 2014.

[64] Heiko Kern. The interchange of (meta) models between metaedit+ and eclipse emf using m3-level-based bridges. In *8th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA*, volume 2008, 2008.

[65] Konstantinos Barmpis and Dimitrios S. 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, June 2013. ACM.

[66] Konstantinos Barmpis, Seyyed Shah, and Dimitrios S. Kolovos. Towards incremental updates in large-scale model indexes. In *Proceedings of the 11th European Conference on Modelling Foundations and Applications. ECMFA'15*, July 2015.

[67] Konstantinos Barmpis and Dimitrios S. Kolovos. Towards scalable querying of large-scale models. In *Proceedings of the 10th European Conference on Modelling Foundations and Applications. ECMFA'14*, July 2014.

[68] Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N. and Polack, F.A. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Proc. 14th IEEE International Conf. on Engineering of Complex Computer Systems*, Potsdam, Germany, 2009.

[69] Willink, E. Aligning OCL with UML. In *Proceedings of the Workshop on OCL and Textual Modelling*, Electronic Communications of the EASST, 2011.

[70] Kolovos, D.S., Rose, L., Garcia, A.D. and Paige, R.F. *The Epsilon Book.* 2008.

[71] Dimitrios S Kolovos, Ran Wei, and Konstantinos Barmpis. An approach for efficient querying of large relational datasets with ocl-based languages. In *XM 2013–Extreme Modeling Workshop*, page 48, 2013.

[72] Alexander Egyed. Instant consistency checking for the uml. In *Proc. of the 28th International Conference on Software Engineering*, ICSE '06, pages 381–390, New York, NY, USA, 2006. ACM.

[73] Alexander Egyed. Automatically detecting and tracking inconsistencies in software design models. *Software Engineering, IEEE Transactions on*, 37(2):188–204, 2011.

[74] Hugo Brunelire, Jordi Cabot, Grgoire Dup, and Frdric Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012 – 1032, 2014.

[75] GraBaTs. 5th Int. Workshop on Graph-Based Tools, 2009.

[76] Jean-Sebastien Sottet and Frédéric Jouault. Program comprehension. In *Proc. 5th Int. Workshop on Graph-Based Tools*, 2009.

[77] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental evaluation of model queries over emf models. In DorinaC. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin Heidelberg, 2010.

[78] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental pattern matching in the viatra model transformation system. In *Proceedings of the Third International Workshop on Graph and Model Transformations*, GRaMoT '08, pages 25–32, New York, NY, USA, 2008. ACM.

[79] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. Emf-incquery: An integrated development environment for live model queries. *Science of Computer Programming*, 98, Part 1:80 – 99, 2015. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).

[80] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. Incquery-d: A distributed incremental model query framework in the cloud. In *Model-Driven Engineering Languages and Systems*, pages 653–669. Springer, 2014.

[81] Gábor Szárnyas, Oszkár Semeráth, István Ráth, and Dániel Varró. The ttc 2015 train benchmark case for incremental model validation. *8th Transformation Tool Contest (TTC 2015)*, 2015.