Every effort has been made to ensure that all statements and information contained herein are accurate, however the Project Partners accept no liability for any error or omission in the same.

© 2015 Copyright in this document remains vested in the MONDO Project Partners.
## Project Partner Contact Information

<table>
<thead>
<tr>
<th>Project Partner</th>
<th>Contact Information</th>
</tr>
</thead>
</table>
| **ARMINES**     | Massimo Tisi  
Rue Alfred Kastler 4  
44070 Nantes Cedex, France  
Tel: +33 2 51 85 82 09  
E-mail: massimo.tisi@mines-nantes.fr |
| **Autonomous University of Madrid** | Juan de Lara  
Calle Einstein 3  
28049 Madrid, Spain  
Tel: +34 91 497 22 77  
E-mail: juan.delara@uam.es |
| **BME**         | Daniel Varro  
Magyar Tudosok korutja 2  
1117 Budapest, Hungary  
Tel: +36 146 33598  
E-mail: varro@mit.bme.hu |
| **IKERLAN**     | Salvador Trujillo  
Paseo J.M. Arizmendiarrieta 2  
20500 Mondragon, Spain  
Tel: +34 943 712 400  
E-mail: strujillo@ikerlan.es |
| **Soft-Maint**  | Vincent Hanniet  
Rue du Chateau de L’Eraudiere 4  
44300 Nantes, France  
Tel: +33 149 931 345  
E-mail: vhanniet@sodifrance.fr |
| **SOFTEAM**     | Alessandra Bagnato  
Avenue Victor Hugo 21  
75016 Paris, France  
Tel: +33 1 30 12 16 60  
E-mail: alessandra.bagnato@softeam.fr |
| **The Open Group** | Scott Hansen  
Avenue du Parc de Woluwe 56  
1160 Brussels, Belgium  
Tel: +32 2 675 1136  
E-mail: s.hansen@opengroup.org |
| **UNINOVA**     | Pedro Maló  
Campus da FCT/UNL, Monte de Caparica  
2829-516 Caparica, Portugal  
Tel: +351 212 947883  
E-mail: pmm@uninova.pt |
| **University of York** | Dimitris Kolovos  
Deramore Lane  
York YO10 5GH, United Kingdom  
Tel: +44 1904 32516  
E-mail: dimitris.kolovos@york.ac.uk |
Contents

1 Introduction 3
  1.1 Purpose of the deliverable 3
  1.2 Structure of the document 4
  1.3 Contributors 4

2 Analysis and architecture 4
  2.1 Model Indexing 4
  2.2 Hawk 5
    2.2.1 Index structure 5
    2.2.2 Component-based architecture 7

3 Design 10
  3.1 Back-end Persistence 10
    3.1.1 Background: Neo4j 10
    3.1.2 Background: OrientDB 11
  3.2 Hawk Mapping Layers 12
    3.2.1 Model Layer 12
    3.2.2 Graph Layer 13
  3.3 Version Control Managers 15
    3.3.1 LocalFolder Manager 16
    3.3.2 Workspace Manager 16
    3.3.3 SVN Manager 16
  3.4 Metamodel/Model Resource Factories 17
    3.4.1 EMF Resource Factories 17
    3.4.2 Other Resource Factories 18
  3.5 Metamodel/Model Updaters 18
    3.5.1 Metamodel Updater 18
    3.5.2 Model Updater 19
  3.6 Querying Hawk 25
    3.6.1 Native Querying 25
    3.6.2 Back-end Independent Navigation and Querying 26
  3.7 System Lifecycle 26
    3.7.1 Synchronization procedure 27
Document Control

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>First complete draft for internal review</td>
<td>2 December 2015</td>
</tr>
<tr>
<td>1.0</td>
<td>Version for submission</td>
<td>21 December 2015</td>
</tr>
</tbody>
</table>
Executive Summary

The trend showing the increase of MDE adoption in industry in the past decade continues, as MDE provides several benefits compared to traditional software engineering practices. These benefits, such as improved productivity and reuse, allow for systems to be built faster and cheaper. Nevertheless, there are limitations in MDE, like scalability, that prevent it from being more widely used. Scalability issues mainly arise when large models are used in MDE processes.

This deliverable presents an integrated description of the final version of the Hawk model indexing framework. Hawk enables the collaborative development of MDE artefacts by monitoring the contents of remote version control repositories and indexing their models and model fragments in a scalable manner. The indexed models can then be queried in a much more efficient and convenient manner, following the approach described in deliverable D5.4 [18]. Hawk has been integrated in the MONDO platform described in deliverable D6.3 [20], which uses the Thrift-based model encoding presented in deliverable D5.6 [19].

The work presented in this document is done in the context of work package (WP) 5, and in particular it is related to Task 5.3:

This task will develop a framework for monitoring changes to models and model fragments, parsing them into a consistent representation using the framework developed in Task 5.2, and indexing them using suitable facilities which will enable them to be queried in an efficient way. This indexing framework will be used both in the developer workbench, to monitor the models in the user’s workspace and in a standalone mode to monitor the models contained in heterogeneous remote repositories (SVN, CVS, FTP, shared network drives), with a focus on efficiency, security and access control.

This report starts with a motivation for scalable model indexing, and presents a high-level view of the requirements imposed on Hawk, the structure of a Hawk model index and the architecture that integrates its various components. The design that implements the architecture is then described in further detail, describing the various classes of components that integrate into the core of Hawk: storage back-ends that represent models as graph databases, parsers that allow Hawk to index models implemented in various technologies, connectors that integrate Hawk with various version control systems, updaters that build the graph from the models in a backend-agnostic way and query languages that provide concise and efficient ways to extract information from the indexed models. These components are integrated into a lifecycle that covers addition, update and removal of model repositories and files. Several optimizations for improved performance and convenience are then outlined, with the most prominent being incrementally updated indexed and derived attributes. Finally, the tooling for Hawk is shown, with a plugin for internal validation of update operations and a user guide for the Eclipse-based user interface.

The correctness and performance of Hawk has been evaluated in several scenarios. Two benchmarks have been used: the SharenGo Java Legacy Reverse-Engineering use case from the Grabats 2009 contest, and the interoperability repository maintained by the OMG BPMN Model Interchange Working Group. When backed by Neo4j, Hawk quickly outperformed the performance of XMI for repeated queries in any model size, and when backed by OrientDB, Hawk outperformed XMI for large models.
with millions of elements. Hawk required about 2.9 times more disk space than XMI when backed by Neo4j, and 4.3 times when backed by OrientDB. When updating models instead of inserting them from scratch, the required times have been shown to scale linearly with the magnitude of the change. Another study on derived and indexed attributes has shown that they can speed up some queries between 72.5% and 89.5%. Finally, a study on continuous index updates based on 25 commits from the BPMN MIWG repository with over 250 files changed in each showed that Hawk could process repeated changes efficiently, taking 6.2 minutes in total when backed by Neo4j and 4.2 minutes when backed by OrientDB.
1 Introduction

The trend showing the increase of MDE adoption in industry in the past decade continues, as MDE provides several benefits compared to traditional software engineering practices, such as improved productivity and reuse, which allow for systems to be built faster and cheaper, some of the key factors in industry [16]. Nevertheless, the limitations of MDE, like scalability, which prevent its even wider use [13] still need to be overcome. Scalability issues mainly arise when large models are used in MDE processes.

Collaborative development of large MDE artefacts is one of the main challenges of contemporary MDE. In a typical collaborative development environment, artefacts are stored in a repository such as a Version Control System (VCS - e.g. CVS, SVN, Git etc.), an FTP server or a shared network folder, and synchronised over the network. In such a scenario even files of the order of tens of megabytes are challenging to manage as for every change they need to be transferred back and forth between the local copy and the remote repository. Therefore, collaborative development of large MDE artefacts in this fashion can be problematic.

One approach to overcome this issue is instead of storing a modelling artefact into one monolithic file, splitting it into multiple, smaller, cross-referenced physical files (model fragments), which can be easily managed by a repository system, using tools like EMF-Splitter [9]. However, this approach has its own limitations. The main limitation is that using this approach with current state-of-the-art technologies makes it impossible to compute queries of global nature on the models such as “find all classes that are sub-classes of X” without going through all the model fragments from the remote repository every time. To address this limitation, in MONDO we have developed a model indexing framework that can monitor the contents of remote version control repositories, and index the models and model fragments they contain in a scalable manner.

1.1 Purpose of the deliverable

The purpose of this document is to report on the implementation of the model indexing framework, which provides facilities for monitoring changes to models and model fragments in local and remote repositories and indexing them in a scalable database which enables the evaluation of global queries in an efficient way.

The work presented in this document is done in the context of work package (WP) 5. The results presented in the following sections are related to the following tasks (from the MONDO DoW):

Task 5.3: Scalable Model Indexing - This task will develop a framework for monitoring changes to models and model fragments, parsing them into a consistent representation using the framework developed in Task 5.2, and indexing them using suitable facilities which will enable them to be queried in an efficient way. This indexing framework will be used both in the developer workbench, to monitor the models in the user’s workspace and in a standalone mode to monitor the models contained in heterogeneous remote repositories (SVN, CVS, FTP, shared network drives), with a focus on efficiency, security and access control.
1.2 Structure of the document

The rest of the document is structured as follows. In Section 2, we present an overview of the architecture and main features of the Hawk indexing framework. In Sections 3 and 4, we provide an integrated description of the final version of the Hawk model indexer, while in Section 5, we present the results of the evaluation we have conducted in order to assess the correctness and performance of Hawk. Finally, we conclude this report in Section 6.

1.3 Contributors

The main contributor of this deliverable is the University of York. All project partners contributed to this document by providing system requirements, bug reports, as well as by providing feedback and suggestions for editing and refinements of this document.

2 Analysis and architecture

In this section we provide an overview of the model indexing framework in order to provide the necessary context for the rest of the deliverable. For a detailed discussion of how Hawk supports multiple model formats and integrates with other model management languages, readers are referred to deliverable D5.4 [18].

2.1 Model Indexing

In a collaborative environment, models need to be version-controlled and shared among many developers. The default approach for doing this is to use a file-based version control system such as Git or SVN. This has certain advantages as such version control systems are robust, widely-used and orthogonal to modelling 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. This can be particularly inefficient for global queries (e.g. is there a UML model in my repository that contains a class named “Customer”? ) that need to be executed on a large number of models. 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, several open-source and proprietary model-specific version control systems such as CDO [5], EMFStore [6] and MagicDraw’s TeamServer [15] have been developed over the last decade. While such systems address some of the limitations above, they require tight coupling with modelling tools, they impose an administration overhead, and they lack the maturity, robustness and wide adoption of file-based version-control systems.

In what can be seen as a happy medium between the two approaches to model version control, model indexing 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 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. Previous work [2, 3] has demonstrated promising results with regards to query execution times, with up to 95.1% decrease in execution time for querying model indexes [2], compared to direct querying of their constituent EMF XMI-based models, and up to a further 71.7% decrease in execution time, when derived (cached) attributes were used [3]. This motivates us to improve upon this technology by improving the efficiency of handling model evolution in such model indexes.

2.2 Hawk

Hawk is a model indexing system that can work with diverse file-based version control systems (VCS) and model persistence formats whilst providing a comprehensive API through which modeling and model management tools can issue queries. Hawk has to be scalable so that it can accommodate large sets of models. Hawk must also be able to access model repositories without requiring any special configuration on their part.

The system architecture of a Hawk index is shown in Figure 1. Hawk produces a graph database from a collection of models stored in one or more version control systems[1] and then monitors the VCS for changes. When a change is detected in one of the models, these changes are propagated to the graph. Developers can then use the graph to perform their queries faster than if they had to download and parse the entire model.

2.2.1 Index structure

Based on results obtained through extensive benchmarking in Task 5.1, published in [2, 22], we have decided to use a NoSQL graph database (Neo4j[2]) for persisting model indexes. An example of such an index, containing a simple library metamodel and a model that conforms to it, is illustrated in Figure 2. In general, a model index typically contains the following entities:

- **File nodes.** These represent files in a repository and contain information on the file such as the path, current revision and type. They are linked with relationships to the *Elements* they contain.
- **Metamodel nodes.** These represent metamodels and contain their names and their unique namespace URIs. They are linked with relationships to the (metamodel) *Types* they contain.
- **Type nodes.** These represent metamodel types (*EClasses* in EMF[3] terminology) and contain their name. They are linked with relationships to their (model) *Element* instances.
- **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.
- **Indexes.** Metamodel nodes and File nodes are indexed in the store, so that their nodes can be efficiently accessed for querying. The metamodel index is used as a starting point for type-based queries (e.g. all instances of a type), and the file index is used as a starting point for file-based queries (e.g. the entire contents of a file or a set of files).

---

1Local folders and Eclipse workspaces are viewed as version control systems by Hawk.
2Neo4j - [http://neo4j.org/](http://neo4j.org/)
The File index uses keys that combine the repository URL and the path of the file within the repository: in order to retrieve all the files within a repository, Hawk queries the index for all the keys that start with the repository URL.
String representations are used to store the notion of model file “version”, abstracting over any details of how versions are identified: for instance, versions might be encoded as integers (as in SVN) or as SHA-1 checksums (as in Git). The implementation layer is responsible for properly interpreting the contents of the string. In erroneous scenarios the version can become de-synchronised when an unparsable new file is found. In this case, Hawk rolls-back back to the latest correct version before the update. Hawk will assume the error is transient and attempt to reach a consistent state by attempting to reinsert the file periodically. The version of each indexed file will always be consistent after such a rollback occurs.

2.2.2 Component-based architecture

Figure 3 illustrates the components that comprise Hawk. Figure 4 shows the related API provided by Hawk for these components. These interfaces are discussed in detail in Section 3, with their importance being explained.

- **Model parser components**: these components provide parsers for specific model persistence formats, such as Ecore models persisted in XMI or UML models persisted in the custom Modelio project format. These parsers take as input the contents of a file stored in version control systems and produce as output a Hawk resource (*IHawkModelResource*).
• **Model indexer components**: these components, specific for each back-end used (such as Neo4J [24] or OrientDB NoSQL databases), receive a model resource and a file revision number/id and insert the former 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 the embedded indexes commonly found in NoSQL stores, such as the Apache Lucene Index). A detailed discussion on this procedure and a case-study using Neo4J and OrientDB, can be found in [1], while a performance evaluation of data stores for model persistence can be found in [22]. Both papers are briefly discussed below.

In [22] we presented a framework and methodology for benchmarking NoSQL datastores in the context of large-scale modelling applications. The framework builds on the Blueprints property graph model interface and provides a layered architecture to avoid repetition and to allow future support of non-EMF modelling technologies. The framework advocates the selection of candidate stores via a multi-stage process, which involves drawing from the literature and existing benchmarks. In [1], we explored the use of graph-based NoSQL databases to support scalable persistence of large models by exploiting the index-free adjacency of nodes provided by these stores. Prototypes for integrations of both Neo4J and OrientDB with EMF have been implemented and demonstrate performance results which surpass XMI as well the Teneo/Hibernate solution.[4]

• **VCS components**: specific for each version control system, these components take a VCS URL and a revision number/id (representing the current top-level revision in the relevant index) and compute a set of changed files with respect to that revision. Currently supported version control systems are Subversion repositories, local folders, Git repositories (treated as a folder where the .git subfolder is ignored) and Eclipse workspaces.

• **Core component**: responsible for initializing, managing and gracefully terminating Hawk. This component keeps track of which model indexers are active and synchronize the version control system(s) at time intervals defined by some heuristic. This comprises getting the current indexer revision, querying the VCS for changed files, giving these files to the parser and finally passing the returned resources to the appropriate indexer(s).

• **GUI component**: provides a front-end to Hawk, thus allowing for indexes to be added/removed and have their status checked. It can also be used to perform pre-defined queries on the model indexes such as total number of elements in the index or verification queries (such as that there are no references pointing to unresolvable proxies).

• **Query API**: provides a bridge between Hawk and modelling and model management tools that need to query its indexes. The query API has been used to integrate model management frameworks such as Epsilon [21] or the query languages developed in Work Package 3.
Figure 4: Interfaces of Hawk
3 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.

3.1 Back-end Persistence

As mentioned above, Hawk focuses on the use of Graph-based NoSQL databases for its back-end. Initial evaluation of these stores was performed between two largely used graph databases – Neo4J and OrientDB. While 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 reduce its performance for large models; detailed analysis of this evaluation is presented in Section 5.3.1. For this reason, the Neo4j store received more attention during the early stages of MONDO. Between D5.3 and D5.5, the OrientDB backend was updated to work with the current architecture of Hawk. Both backends will be discussed within this deliverable.

3.1.1 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\(^5\). 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\(^6\), 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. Neo4j is published under a dual licensing scheme, with several commercial editions that can be embedded in closed source projects, and an open source edition under version 3 of the GNU General Public License.

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)

\(^5\) [http://neo4j.com/](http://neo4j.com/)
6. a set of files storing (Lucene) index information
7. a set of files containing the latest logical logs (the latest transactions committed 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, which are 64-bit integers.

3.1.2 Background: OrientDB

OrientDB is a Java-based document-oriented database with support for direct links between documents, which has evolved to support various usage scenarios, such as object-oriented databases or graph databases. Most components (including support for multi-master replication) are available under the open source Apache Software License 2.0: the Enterprise Edition adds additional development and monitoring tools and distributed clustering capabilities.

OrientDB can store documents following a schema-free or schema-aware approach, and documents are assigned to a “class” (their semantic type) and occupy a position within one of the “clusters” (their unit of physical storage) assigned to the class. Using different clusters, it is possible to split the data for more efficient retrieval of specific subsets: clusters are also important for the multi-master operational mode of OrientDB, as it allows multiple insertions to be performed concurrently. Hawk currently uses OrientDB in local embedded mode, following a schema-free approach and with different OrientDB document classes for each type of node shown in Section 2.2.1. Edges are represented by pairs of document links between their sources and targets.

Like most graph databases today, OrientDB contributes its own implementation of the generic Tinkerpop Blueprints API for graph querying, and has support for the Gremlin query language in addition to its own SQL-like query language. Nevertheless, internal testing while upgrading the Hawk OrientDB backend to the 2.x releases found that the native Java APIs had much better insertion performance than the Blueprints API. The Hawk OrientDB backend can operate in two modes: the regular transactional mode used to update existing models, and a “batch mode” for new models which temporarily disables transactions and persists changes to disk in batches to improve performance while placing an upper limit on the memory usage, making it possible to index large models with millions of elements.

It is possible to index graph nodes by the value of their attributes, using several algorithms: SB-Tree is the recommended default and can support all types of values, Hash is faster for direct lookup but does not support range queries, and Lucene is specialized for full-text queries. Index keys can be scalar values or tuples of specific types, and indexes can be fine-tuned for unique or non-unique keys. The Hawk OrientDB backend uses SB-Tree indexes exclusively, as Hawk requires range queries and must index not only string-based keys, but also integer and boolean keys.

---

7Neo4J supports the use of a custom configuration embedded Lucene index implementation for indexing commonly accessed node properties (either automatically or manually). In section 3.8, 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.

8http://orientdb.com/
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 ones presented above), two mapping layers had to be introduced by Hawk.

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, 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 5):

- **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 IHawkDataType) it contains (by name), or all of them.
  - **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 IHawkClasses 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 (aka 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 (aka 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 (integer proxy to its current state); this number 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 3.5.2).
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) \cite{2, 22} for the types of queries that are of interest to a system like Hawk. 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, link-
ing 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 6):

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

![Figure 6: The Hawk Graph Layer](image)

**IGraphDatabase**

- `getNodesByIds(ids: Object[]): IGraphNode` - Retrieves nodes by their IDs.
- `createNode(properties: Map<String, Object>, type: String): IGraphNode` - Creates a new node with specified properties and type.
- `createRelationship(source: IGraphNode, end: IGraphNode, type: String, properties: Map<String, Object>): IGraphEdge` - Creates a new relationship between two nodes.
- `getAllNodes(label: String): Iterable<IGraphNode>` - Gets all nodes with a specific label.
- `beginTransaction(): IGraphTransaction` - Begins a new transaction.

**IGraphTransaction**

- `success(): void` - Marks the transaction as successful.
- `void failure(): void` - Marks the transaction as failed.
- `void close(): void` - Closes the transaction.

**IGraphChangeListener**

- `getName(): String` - Gets the name of the listener.
- `synchroniseStart(): void` - Marks the start of a synchronization.
- `synchroniseEnd(): void` - Marks the end of a synchronization.
- `changeStart(): void` - Marks the start of a change.
- `changeSuccess(): void` - Marks the success of a change.
- `changeFailure(): void` - Marks the failure of a change.
- `metamodelAddition(): void` - Handles the addition of a metamodel.
- `classAddition(class: I HawkClass, baseClass: IGraphNode): void` - Handles the addition of a class.

**IGraphNode**

- `getId(): Object` - Gets the ID of the node.
- `getPropertyKeys(): Set<String>` - Gets the keys of the properties.
- `getProperty(name: String): Object` - Gets the value of the specified property.
- `setProperties(name: String, value: Object): void` - Sets the value of the specified property.
- `getEdgesWithLabel(type: String): Iterable<IGraphEdge>` - Gets edges with the specified type.
- `getOutgoingWithLabel(type: String): Iterable<IGraphEdge>` - Gets outgoing edges with the specified type.
- `getIncomingWithLabel(type: String): Iterable<IGraphEdge>` - Gets incoming edges with the specified type.
- `delete(): void` - Deletes the node.
- `removeProperty(name: String): void` - Removes the specified property.

**IGraphEdge**

- `getId(): Object` - Gets the ID of the edge.
- `getLabel(): String` - Gets the label of the edge.
- `getProperties(): Map<String, Object>` - Gets the properties of the edge.
- `getReference(): INode` - Gets the reference to the node.
- `getReferenceType(): String` - Gets the type of the reference.
- `getReferenceLabel(): String` - Gets the label of the reference.
- `getReferenceStart(): INode` - Gets the start node of the reference.
- `getReferenceEnd(): INode` - Gets the end node of the reference.
- `getDirection(): String` - Gets the direction of the reference.
- `addNode(node: IGraphNode, props: Map<String, Object>): void` - Adds a node with properties to the edge.
- `removeNode(node: IGraphNode): void` - Removes a node from the edge.
- `addProperty(name: String, value: Object): void` - Adds a property to the edge.
- `removeProperty(name: String): void` - Removes a property from the edge.

**IGraphNodeIndex**

- `getName(): String` - Gets the name of the index.
- `query(key: String, valueExpr: Object): Iterable<IGraphNode>` - Queries nodes with a specific key and value.
- `query(key: String, from: int, to: int, fromInclusive: boolean, toInclusive: boolean): Iterable<IGraphNode>` - Queries nodes with a specific key and value range.
- `addNode(node: IGraphNode, props: Map<String, Object>): void` - Adds a node with properties to the index.
- `removeNode(node: IGraphNode): void` - Removes a node from the index.

**IGraphModelUpdater**

- `getName(): String` - Gets the name of the model updater.
- `updateModel(): void` - Updates the model.

**IGraphModelUpdater**

- `getName(): String` - Gets the name of the model updater.
- `updateModel(): void` - Updates the model.

**IGraphModelUpdater**

- `getName(): String` - Gets the name of the model updater.
- `updateModel(): void` - Updates the model.
• **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 *IGraphNodes*. 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.

### 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) 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 possible changes according to its current update strategy heuristic (default settings as well as the synchronization strategy Hawk uses are found in Section 3.7) and can also be prompted to synchronize on demand through its interface.

### 3.3.1 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).

### 3.3.2 Workspace Manager

Hawk supports indexing models located in a local Eclipse Workspace. This driver is an extension of the LocalFolder driver (presented in Section 3.3.1) that can also use Eclipse’s notifications to inform Hawk whenever a model file has changed, instead of only relying on Hawk’s periodic updates (presented in Section 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.

### 3.3.3 SVN Manager

This component allows Apache Subversion (SVN) repositories to be monitored by Hawk. Every time Hawk polls 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).

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 work only with model files it is interested in. 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.

Below we list the various factories currently implemented: more details are provided in D5.4 [18].

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 3.2.1) a metamodel is an IHawkPackage which has a unique global namespace identifier and contains a collection of IHawkClasses. 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 IHawkPackages (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 EPackages with EClasses. 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 in Hawk is defined as 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 IHawkModelResources created by parsing the file-based persistence, and containing a collection of IHawkObjects. 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 EObjects with values for their relevant features. As such, the implementation of this component comprises of producing wrappers around the EMF resource and its contents (the EObjects).

### 3.4.2 Other Resource Factories

Several other resource factories were created for Hawk, in order to allow it to index other technologies than basic EMF, such as Industry Foundation Classes models, Business Process Model and Notation models and Modelio UML models. More details about these factories are presented in D5.4 [18].

### 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.

#### 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 IHawkClasses. Hawk’s Graph Layer (Section 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.
3.5.2 Model Updater

When using the default model updater\(^{10}\), Hawk performs Algorithm 1 every time it finds a changed (added, removed, updated) model file from any of its monitored repositories. As demonstrated in the sequel, using this incremental updating when a model file is already indexed provides a large performance gain when compared to naively deleting and re-indexing it every time it is modified.

Algorithm 1: Hawk Update Overview

```
if model file already indexed or model change size greater than threshold then
    if change of type added/updated then
        incremental update
    else if change of type removed then
        delete indexed elements of file, keeping cross-file references to these elements as proxies
    end
else
    if change of type added/updated then
        naive insertion
    end
end
```

Naive Insertion For a naive 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 at least an order of magnitude better performance in terms of execution time when compared to on-line (transactional) updating. In the case of the OrientDB backend, transactions and the Write Ahead Log are disabled, speeding up the process considerably and reducing disk usage when loading very large models.

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

---

\(^{10}\) 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.

\(^{11}\) As an incremental update of a large number of elements is time consuming, a threshold can be provided to direct any update with too many changes to use naive insertion instead in order to improve its execution time.
Algorithm 2: Naive (batch) 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
   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)
3. endforeach
4. foreach element in the model resource do
   5. foreach reference in the references of the element do
      6. if reference of element is set then
         7. foreach referenced element do
            8. if referenced element is not a proxy then
               9. create relationship from this node to the node of the referenced element
            10. else
                11. add new proxy reference
            12. end
         13. end
      14. end
   15. end
5. end

feature of the element needs to be accounted for. As such, the following features are used to calculate the signature of 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\(^{12}\)) 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\(^{13}\) digest representation as this identifier, to allow rapid comparisons.

This signature is used to efficiently find changes in model elements, as detailed below.

---

\(^{12}\)One-way hash function that takes arbitrary-sized data and outputs a fixed-length hash value.

\(^{13}\)A cryptographic hash function designed by the United States National Security Agency; it produces a 20 byte value.
Algorithm 3: Incremental update algorithm

let \( nodes \) be the set containing the ids and pointers to all the nodes (in the model index – linked with the updated file)

let \( signatures \) be the set containing the ids and signatures of the \( nodes \)

let \( delta \) be the set containing changed elements

let \( added \) be the set containing new elements (to be added to the model index)

let \( unchanged \) be the set containing elements which are the same

foreach node from all nodes in the model index that are linked with the updated file do

| add node to \( nodes \ |

| add signature of node to \( signatures \ |

end

foreach element in elements of model resource do

| if element id exists in \( signatures \ |

| if element signature not equal to current signature then

| \( \text{add element to} \ \delta \) |

| else

| \( \text{add element to} \ \text{unchanged} \) |

| end |

| else

| \( \text{add element to} \ \text{added} \) |

| end

end

/* add new nodes to model index */

foreach element in \( added \) do

| add this new element in model file to model index

end

/* delete obsolete nodes and change node attributes */

foreach node in \( nodes \) do

| if node id exists in \( delta \ |

| retrieve element in \( delta \) represented by the node

| foreach model attribute in element do

| if attribute value is different to node property value then

| update property value

| end |

| end |

else if node id does not exist in \( unchanged \ |

| de-reference node (keeping dangling incoming cross-file references as proxies)

| delete node

| end |

end

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

```java
/* change altered references */

foreach element in delta do
    foreach reference in references of element do
        if reference is set then
            foreach referenced element in referenced elements of reference do
                if referenced element is not proxy then
                    add id of referenced element to targetIds
                else
                    add new proxy reference to model index
            end
        end
        foreach relationship in relationships of node linked with the element do
            if relationship target has id which exists in targetIds then
                remove target from targetIds
            else
                delete relationship as new model does not have it
            end
        end
    end
    foreach id in targetIds do
        add new relationship to model index
    end
else
    foreach relationship in relationships of node, with the same name as the reference name do
        delete this relationship
    end
end
```

**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 $O(m + n + d \times r)$ 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, $d$ is the number of changed elements and $r$ is the number of target elements referenced by the changed element.
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 (the metamodel is seen in Figure 7a and a key to the model diagram is seen in Figure 7b). In this scenario, assume that the following changes have been made to the original BPMN loan model in Figure 8 (the new model is seen in Figure 9):

1. The attribute name of Task named “Check Applicant Info” is changed to “Verify Applicant Information Authenticity”
2. A new Task with name “Notify Authorities of Fraud” is introduced
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 Sequence-Flow 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 Authorities 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):

---

Figure 8: Running Example – original BPMN loan model

Figure 9: Running Example – altered BPMN loan model
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.

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: native querying and back-end independent querying. The rest of the section compares these approaches.

3.6.1 Native Querying

The most straightforward, and often the most performant, way 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.
- 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.
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 the Object Constraint Language (OCL) and the Epsilon Object Language (EOL) [14] (from the Epsilon [21] platform), which abstract over concrete model representation and persistence technologies using intermediate layers such as the *OCL pivot metamodel* [25] and *Epsilon Model Connectivity* [12] layer.

During the MONDO project, an Epsilon Model Connectivity driver has been developed for Hawk, making it possible to query Hawk indexes concisely while taking advantage of derived and indexed attributes, among other advanced features. More details are provided in D5.4 [13].

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 heuristic (time is in seconds, and the minimum and maximum values are configurable and have default values of 1 and 512 seconds, respectively):

    ```
    if (changes == false)
      if (timeToUpdate < maximum)
        timeToUpdate = timeToUpdate * 2
      else
        timeToUpdate = minimum
    ```

    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).

### 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/re-named/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.

Algorithm 4: Synchronization Procedure

```plaintext
if existsInteresting() then
  for modelFile m : modelFiles() do
    if added(m) then
      Parse the model file into a resource. Insert it into the index (refer to Section 3.5.2).
    else if removed(m) then
      Delete the model from the index.
    else if updated(m) then
      Update the model in the index (refer to Section 3.5.2).
    else if (renamed(m) or moved(m)) then
      Update the model in the index (refer to Section 3.5.2).
  end
  resolveModelProxies()
end
scheduleNextUpdate()
```

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 Section 4.2. As mentioned above, regardless of input, an *IHawkMetaModelResource* is created from the file and is ready to be parsed for database insertion. 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 [2].

**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.

**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 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 meta-
model(s); when the relevant metamodel is inserted the model will also be inserted in the next syn-
chronization 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 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.

### 3.8 Advanced Features and Optimizations

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

#### 3.8.1 Derived Attributes

Regardless of the use of native or back-end independent querying, in order to respond to a query
requesting slow BPMN Tasks (instances with execution time percent deviation\(^{14}\) of more than +25%),
the following steps would have to occur:

\[ \frac{v - m}{m} \times 100\% \text{ where } v \text{ is the value and } m \text{ is the mean of all the values} \]
1. The starting point of the query would have to be found. In this case, the collection of all instances of Task in the model would have to be retrieved.

2. For each task node, its execution time would have to be retrieved and recorded alongside all other execution times of other tasks, in order to compute a mean. If the query language does not support the use of variables then this process would have to happen redundantly for each task – otherwise this only needs to be computed once.

3. For each task node, its execution time would have to be compared to the mean computed above and if it is greater than 125% of the mean, the node is returned as one of a slow task.

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 Task 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 Task type.

Step 2 where we can begin optimizing to improve the execution time of queries which have to perform some non-trivial calculation or search 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.

Such attributes are computed using expressions formed in the expression language of a known Query Engine. A query engine, as discussed in Section 3.6, allows for expression languages (such as OCL or Epsilon’s EOL [14]) 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 10: Pre-computing whether a task is slow
A simple example is shown in Figure 10; here, slow tasks are pre-computed (using the EOL program presented in Listing 1) and this information is stored in a new DerivedAttribute node 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 get slow tasks would change from the one presented above to Task.all.select(a | a.isSlow) (in both cases returning a collection containing the slow tasks).

Listing 1: EOL program used to calculate whether a task (self) is slow

```javascript
var totalExecutionTime = 0;
var totalTasks = 0;
var mean;
for(a in Task.all){
    totalExecutionTime += a.executionTime;
    totalTasks++;
}
mean = totalExecutionTime / totalTasks;
return self.executionTime / mean > 1.25;
```

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

### 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 [7], 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 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:

15 A new node is used for each derived attribute to overcome a limitation found during incremental updating of derived attributes; further information on this can be found in Section 3.8.2
16 A derived attribute cannot depend on another derived attribute: this restriction is imposed to reduce the inherent complexity of the incremental derived attribute update process.
A model element has been created / deleted
A property of a model element has been altered
A reference of a model element (to another one) has been created / deleted
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 a property / reference of a model element
- Access to the collection of model elements of a specific type / 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.

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 change then
4.     add any derived attribute which accesses this element (or any of its structural features) to nodesToBeUpdated
5.   else if the change is a structural feature change 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

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 (or if a Task is added/removed). As demonstrated by [8], 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 [8].
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 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 through 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 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.

Figure 11: Example of database indexing in Hawk: execution times are keys to their BPMN Tasks
### 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 Epsilon 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. 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.

It is notable that such `OptimisableCollections` are only created when a `.getAllofType` or `.getAllofKind` operation is called in an EOL program and hence its contents are all of the same `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 is optimizeable should any of the attributes of the `type` be database indexed.

Now, should a `select` operation (or a `select`-based operation like `reject`, `exists`, `one`, or `selectOne`) 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:

- If the `select` condition contains an expression of the form “`x.attr = value`” (thus is optimizeable 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.
- If the `select` condition contains an expression of the form “`x.attr rel value`” where ‘`rel`’ is among `{<, >, <=, >=}` and ‘value’ is a numeric (real or integer) value (also optimizeable by use of custom attribute indexes), it will check if an index of attribute ‘`attr`’ exists for elements of that type. If such an index exists, it will perform a range query to directly get the subcollection required without traversing all the persisted elements. The indexes used in the Neo4j and OrientDB backends support range queries for integer and real values.
- If the `select` condition contains an expression of the form “`a and b`” where ‘`a`’ and ‘`b`’ are any sub-expressions, it will first check whether ‘`a`’ or ‘`b`’ are optimizeable. If both are optimizeable, it will return a collection containing the set intersection of the results of ‘`a`’ and the ones of ‘`b`’. If only one of the two is optimizeable, it will create a collection of the relevant results and pass
that collection to the default EOL driver for parsing the remainder of the expression (effectively performing efficient partial filtering). If none of the two expressions are optimizable it will propagate the sub-expressions of ‘a’ and ‘b’ to be (potentially) optimized and then will take their results and perform a set intersection on them.

- If the select condition contains an expression of the form “a or b” where ‘a’ and ‘b’ are any sub-expressions, it will first check whether ‘a’ and ‘b’ are both optimizable. If both are optimizable, it will return a collection containing the set union of the results of ‘a’ and ‘b’. If one or none of the two expressions are optimizable it will propagate the sub-expressions of ‘a’ and ‘b’ to be (potentially) optimized and then will take their results and perform a set union on them.

- If the select condition contains an expression of the form “a xor b” where ‘a’ and ‘b’ are any sub-expressions, it will first check whether ‘a’ and ‘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 will propagate the sub-expressions of ‘a’ and ‘b’ to be (potentially) optimized and then will take their results and perform the abovementioned exclusive or comparison on them.

- If the select condition contains an expression of the form “a implies b” where ‘a’ and ‘b’ are any sub-expressions, it will first check whether ‘a’ and ‘b’ are both optimizable. If both are optimizable, it will return the original collection with members of it which satisfy ‘a’ removed 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 one the filter of ‘b’ runs on. It will then create a collection containing elements of the sub-collection of ‘a’ minus the ones 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 propagate the sub-expressions of ‘a’ and ‘b’ to be (potentially) optimized and then will take their results and perform the abovementioned implies comparison on them.

- If the select condition contains an expression of the form “not(a)” where ‘a’ is any sub-expression, it will first check whether ‘a’ is optimizable. If it is, it will return the result of the original collection minus the elements satisfying ‘a’. If it is not, it will propagate the sub-expression of ‘a’ to be (potentially) optimized and then will take the result and perform the abovementioned not operation on it.

- Should the select condition contain any other expression (or sub-expression) it delegates to the default Epsilon SelectOperation execute() method, to perform the job.

3.8.5 Hawk Validation Listener

Using Hawk’s update notification framework (presented in Section 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 synchro-
nize 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.
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
      else
         10. 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
         11. 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
      end
   end
   12. if there are any left over model elements not mapped to a node then
      13. inform that validation has failed alongside information about the model elements not found in Hawk
   end
   14. if totalGraphSize not equal to totalResourceSize then
      15. inform that validation has failed as Hawk and the loaded resources do not contain an identical number of model elements
   end

4 Implementation

This section details the implementation details of the current Hawk prototype and presents a concise user guide for running this tool.

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 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.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.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.ifc plugin This plugin contains the necessary implementations for reading Building Information Modeling (BIM) Industry Foundation Classes (IFC)18-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.

18http://www.buildingsmart.org/
• **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.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.neo4j-v2 plugin** This plugin contains the necessary implementations for connecting with a Neo4J NoSQL database. It depends on the core plugin, the graph 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` files 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.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 the `org.tmatesoft.svnkit` and `org.tmatesoft.sqlj.et` plugins. These plugins can be found in the Eclipse update site of the official svnkit website. It implements an extension for Hawk’s `VCSExtensionPoint` 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.ui2 plugin** This plugin contains the necessary implementations for offering a user interface for Hawk integrated into Eclipse. It depends on the core plugin, the core.dependencies plugin, as well as various standard Eclipse UI plugins.

• **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. It implements extensions for Hawk’s `ModelExtensionPoint` and `MetaModelExtensionPoint` extension points.

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 Section 4.2 below.

[^19]: http://download.eclipse.org/bpmn2-modeler/updates
4.2 User Guide

Hawk offers an Eclipse-based user interface for running and maintaining model indexes. Figure [12] 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).

Figure 12: 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 [13] 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 [14] 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 [15].

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 [16]. After clicking the “OK” button in the window the chosen folder is now indexed in Hawk, as in Figure [17].

Now that a folder is indexed, in this case one with the model set0.xmi inside it, 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 [18]. This program points to the file grabats.eol as expected. 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 [19]. This opens the configuration window shown in Figure [20], where model can be named and pointed to 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 if desired. Now the query can be run on the Hawk model by pressing the “Run” button, as shown in Figure [21].
The Hawk model index can be further optimized by adding derived attributes (described in Section 3.8). This can be done by selecting the “Derived Attributes” tab of the configuration window of Hawk, as shown in Figure 22. By clicking the “Add..” button in the window the option to create a new derived attribute for Hawk is available, as shown in Figure 23. After clicking “OK” the new derived attribute that has been added is seen, as shown in Figure 24.

Finally, (database) indexed attributes can be added, as described in Section 3.8. This is done by selecting the “Indexed Attributes” tab of the configuration window of Hawk, as shown in Figure 25. 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 26.

A complete set of screencasts for running Hawk can be found online.

https://github.com/mondo-project/mondo-hawk
https://www.youtube.com/channel/UCfJydYyyFCeg60kaSzC-LQ
Figure 14: Metamodel Configuration

Figure 15: JDTAST Metamodels Added to Hawk
Figure 16: Indexed Folder Configuration

Figure 17: Folder added to Hawk
Figure 18: Setting the source EOL file

Figure 19: Adding a Hawk model
Figure 20: Configuring the Hawk model

Figure 21: Running the EOL Query
Figure 22: Configuring Derived Attributes

Figure 23: Adding a new Derived Attribute
Figure 24: Viewing current Derived Attributes

Figure 25: Adding a new Indexed Attribute
Figure 26: Viewing current Indexed Attributes
5 Evaluation

To evaluate the features of Hawk, we have performed a set of experiments. In this section we present these experiments, and where appropriate we compare the obtained results to existing technologies. A detailed survey of such technologies can be found in deliverable D5.1 [17].

5.1 Overall 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.

5.1.1 Tool correctness

Evaluating the correctness of Hawk can be broken down into two aspects: the correctness of the model index and the correctness of the results returned by querying this index.

Index 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 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 abovementioned changes. 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 3.2.2) and run the Hawk validation listener (Section 3.8.5) after Hawk attempts to perform any synchronize with its monitored repositories.
Refer to sections 5.3.4 and the discussion of the IncQuery Train Benchmark in deliverable D5.4 [18] for empirical data collected on tool correctness.

**Query correctness** In order to ensure correct results to queries performed on Hawk, 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. 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 and the discussion of the IncQuery Train Benchmark in D5.4 [18] for more details.

### 5.1.2 Tool performance

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

**Query performance** Hawk aims at providing a scalable way to perform 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 a uniform manner. As described in Section 2.2.1, as the contents of Hawk’s model indexes are stored as homogeneous property graphs, by construction, it does not matter the types or number of models stored in Hawk, the approach used for returning results to a global query is the same. Empirical data gathered on querying large models can be found in Section 5.3.1 and improvements observed when using derived and indexed attributes can be found in Section 5.3.3.

**Update performance** Hawk claims to have the ability to handle constantly evolving models stored in version control systems. 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 consistent 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 the various MONDO partners), various tools have been integrated with Hawk. Details can be found in deliverable D5.4 [18].

5.2 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.

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 use case[21] presented in the Grabats 2009 contest [11] described below, as well as the five models also provided in the contest.

Figure 27: Small subset of the Java JDTAST metamodel

A subset of the Java JDTAST metamodel is presented in Figure [27]. In this figure, there are *Type-Declarations* that are used to define Java classes and interfaces, *MethodDeclarations* that are used to define Java methods (in classes or interfaces, for example) and *Modifiers* that are used to define 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 70447 model elements (set0) to one with 4961779 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) [23]. 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).

**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, or general-purpose languages such as Epsilon’s EOL.

Below we see the Grabats query written in these three forms. Listing 2 requires knowing the native API of the specific backend being used and the structure of the underlying graph. Listing 3 replaces the native API with the Cypher query language: in addition to being coupled to Neo4j, it still requires knowing the structure of the underlying graph and its large amount of punctuation makes it hard to read. Listing 4 is a query written in the Epsilon Object Language using the Hawk EMC EOL driver mentioned in Section 3.6.2: it does not depend on any backend, it is the most concise of the three and it does not require knowledge about the underlying graph.

Listing 2: Code excerpt for the Grabats query implemented in Java for Neo4J

```java
1 {
2    ...
3 109    for (Relationship outEdge : typeDeclaration.getRelationships(
4      Direction.OUTGOING, DynamicRelationshipType.withName("bodyDeclarations"))) { 
5 110      Node methodDeclaration = outEdge.getEndNode();
6 111      if (new MetamodelUtils().isOfType(methodDeclaration, new
7        MetamodelUtils().eClassNSURI(methodDeclarationClass))) {
8        boolean isPublic;
9        boolean isStatic;
10       String currMethodName;
11      115      for (Relationship methodDeclarationOutEdge : methodDeclaration.
116      getRelationships(Direction.OUTGOING, DynamicRelationshipType.
117      withName("name"))) {
118        Node name = methodDeclarationOutEdge.getEndNode();
119    }
120  }
121 }
```
currMethodName = name.getProperty("fullyQualifiedName").toString();
for (Relationship methodDeclarationOutEdge : methodDeclaration.getRelationships(Direction.OUTGOING, DynamicRelationshipType.withName("modifiers"))) {
...
}

Listing 3: Code excerpt for the Grabats query implemented in Cypher for Neo4J

```java
ExecutionEngine engine = new ExecutionEngine(graph);
ExecutionResult result = engine.execute("START dom=node:METAMODELINDEX('id:org.amma.dsl.jdt.dom') MATCH dom<-[t|-(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]]-(returntypename)
WHERE nodename.fullyQualifiedName==returntypename.
MATCH methodnode-[[:name]]-(methodnodename)
RETURN DISTINCT nodename.fullyQualifiedName,methodnodename.

");
```

Listing 4: The Grabats 2009 query expressed in EOL

```java
TypeDeclaration.all
  .collect(td|td.bodyDeclarations
  .select(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
  )
  )
  .flatten();
```

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.
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. 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.

Currently (October 2015), 28 tools provide 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 providing Hawk with this repository and its various versions.

## 5.3 Evaluation Results

This section presents an analysis of the empirical data obtained during this work, using the evaluation benchmarks described above. As this work spans multiple years, the execution environment used may vary, and so is explicitly stated in each case.

### 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. This work has been published in [2].

**Execution Environment**  The presented performance figures have been measured on a PC with an Intel(R) Core(TM) i5-2300 CPU @ 2.80GHz, with 8GB of physical memory, and running the Windows 7 (64 bits) operating system. 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 1: Configuration Options for Benchmarks

<table>
<thead>
<tr>
<th>Config</th>
<th>XMI</th>
<th>Teneo/Hibernate</th>
<th>Persistence Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>n/a</td>
<td>default</td>
<td>default</td>
</tr>
</tbody>
</table>

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

**Model Insertion**  Tables 2 and 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 a timeout 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. It is worth noting that for set3 – set4, due to the sizes of the files, the computer’s RAM is exhausted hence the operation is greatly bottlenecked by I/O from the hard disk. This results in a greater variance in the results and hence the averages presented here are influenced by multiple factors such as the physical location of each database on the hard disk.

**Query Execution Time and Memory Footprint**  Table 4 shows the results for performing the Grabats query (Section 5.2.1) on the databases. As previously mentioned, the Grabats query finds all occurrences of TypeDeclaration elements that declare at least one public static method with the declared type as its returning type.

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[Neo4J] demonstrates the best performance in terms of execution time and Hawk[OrientDB] is faster than XMI but also uses a comparable memory footprint. CDO has the lowest memory consumption for the queries it can run
Table 2: Model Insertion (Persistent to Database) Size Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set0</td>
<td>8.75</td>
<td>38.6</td>
<td>26.0</td>
<td>34.8</td>
<td>29.4</td>
</tr>
<tr>
<td>Set1</td>
<td>26.59</td>
<td>83.1</td>
<td>67.0</td>
<td>75.7</td>
<td>85.9</td>
</tr>
<tr>
<td>Set2</td>
<td>270.12</td>
<td>-</td>
<td>539</td>
<td>551</td>
<td>794</td>
</tr>
<tr>
<td>Set3</td>
<td>597.67</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1750</td>
</tr>
<tr>
<td>Set4</td>
<td>645.53</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1890</td>
</tr>
</tbody>
</table>

Table 3: Model Insertion (Persistent to Database) Execution time Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set0</td>
<td>n/a</td>
<td>58.7</td>
<td>11.8</td>
<td>26.2</td>
<td>12.4</td>
<td>19.6</td>
</tr>
<tr>
<td>Set1</td>
<td>n/a</td>
<td>218.2</td>
<td>19.2</td>
<td>66.7</td>
<td>32.5</td>
<td>57.1</td>
</tr>
<tr>
<td>Set2</td>
<td>n/a</td>
<td>-</td>
<td>778.5</td>
<td>647.5</td>
<td>499.1</td>
<td>590.8</td>
</tr>
<tr>
<td>Set3</td>
<td>n/a</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2210</td>
<td>2245</td>
</tr>
<tr>
<td>Set4</td>
<td>n/a</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2432</td>
<td>2397</td>
</tr>
</tbody>
</table>

(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[Neo4J] and comparable to Hawk[OrientDB].

Using this empirical data we can deduce that even though Hawk[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[Neo4J], which uses a pure graph-based database.

The last two columns of Table 4 show the results for performing the Grabats query using the EMC query on the databases. 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 29 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[Neo4J] query execution times (seen in Table 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[^Neo4J] is preferable to XMI after around 35 repeats while Hawk[^OrientDB] after around 90. For set3, Hawk[^Neo4J] is around 28 while Hawk[^OrientDB] 37. For set4, Hawk[^Neo4J] is around 18 while Hawk[^OrientDB] 21.

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 effects its overall performance causing to be quickly surpass that of XMI for any model size.

**Disc Space**  
As expected, Hawk[^Neo4J] and Hawk[^OrientDB] require more disk space than XMI. Figure 28 shows the ratios of relative disk space needed to store the different models (set0 – set4) for the different technologies, using the results in Table 2.

![Figure 28: Ratios of relative disk space used for the different persistence mechanisms](image.png)

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 a Hawk[^OrientDB] store to be around 4.3x as large as its XMI file and a Hawk[^Neo4J] store around 2.9x as large. Furthermore the results seem to show that Hawk[^Neo4J] is more efficient in storing the data relative to Hawk[^OrientDB], which can be expected as it handles references in a more lightweight fashion, as explained in Section 3.1. The ratio between Hawk[^Neo4J]
and Hawk\textsuperscript{[OrientDB]} seems to indicate that for both databases their relative overheads discussed above are similar, but Hawk\textsuperscript{[OrientDB]} is less efficient in that regard (with a 19.2\% delta in the ratio between Hawk\textsuperscript{[Neo4J]} and Hawk\textsuperscript{[OrientDB]} at set0 and the one at set4).
Table 4: Grabats Query Results (in seconds and MB)

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set0</td>
<td>Time: 1.20, Mem (Max): 42, Mem (Avg): 19</td>
<td>1.20</td>
<td>453</td>
<td>0.60</td>
<td>0.11</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
<tr>
<td>Set1</td>
<td>Time: 16.51, Mem (Max): 813, Mem (Avg): 432</td>
<td>16.51</td>
<td>1294</td>
<td>12.2</td>
<td>20</td>
<td>11.1</td>
</tr>
<tr>
<td></td>
<td>Time: 84.91, Mem (Max): 1750, Mem (Avg): 844</td>
<td>84.91</td>
<td>1750</td>
<td>12.0</td>
<td>401</td>
<td>195</td>
</tr>
</tbody>
</table>

Set2

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set1</td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
</tbody>
</table>

Set3

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set1</td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
</tbody>
</table>

Set4

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set1</td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
</tbody>
</table>

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set1</td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
</tbody>
</table>

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set1</td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
</tbody>
</table>

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set1</td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
</tbody>
</table>

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set1</td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
</tbody>
</table>

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set1</td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
</tbody>
</table>

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set1</td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
</tbody>
</table>

| Model | Persistence Mechanism | XMI | Teneo/Hib | CDO | Hawk | Hawk (OrientDB)
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set1</td>
<td>Time: 2.28, Mem (Max): 111, Mem (Avg): 48</td>
<td>2.28</td>
<td>734</td>
<td>112</td>
<td>11.2</td>
<td>0.62</td>
</tr>
</tbody>
</table>
Figure 29: Performance Comparison for full execution of 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 [4].

Execution Environment  The presented performance figures 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. 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 language [14]. EOL is an imperative OCL dialect which supports model modification. 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 (available online). By using these operations in an EOL script we can change the model it is run on in a realistic yet sufficiently random manner.

Model Update Execution Time Table 5 shows the average time taken to complete an update for the models produced by performing the model mutations presented in Section 5.3.2 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 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.

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

---

26 https://github.com/kb634/mondo-hawk/blob/master/model_manipulations.eol
27 Operations often used in manipulation of Java code, such as deleting a Java class
28 For example, by randomizing which Java class is deleted each time
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. 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.

**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 updated. As this approach would have been inefficient compared to the incremental one, it was never implemented so a meaningful comparison cannot be made.

---

29 A study on fragmenting large monolithic models with EMF-Splitter found that the largest fragment of several models with nearly 5 million elements had about 50,718 elements.

30 As it does not block any incoming queries which may need to be performed

31 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.
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 fragmented models, 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.

- Using an integer representation for the signatures 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 and the discussion of the IncQuery Train Benchmark in D5.4 [18] 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 [3] evaluates the performance benefit of using such features for query execution.

Execution Environment

The presented performance figures 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. 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).

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

### Table 6: Grabats Query Execution Time Results

<table>
<thead>
<tr>
<th>Model</th>
<th>Execution Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td></td>
<td>Q1</td>
</tr>
<tr>
<td><strong>Set0</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.040</td>
</tr>
<tr>
<td><strong>Set1</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.081</td>
</tr>
<tr>
<td><strong>Set2</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.850</td>
</tr>
<tr>
<td><strong>Set3</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.664</td>
</tr>
<tr>
<td><strong>Set4</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.124</td>
</tr>
</tbody>
</table>

**Query Definition and Execution Time**  
Table 6 shows the results for performing the first Grabats 2009 query on the various persisted models. For these tests five queries have been written in EOL (Q1 – Q5):
• **Q1** reads:

```java
TypeDeclaration.all.collect(
    td|td.bodyDeclarations.select(
        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 ) )
```

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:

```java
TypeDeclaration.all.collect(
    td|td.bodyDeclarations.select(
        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:

```java
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:

```java
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 that the MethodDeclarations 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 [3] (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 yet rapidly evolving models. This section presents the methodology used and the results obtained during this experiment.

Execution Environment The presented performance figures 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. 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 selected 25 commits, Hawk was initially provided with the first (chronologically) commit and then each successive commit to synchronize with. Figure 30 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 30a), 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.

**Figure 30: BPMN benchmark model change results**

**Update Performance Results**  Figure 31 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 \( \sim 38\% \) in overall time taken to complete the benchmark (taking around 6.2 and 4.2 minutes respectively).
Figures 32 and 33 display the memory use of Hawk throughout this benchmark (using Neo4J and OrientDB respectively). Line $L_1$ denotes the time when the synchronization with the first commit finishes and line $L_2$ denotes the time when the synchronization with the last commit finishes. Line $L_3$ denotes the point in time when Hawk’s memory use stops fluctuating as the repository state is unchanging and Hawk has no other operations to perform (other than periodically confirming that there are no pending changes to synchronize with). Looking at the time-line between the start of the benchmark and $L_1$, 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$^{\text{Neo4J}}$, the old gen space used is largely for the embedded MMIO of Neo4J (hence it rapidly reaches a peak (of around 250MB) and remains there for the duration of the update), while in Hawk$^{\text{OrientDB}}$ (whereby the MMIO is not embedded into the Java Heap but directly uses system RAM), the old gen 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 of Eden space used during the update process.

Looking at the time-line between $L_1$ and $L_2$, Hawk’s memory periodically spikes as numerous batch inserts 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$^{\text{Neo4J}}$, the memory used keeps resetting down to its MMIO baseline every time the Java garbage collector deems necessary, while in the Hawk$^{\text{OrientDB}}$ case it keeps resetting down to nearly 0 as the MMIO is not embedded into the Java Heap. The time between $L_2$ and $L_3$ denotes the brief cooling-down period after the end of the experiment until Hawk’s memory use does not fluctuate any more. Finally, the time-line between $L_3$ and the end of the recording of the benchmark shows a constant memory footprint when no changes are detected by Hawk. The old gen space keeping the Neo4J caches of recently accessed nodes remains constant and so does the Eden space used by OrientDB keeping soft references to its MMIO that is outside the Java Heap.
Update Validation Results  During the execution of this benchmark the Hawk validation listener (Section 3.8.5) 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.

5.4 Evaluation Summary

Three aspects of Hawk were evaluated: its correctness, its performance and its architecture with respect to being able to integrate with current model management tools.

With respect to correctness, Hawk’s ability to accurately index file-based models and maintain a consistent state across multiple commits was investigated. In particular, two correctness tests were run: one using a large collection of small models in the BPMN MIWG repository (and their plethora of historic versions in the repository – Section 5.3.4) and one integrating with IncQuery in order to run its train benchmark (Deliverable D5.4). In the BPMN repository case, Hawk’s validation plugin provided confidence that the insertion and update algorithms of Hawk were correct, and in the train benchmark case, the JUnit tests provided by the benchmark all passed in all cases, providing further confidence of the correctness.

With respect to evaluating the performance of Hawk, both the performance of querying Hawk as well as that of updating it were looked at. With regards to query performance (execution time), a complex query was performed on the various GraBaTs models using EOL as the expression language and Hawk’s EMC integration with Epsilon as the execution engine (Section 5.3.1). Hawk showed a reduction in execution time of up to 95% when compared to using native XMI, 91.6% when compared to using Teneo (which only managed to load set0 and set1), and 74.6% when compared to CDO (which only managed to load set0 to set2). Adding indexed and derived attributes further improved execution time, with observed gains of up to 72% over Hawk without these attributes (Section 5.3.3). Regarding index updating, the incremental strategy used in Hawk was compared to the initial naive approach (Section 5.3.2). In these tests an average of 71% decrease in time taken to update Hawk was observed, providing confidence that the incremental update process is more performant, when compared to the naive approach.

Finally, validation of Hawk’s extensible architecture was done by integrating it with various external model management tools such as Epsilon, IncQuery and EMF. This is discussed in Deliverable D5.4 [18].
Figure 32: Memory graph of full BPMN benchmark execution (Hawk\textsuperscript{Neo4J})
Figure 33: Memory graph of full BPMN benchmark execution (Hawk[OrientDB])
6 Conclusions

This deliverable reported the final results of the development of the Hawk model indexing framework in WP5 of MONDO. Integrated descriptions of the high-level requirements and architecture, the design and the implementation of Hawk were presented. These descriptions included all the advanced features developed during the project, including incremental model updates, derived and indexed attributes, scoped queries, meta-level queries and support for change notifications. A later section provided a set of experiments (extended since D5.3) on the correctness and performance of Hawk, comparing it to other technologies. Hawk is now part of the integrated platform described in D6.3 [20], and is integrated with the other languages in WP3 as described in D5.4 [18].

References


