Project Number 611125

D3.4 – Integration with Model Management Languages
Report

D5.4 – Heterogeneous Model Management Framework - Final Version

Version 1.0
16 December 2015
Final

Public Distribution

University of York

Project Partners: ARMINES, Autonomous University of Madrid, BME, IKERLAN, Soft-Maint, SOFTEAM, The Open Group, UNINOVA, University of York

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>ARMINES</th>
<th>Autonomous University of Madrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Massimo Tisi</td>
<td>Juan de Lara</td>
</tr>
<tr>
<td>Rue Alfred Kastler 4</td>
<td>Calle Einstein 3</td>
</tr>
<tr>
<td>44070 Nantes Cedex, France</td>
<td>28049 Madrid, Spain</td>
</tr>
<tr>
<td>Tel: +33 2 51 85 82 09</td>
<td>Tel: +34 91 497 22 77</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:massimo.tisi@mines-nantes.fr">massimo.tisi@mines-nantes.fr</a></td>
<td>E-mail: <a href="mailto:juan.delara@uam.es">juan.delara@uam.es</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BME</th>
<th>IKERLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daniel Varro</td>
<td>Salvador Trujillo</td>
</tr>
<tr>
<td>Magyar Tudosok korutja 2</td>
<td>Paseo J.M. Arizmendiarieta 2</td>
</tr>
<tr>
<td>1117 Budapest, Hungary</td>
<td>20500 Mondragon, Spain</td>
</tr>
<tr>
<td>Tel: +36 146 33598</td>
<td>Tel: +34 943 712 400</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:varro@mit.bme.hu">varro@mit.bme.hu</a></td>
<td>E-mail: <a href="mailto:strujillo@ikerlan.es">strujillo@ikerlan.es</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Soft-Maint</th>
<th>SOFTEAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vincent Hanniet</td>
<td>Alessandra Bagnato</td>
</tr>
<tr>
<td>Rue du Chateau de L’Eraudiere 4</td>
<td>Avenue Victor Hugo 21</td>
</tr>
<tr>
<td>44300 Nantes, France</td>
<td>75016 Paris, France</td>
</tr>
<tr>
<td>Tel: +33 149 931 345</td>
<td>Tel: +33 1 30 12 16 60</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:vhanniet@sodifrance.fr">vhanniet@sodifrance.fr</a></td>
<td>E-mail: <a href="mailto:alessandra.bagnato@softeam.fr">alessandra.bagnato@softeam.fr</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The Open Group</th>
<th>UNINOVA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scott Hansen</td>
<td>Pedro Maló</td>
</tr>
<tr>
<td>Avenue du Parc de Woluwe 56</td>
<td>Campus da FCT/UNL, Monte de Caparica</td>
</tr>
<tr>
<td>1160 Brussels, Belgium</td>
<td>2829-516 Caparica, Portugal</td>
</tr>
<tr>
<td>Tel: +32 2 675 1136</td>
<td>Tel: +351 212 947883</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:s.hansen@opengroup.org">s.hansen@opengroup.org</a></td>
<td>E-mail: <a href="mailto:pmm@uninova.pt">pmm@uninova.pt</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>University of York</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimitris Kolovos</td>
<td></td>
</tr>
<tr>
<td>Deramore Lane</td>
<td></td>
</tr>
<tr>
<td>York YO10 5GH, United Kingdom</td>
<td></td>
</tr>
<tr>
<td>Tel: +44 1904 32516</td>
<td></td>
</tr>
<tr>
<td>E-mail: <a href="mailto:dimitris.kolovos@york.ac.uk">dimitris.kolovos@york.ac.uk</a></td>
<td></td>
</tr>
</tbody>
</table>
Contents

1 Introduction 3

2 Heterogeneous Model Management Framework (Hawk) 3
   2.1 Architecture ................................................. 4
   2.2 Overview of a Hawk model index .......................... 5
   2.3 Components .................................................. 6

3 Heterogeneous Model Support 10
   3.1 Hawk model parsing interfaces ............................ 10
   3.2 Hawk model parsing components ......................... 11
      3.2.1 Eclipse Modelling Framework ......................... 11
      3.2.2 Building Information Modelling - IFC ............... 12
      3.2.3 Modelio UML models .................................. 17
      3.2.4 Business Process Model and Notation ................ 21

4 Index Querying Using Epsilon 22
   4.1 Epsilon ....................................................... 22
   4.2 Querying a Hawk Model Index Using the Epsilon Object Language ................................................. 23
      4.2.1 IModel interface method implementations ............ 23
      4.2.2 Reverse reference navigation .......................... 24
      4.2.3 Scoped queries .......................................... 25
      4.2.4 Meta-level queries ...................................... 25
   4.3 Sample Index Queries ...................................... 26

5 Integration of Additional Languages (D3.4) 27
   5.1 Common interfaces for extended EMF resources .......... 27
   5.2 Hawk EMF resource abstraction for local indexes .......... 29
   5.3 Hawk EMF resource abstraction for remote indexes ......... 31
   5.4 Integration with Epsilon Exeed: lazy browsing .......... 37
   5.5 Integration with CloudATL: consuming remote models ...... 38
   5.6 Integration with IncQuery: lazy loading with incremental querying .................................................. 39
   5.7 Future integration in scalable notations: efficient retrieval of candidates ..................................... 42

6 Conclusions 43

16 December 2015 Version 1.0
# Document Control

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>First full draft</td>
<td>3 December 2015</td>
</tr>
<tr>
<td>1.0</td>
<td>Version for submission</td>
<td>16 December 2015</td>
</tr>
</tbody>
</table>
Executive Summary

This document is the final outcome of Task 5.2 (Heterogeneous Model Management) and Task 3.6 (Integration with scalable persistence mechanisms), and includes deliverables D3.4 (Integration with Model Management Languages Report) and D5.4 (Heterogeneous Model Management Framework - Final Version). We have opted for a combined deliverable in order to reduce repetition, minimise cross-deliverable references, and improve readability as the results reported in D3.4 are highly interwoven with those of D5.4 and both were led by YORK.

The combined deliverable presents the foundations, interfaces and organisation of the heterogeneous model management framework that underpins the scalable model indexing framework implemented in Task 5.3. The final version of the scalable model indexing framework supports both Neo4j (the best performer in the technology evaluation of Task 5.1) and OrientDB (an alternative graph database with a simpler license regarding redistribution) using the common database abstraction layer.

A number of interfaces have been defined to support indexing of models persisted in heterogeneous formats, and querying indexes of such heterogeneous models. These interfaces provide a uniform extension mechanism through which support for additional modelling technologies can be implemented. Implementations of these interfaces have been developed for the Modelio UML tool native model persistence format, the IFC2x3 and IFC4 Building Information Model formats and the Business Process Model and Notation formats. In terms of querying, a component has been developed that allows users to query model indexes using the EOL model-query language.

Related to Task 3.6, two abstractions have been developed to expose Hawk indexes and their updates as EMF models and change notifications. These abstractions can be used to integrate Hawk with any EMF-based tool and language, both those developed within MONDO (CloudATL, IncQuery and DSL-tao) and outside MONDO (Epsilon Exeed).
Changes since D5.2

This document contains the following major changes since D5.2 [16] from M12:

- The overview of a Hawk model index in Section 2.2 has been updated: repositories are encoded within the file index and SHA-1 checksums are used to detect changes in model elements.
- The components within Hawk in Section 2.3 have been updated: in addition to new components, the Eclipse UI component has been generalized to a class of “client” components.
- The EMF parser has been moved to Section 3.2.1 to improve readability, and the BPMN parser has been added (Section 3.2.4). The IFC parser in Section 3.2.2 has been extended to add support for IFC4 and the Modelio parser in Section 3.2.3 has been updated to Modelio 3.2.1.
- All model parsers in Section 3 now include details on how they were validated on real-world models. The Modelio and IFC parsers now include more advanced queries displaying some of the advanced capabilities in Hawk.
- The description of the integration between Hawk and the EOL language has been extended with sections on reverse reference navigation (Section 4.2.2), scoped queries (Section 4.2.3) and meta-level queries (Section 4.2.4).
- The section on integrating additional languages has been rewritten and significantly extended, becoming Section 5. This section constitutes deliverable D3.4 [15]. The section now describes an extended version of the EMF Resource interface, two implementations of this interface for local and remote scenarios and several examples of how these implementations were used to integrate Hawk with Epsilon Exeed (from outside MONDO), IncQuery (from WP3), CloudATL (from WP3) and DSL-tao (from WP2).
1 Introduction

Model driven engineering (MDE) promotes the use of detailed models to represent engineering artefacts of interest in an attempt to raise the level of abstraction at all stages of software development. In recent studies, MDE has been shown to increase productivity and significantly enhance important aspects of software engineering development such as consistency, maintainability and traceability. MDE is therefore increasingly applied to larger and more complex systems. However, the current generation of modelling and model management technologies are being stressed to their limits in terms of their capacity to accommodate collaborative development, efficient management and persistence of models larger than a few hundreds of megabytes in size. Thus, recent research has focused on scalability across the MDE technical space to enable MDE to remain relevant and to continue delivering its widely recognised productivity, quality and maintainability benefits.

The MONDO project investigates the scalability issues for MDE. Typically, achieving scalability involves being able to construct large models and domain specific languages in a systematic manner; enabling large teams of modellers to construct and refine large models in a collaborative manner; advancing the state of the art in model querying and transformations tools so that they can cope with large models (of the order millions of model elements); and providing an infrastructure for efficient storage, indexing and retrieval of such models [12]. As part of the MONDO project, this report looks closely into how support for heterogeneous modelling formats can be achieved, in the context of scalable modelling.

Besides the XML Metadata Interchange (XMI) format, there is a wide variety of modelling formats and formalisms. A modelling tool may define a format that suits its requirements and use cases, and include features not supported in standard formats such as XMI. Tools may also be used to capture models that represent physical entities, such as building or products. Such tools may provide limited support for lazy or partial loading of models, features that are essential for managing large models. While there are existing works which attempt to explore alternatives to XMI, the MONDO project proposes a systematic solution for scalable model indexing that integrates with existing model driven development tools, and can be of benefit to a wide range of heterogeneous modelling contexts.

This Heterogeneous Model Management framework developed in Task 5.2 provides fundamental support for several tasks of the project: Task 5.3, (indexing, change monitoring, efficient querying); Task 3.6 (integration with other languages) and Task 4.2 (scalable collaboration). This final version is an updated version of D5.2 [16] that integrates the improvements performed in the framework since then and the new requirements provided by the industrial partners on the interoperability layers that were originally implemented. Section 5 constitutes deliverable D3.4, with reports on two abstractions that expose Hawk indexes as regular models for any EMF-based tool and on several specialized integrations with tools developed inside and outside MONDO that take advantage of the additional features of these abstractions.

2 Heterogeneous Model Management Framework (Hawk)

The aim of the Heterogeneous Model Management Framework (Hawk) developed in this task is to enable developers to perform queries on heterogeneous models stored in established file-based version control systems, without needing to maintain a complete copy of them in their local workspace.
To achieve this, Hawk acts as a middle-man that creates and maintains indexes of models stored in remote file-based version control repositories; a model index is a persisted form of a collection of (potentially interconnected and heterogeneous) models, and its aim is to provide support for efficient querying of these models at a model element granularity. As discussed in [5], this provides an orthogonal approach for addressing the scalability concern that intentionally refrains from interfering with the current state of practice regarding version control.

This section briefly describes the architecture, design, and prototype implementation of Hawk to provide context for how it is used for indexing large models and consequently to efficiently query such model indexes. Hawk aims at delivering a system capable of working with diverse file-based version control systems (VCS) and model persistence formats whilst providing a comprehensive API through which modelling and model management tools can query it. It needs to be scalable so that it can accommodate large sets of models, and non-invasive (the VCS repositories should not need to be modified or configured).

Hawk comprises components which monitor a set of version control systems, parse and index relevant models stored in them. For details on supported version control systems, model formats, index persistence back-ends as well as additional components of Hawk readers can refer to [5].

### 2.1 Architecture

Figure [1] provides an overview of the architecture of Hawk, the proposed framework which aims at delivering the following capabilities:

- Work with diverse file-based version control systems (e.g. SVN, Git) and model persistence formats (e.g. XMI, IFC, Modelio);
- Be scalable so that it can accommodate large sets of models with significant sizes;
- Provide a comprehensive API through which modelling and model management tools can execute queries;
- Provide extensibility mechanisms for accommodating new types of VCSs and model representation formats;
- Be non-invasive: the VCS repositories should not need to be modified or configured in any way.

Hawk can be configured to monitor a set of VCS servers for model-related events (e.g. creation of new models, modification of existing models) and maintain a copy of the latest version of all interesting (based on its configuration) models in these servers in a scalable index, the details of which are discussed in D5.5 [17]. Modelling and model management tools can then perform global queries through Hawk, exploiting its fast, scalable and synchronized model index. With regard to the VCS servers, Hawk acts as a standard read-only client to allow it to be non-invasive.
2.2 Overview of a Hawk model index

Based on results obtained through extensive benchmarking in Task 5.1, published in [6] [22], we have decided to use NoSQL graph databases (e.g. 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 (EObjects 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-

---

\(^1\)This should not to be confused with a database index provided by many SQL and NoSQL databases

\(^2\)http://neo4j.org/

\(^3\)Eclipse Modeling Framework - http://www.eclipse.org/emf
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).

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.

2.3 Components

Figure 3 illustrates the components that comprise Hawk.

- **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 – discussed in Section 3.1).
**Model indexer components**: these components, specific for each back-end used (such as Neo4J [28] 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 [4], 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 [4], 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).

• **Client components**: provides front-ends to Hawk, thus allowing for indexes to be added/removed and have their status checked. They 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). Hawk itself provides an Eclipse-based user interface, and WP6 has embedded Hawk into the MONDO Server component and exposed it through a Thrift API (more details in D6.3 [19]).

• **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.

The design of Hawk aims to provide an extensible model indexing framework. The next sections of this report describe how the indexer provides heterogeneous model parsing and heterogeneous index querying.
Figure 4: UML class diagram with the Java interfaces of key Hawk component types
3 Heterogeneous Model Support

This section will describe how Hawk can support multiple modelling technologies through an abstraction layer, and describe how this layer has been used to integrate various types of models within MONDO. The complete lifecycle of models and metamodels in Hawk (e.g. how the system responds when a model/metamodel is updated or when a metamodel is registered/unregistered) is beyond the scope of this deliverable and is discussed in D5.5 [17].

3.1 Hawk model parsing interfaces

As mentioned in Section 2.3, Hawk can support various model persistence formats through model parser components. These are implemented on top of an abstraction layer that decouples Hawk from the formats used by the models to be indexed.

Figure 5 shows the main elements of this abstraction layer, which is a set of Java interfaces that represent common modelling concepts (e.g. classes, attributes or references). Due to the popularity and expressiveness of the Eclipse Modelling Framework, these interfaces are inspired on EMF concepts such as EClass, EStructuralFeature and so on: Section 3.2.1 provides more details on the mapping. However, the Hawk interfaces are limited to the essential functionality needed by Hawk itself, making it easier to integrate new modelling technologies.

The two key interfaces that need to be implemented to produce a new model parser component are IHawkModelResource and IHawkMetaModelResource (shown in Figure 4), which represent a single model and a single metamodel respectively. Doing so will require implementing these other interfaces as well:

- IModelResourceFactory: produces the appropriate IHawkResource instances from a file.
- IMetaModelResourceFactory: produces the appropriate IHawkMetaModelResource from a file, and also contributes “predefined” metamodels (e.g. the BPMN metamodels for the BPMN parser).
- IHawkAttribute: an attribute of a IHawkClass, whose type is a IHawkDataType.
- IHawkAnnotation: an annotation on a model element.
- IHawkClass: a class of objects (e.g. “Person”).
- IHawkClassifier: either a IHawkClass or a IHawkDataType.
Once these interfaces are implemented, the resulting component can be integrated into Hawk as a library (when using Hawk outside the Eclipse IDE) or as an plugin (when using Hawk within the Eclipse IDE). Hawk provides two Eclipse extension points that accept plugin contributions: org.hawk.core.ModelExtensionPoint (for the IModelResourceFactory implementation) and org.hawk.core.MetaModelExtensionPoint (for the IMetaModelResourceFactory implementation).

3.2 Hawk model parsing components

In this section, the model parsing components that enable Hawk to process generic EMF-based models, IFC models, Modelio UML models and BPMN models are presented in detail.

3.2.1 Eclipse Modelling Framework

The EMF parser implements the required interfaces to allow EMF-based models serialised as XMI documents to be indexed by Hawk: its IHawkMetaModelResource implementation converts Ecore
//create a ResourceSet to keeps track resources and resource loaders
ResourceSet resourceSet = new ResourceSetImpl();

//.xmi files are read by XMIResourceFactoryImpl
resourceSet.getResourceFactoryRegistry()
   .getExtensionToFactoryMap().put("xmi",new XMIResourceFactoryImpl());

//load the file into an EMF resource
Resource resource = resourceSet.createResource(
   URI.createFileURI("models/library.xmi"));
resource.load(null);

Listing 1: Code to read XMI models into EMF Resources.

metamodels to instances of IHawkMetaModelResource, and its IModelResourceFactory implement-
ation converts models to instances of IHawkModelResource. An outline of the mapping between the
Ecore classes and the Hawk interfaces is given in Figure 6: mappings are represented as grey arrows.
Before indexing an EMF-based model, Hawk needs to be aware of the metamodel it conforms to.
To provide fine-grained control to the end user and to avoid unintentional conflicts, Hawk does not
attempt to discover metamodels automatically in the repositories it monitors. Instead, users explicitly
register metamodels through the registerMetamodel method of the IModelIndexer interface (listed in
Figure 4).

Implementation

These classes implement the key interfaces of the abstraction layer:

- EMFModelResource exposes a single EMF model (Resource).
- EMFModelResourceFactory parses XMI files into EMFModelResources.
- EMFMetaModelResource exposes a single EMF metamodel (EPackage).
- EMFMetaModelResourceFactory parses .ecore files into EMFMetaModelResources.

A simplified version of the code used in EMFModelResourceFactory to parse EMF models is shown
in Listing 1

Validation

The validation of the EMF parser has been done by indexing the GraBaTs 2009 set*.xmi sample
models while having the same IGraphChangeListener that was used for the BPMN MIWG validation
process, as mentioned in Section 3.2.4.

3.2.2 Building Information Modelling - IFC

EXPRESS is part of the Standard for the Exchange of Product Model Data (STEP) and is defined
in the ISO standard ‘Industrial automation systems and integration - Product data representation and
exchange’ (ISO-10303) [20]. EXPRESS is used for the representation and interchange of physical product models in the architecture, engineering and construction industries. STEP is a broad specification and EXPRESS forms one component of the standard; part 11 defines the abstract syntax, part 21 defines a concrete ASCII-based ‘STEP’ syntax and part 28 defines a standard concrete XML-based syntax for the EXPRESS language [13].

EXPRESS is intended to be a generic format; it specifies an information domain in terms of entities, with classes of objects sharing common properties, which are represented by associated attributes and constraints. In EXPRESS, constraints are written using a mixture of declarative and imperative language elements. EXPRESS is used extensively in Computer Aided Design (CAD) tools such as Autodesk Inventor, PTC Creo and Siemens NX, as a common format to interchange models of physical products. EXPRESS-G is a formal graphical notation of EXPRESS which can visually represent the static components of an EXPRESS model including entities, attributes, type declarations, and hierarchies of inheritance.

EXPRESS is also used to represent and exchange detailed architectural plans of buildings, via the Industry Foundation Classes (IFC) standard (ISO 16739) [10]. The IFC standard defines a common format to represent and exchange models of buildings during design and development. Building Information Model (BIM) servers are used to enable collaboration between building stakeholders throughout the buildings life cycle - e.g. architects, engineers, financial planners and customers. Tool such as Autodesk Revit and Bentley Systems use the IFC format for interchange and collaboration. However, handling large and complex models of buildings is not the norm, due to poor performance, and online collaboration is not possible.

Implementation

The Hawk IFC model parser is based on the BIMserver [27] 1.4.0 libraries. It reuses the BIMserver IFC-XML and IFC-STEP parsers to create a Java object graph for a given file, which is then adapted to the interfaces in [31]. IFC models must conform to one of the fixed IFC2x3 and IFC4 metamodels as implemented by the BIMserver libraries, which are provided automatically to Hawk on startup.

The classes of interest are:

- **IFCModelResource** represents an IFC model.
- **IFCModelFactory** parses an IFC model into an **IFCModelResource** using the BIMserver libraries.
- **IFCMetaModelResource** represents one of the IFC fixed metamodels.
- **IFCMetaModelResourceFactory** produces the two instances of **IFCMetaModelResource**: one for the IFC2x3 metamodel and another for the IFC4 metamodel. The metamodels are created from the schemata in EXPRESS format (IFC2X3_TC1.exp and IFC4.exp) via the Express2EMF component of the BIM server.

The code fragment in Listing 2 demonstrates how an IFC-STEP file conforming to the IFC2x3 metamodel is parsed into a Java object graph using the BIMserver libraries. Once a model has been read, all elements of the IFC model are wrapped by classes in the package *org.hawk.ifc* for use within Hawk.
After sampling the beginning of the file, create the appropriate Deserializer subclass and select the appropriate EMF package.

```java
Deserializer d = new IfcStepDeserializer();
String packageName = Ifc2x3tc1Package.eINSTANCE.getName().toLowerCase();

// Load the schema from the BIMserver.
final PluginManager bimPluginManager = createPluginManager();
final MetaDataManager bimMetaDataManager = new MetaDataManager(bimPluginManager);
bimMetaDataManager.init();
final PackageMetaData packageMetaData = bimMetaDataManager.getPackageMetaData(packageLowerCaseName);

// Initialise the schema for the deserializer.
d.init(packageMetaData);

// Load the model from the file (may need to be unzipped first).
IfcModelInterface models = d.read(new File("samples/example.ifc"));

// Build the internal set of elements, taking care to handle "floating" IDeOjects that are not reported by getValues(). These normally correspond to IFC entities that are nested inside others.
```  

Listing 2: Simplified excerpt of the code used to parse IFC files.

One important detail is learning in advance whether the file to be parsed conforms to IFC2x3 or to IFC4, and whether the file uses IFC-STEP (a concise line-based textual format) or IFC-XML. The file extension cannot be relied on for this, as .ifc may be used in either case. Practitioners also tend to share IFC models in zipped form, especially the highly compressible IFC-STEP format, so Hawk needs to be able to consume those files directly: in those cases, .ifc.zip of .ifczip is normally used, but cannot be relied upon either. Instead, the IFC model parser samples the files to determine their type from their contents. The detailed process is shown as an appendix in page 44. Files with unknown types are ignored.

**Validation**

The parser has been validated on a collection of IFC models in various formats provided by UNINOVA, ranging from small models (several kilobytes) to large ones (several hundreds of megabytes, with millions of model elements). These models helped find and correct several latent issues in the
var doors = 'http://buildingsmart.ifc2x3tc1.ecore::IfcDoor'.all;

-- Produces a sequence of maps with information about each door
var results : Sequence;
for (door in doors) {
    var storey = getDoorStorey(door);
    var style = getDoorStyle(door);
    var opening = getDoorOpening(door);
    var placement = door.ObjectPlacement;
    var repr = door.Representation;

    results.add(Map{
        'Door' = door.getTypeName() + ' ' + door.getId(),
        'Name' = door.Name,
        'OverallHeight' = door.OverallHeight,
        'OverallWidth' = door.OverallWidth,
        'Placement' = placement.getTypeName() + ' ' + placement.getId(),
        'Representation' = repr.getTypeName() + ' ' + repr.getId(),
        'Style' = style.getTypeName() + ' ' + style.Name + ' ' + style.getId(),
        'Opening' = opening.getTypeName() + ' ' + opening.Name + ' ' + opening.getId(),
        'Storey' = storey.getTypeName() + ' ' + storey.Name + ' ' + storey.getId()
    });
}

return results;

operation getDoorStorey(door) {
    return door
        .revRefNav_RelatedElements
        .revRefNav_ContainsElements
        .flatten.first;
}

operation getDoorStyle(door) {
    return door.IsDefinedBy.RelatingType.flatten.first;
}

operation getDoorOpening(door) {
    return door.FillsVoids.RelatingOpeningElement.flatten.first;
}

Listing 3: EOL query for a detailed listing of all the doors in an IFC model
-- Lists all doors by going from the type node directly to the instances
var doors = 'http://buildingsmart.ifc2x3tc1.ecore::IfcDoor'.all;

-- Produces a name → height → width → storey → count map
var results : Map;
for (door in doors) {
  var storey = getDoorStorey(door);
  results.getOrPutEmptyMap(door.Name)
  .getOrPutEmptyMap(door.OverallHeight)
  .getOrPutEmptyMap(door.OverallWidth)
  .increaseOrSet(storey.Name);
}
return results;

-- Extends Java maps: safe get that puts an empty map if the key is not present
operation Map getOrPutEmptyMap(key) {
  var value = self.get(key);
  if (not value.isDefined()) {
    value = Map {};
    self.put(key, value);
  }
  return value;
}

-- Extends Java maps: safe get+increment that puts 1 if the key is not present
operation Map increaseOrSet(key) {
  var value = self.get(key);
  if (not value.isDefined()) {
    value = 1;
  } else {
    value = value + 1;
  }
  self.put(key, value);
  return value;
}

operation getDoorStorey(door) {
  return door
  .revRefNav_RelatedElements
  .revRefNav_ContainsElements
  .flatten.first;
}

Listing 4: EOL query for counting doors by story, type, height and width in an IFC model
original parser, such as the common use of compression by practitioners or the existence of “floating” instances, as mentioned in Listing 2.

For one particular IFC model (Munkerud.ifc), with 573,887 model elements, UNINOVA provided an Excel spreadsheet produced by one of their analysis tools as an example of several common queries they need to perform on their IFC models. This model was indexed by Hawk and several example queries that concisely reproduced some of the functionalities of their analysis tool were developed:

- Simple queries for counting all the instances of a specific type, in the form of “return Type.all.size;”. These are needed for the bill of materials of a building, e.g. to know how many doors are in the building.
- Listing 3 lists all the doors in the model and their characteristics. The query takes advantage of the index-free element adjacency of the graph-based persistence layer, operating in time proportional not to the size of the entire model, but only to the total number of doors in the building.
- Listing 4 produces numbers of doors by type, height and width for each story in the building. Again, its running time depends only on the number of doors in the building and not the size of the entire model.

### 3.2.3 Modelio UML models

Modelio is an open source modelling environment that supports modelling standards such as UML and BPMN. There are freely available open source “community” editions and paid-for closed source editions of Modelio, with different features and levels of support. Modelio is based on the Eclipse Rich Client Platform and has a range of software development features that complement modelling, such as model transformation, code-to-model reverse engineering, code generation or round-trip engineering.

Modelio uses its own file format for persisting models. The file format may be updated with each new revision of the tool: additional plugins are provided to migrate older formats to the new versions. Internally, a project file is a .zip file arranged into subfolders and files. An XML project.conf file describes project metadata. Models are split into fragments, conforming to a custom XML schema. The plugin dependencies of a project are included in the project archive as .jmdac Modelio MDA component files. Modelio also supports importing and exporting models into the standard UML and EMF XMI formats.

### Implementation

The Modelio parser embeds a modified version of Modelio 3.2.1 that excludes the components not needed for MONDO and makes some internal APIs available for reuse. These internal APIs are used to load the custom Modelio project format into an object graph, rooted by a single Modelio GProject object which contains a set of packages. These packages are then exported to XMI format and parsed using EMF facilities.

---

[http://github.com/mondo-project/hawk-modelio]
// Load the internal Modelio Metamodel
MetamodelLoader.Load();

// Read the model descriptor
ProjectDescriptor descriptor =
    new ProjectDescriptorReader().read(resolve, DefinitionScope.LOCAL);

// Obtain the IProjectService from Modelio
Modelio modelio = Modelio.getInstance();
IEclipseContext modelioContext = /* ... */;
IProjectService projectService = modelioContext.get(IProjectService.class);

// Open the project and obtain the GProject instance
projectService.openProject(descriptor, new NoneAuthData(),
    new NullProgressMonitor());
GProject project = projectService.getOpenedProject();

// Obtain all the packages in the model
final List<Package> pkgs = new ArrayList<>();
for (IProjectFragment fragment : project.getOwnFragments()) {
    for (MObject mObj : fragment.getRoots()) {
        if (mObj instanceof Package) {
            pkgs.add((Package) mObj);
        } else if (mObj instanceof Project) {
            pkgs.add(((Project) mObj).getModel());
        }
    }
}

// Export packages into XMI for import into Hawk,
// using the Modelio ExportService
final List<Model> models = new ArrayList<>();
for (Package pkg : pkgs) {
    GenerationProperties genProp = GenerationProperties.getInstance();
    genProp.initialize(new MModelServices(project));
    genProp.setSelectedPackage(pkg);
    ExportServices exportService = new ExportServices(null);
    Model m = exportService.createEcoreModel(pkg, null);
    models.add(m);
}

// Close the project once done
projectService.closeProject(project);

Listing 5: Code to read Modelio projects outside the Modelio development environment.
The classes of interest are:

- *ModelioModelResource* implements *IHawkModelResource* and represents a Modelio model.
- *ModelioModelResourceFactory* parses a Modelio project into a *ModelioModelResource*.
- *ModelioMetaModelResource* represents the fixed Modelio UML metamodel.
- *ModelioMetaModelResourceFactory* provides the fixed Modelio UML metamodel.

The code in Listing 5 is a simplified excerpt of the code used in the class *ModelioModelResourceFactory* to parse Modelio projects, after unpacking their .modelio.zip representations to a temporary directory. Lines 1–16 perform necessary preparations and load the Modelio project into memory, using the standard Modelio services that are injected through E4. Lines 19–28 collect all the root packages in the Modelio project into a list: a single Modelio project can have several root packages (e.g. one defined by the user and another imported through a .ramc model library file). Lines 32–41 reuse the Modelio *ExportServices* to export the contents of these packages into EMF/UML2 3.2.0 objects that are ready to be adapted to the Hawk API.

**Validation**

In order to validate the Modelio parser, SOFTEAM (the developers of the Modelio tool) provided a collection of Modelio 3.2.1 projects. Hawk was able to index the parts of the projects that conformed to the UML2 metamodel, excluding elements from other metamodels (e.g. TOGAF or the JavaDesigner-EE metamodel from the closed-source edition).

Some of these models were already made publicly available by SOFTEAM as part of their [http://opensource-uml.org](http://opensource-uml.org) website, which performs reverse engineering on well-known open source projects and extracts UML models from them, helping the community in understanding their architectures. In particular, one of the indexed models contained reverse-engineered UML models based on the sources of the OpenJDK, Eclipse Rich Platform, JUnit, Apache Ant and jFreeChart projects, totaling $x$ model elements.

Based on these models, several queries were developed for some common questions on Modelio models. Some of these are shown in Listings 6, 7, 8 and 9. In particular, 8 and 9 take advantage of the fact that derived attributes are also indexed by the persistence layer, speeding up the filtering of classes by their number of operations. The details behind derived attributes are described in further depth in D5.5 [17].

These queries have been added to the public Hawk wiki and they were presented at the Paris Open Source Summit in November 2015. To make it easier for first-time users, the queries in the wiki use a toy model already exported into XMI instead of the previously mentioned .modelio.zip files.

**Listing 6: All superclasses of a UML class**

```java
return Class.all
    .select(c|c.qualifiedName='package::Class')
    .superClass.flatten.name;
```

[https://www.youtube.com/watch?v=EjJG281b_MA](https://www.youtube.com/watch?v=EjJG281b_MA)
Listing 7: All subclasses of a UML class. Uses reverse reference navigation (see D5.5 [17]).

Listing 8: All classes with at least one operation. Uses a derived attribute, as described in D5.5 [17].

Listing 9: Sample query on Modelio models producing a list of pairs (> x, y), where each pair indicates that y classes had more than x operations.
3.2.4 Business Process Model and Notation

The OMG Business Process Model and Notation 2.0 specification [14] (BPMN 2.0) is a standard modelling notation for describing business processes as workflows of concurrent activities that exchange messages with each other through various types of links and gateways.

BPMN is a mature and widely used specification and is being used both for process analysis and reengineering and for process automation. BPMN is implemented by vendors such as IBM, Oracle, SAP, Intalio or Bonita. The Eclipse BPMN2 Modeller project provides a development environment for BPMN models, including a parser, editor and several validation tools [9].

The BPMN file format is based on XML and is described through several XML Schema documents. The BPMN2 Modeller project uses these XML Schema documents to generate an Ecore metamodel, and has a specialized parser to deal with some of the particularities of the BPMN file format, such as the ability to combine the process description and its graphical representation in the same file.

Implementation

The BPMN model parsing component reuses the specialized parser and the metamodels from the Eclipse BPMN2 Modeller project. .bpmn and .bpmn2 files are parsed into object graphs, which are then adapted to the interfaces in Section 3.1 in a similar way to that used for EMF in Section 3.2.1.

The classes of interest are:

- `BPMNModelResource` represents a BPMN model.
- `BPMNModelFactory` parses a BPMN model into an `BPMNModelResource`.
- `BPMNMetaModelResource` represents one of the BPMN fixed metamodels.
- `BPMNMetaModelResourceFactory` produces the various BPMN metamodels, listed below.

The code in Listing 10 shows how a .bpmn or .bpmn2 file is loaded by the Hawk parser and wrapped into a `IHawkModelResource` implementation. Lines 1–5 create the required EMF `ResourceSet` and register the specialized BPMN parser into its configuration. The standard EMF API for loading models is then used in lines 7–8, and the resulting `Resource` is wrapped into the Hawk `BPMNModelResource` adapter class in line 10.

```
ResourceSet resourceSet = new ResourceSetImpl();
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap()
    .put("bpmn", new Bpmn2ResourceFactoryImpl());
resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap()
    .put("bpmn2", new Bpmn2ResourceFactoryImpl());
Resource r = resourceSet.createResource(URI.createFileURI(f.getPath()));
r.load(null);
IHawkModelResource resource = new BPMNModelResource(r, this);
```

Listing 10: Code to read BPMN models using the BPMN2 Modeller parser.
The BPMN model parsing component registers automatically upon startup the following metamodels provided by the BPMN2 Modeller and Ecore projects, if they are not available already:


**Validation**

The BPMN parser in Hawk has been used on the models in the test suite designed by the BPMN Model Interchange Working Group (BPMN MIWG). In its latest revision (9e461a as of November 26, 2015), this test suite contains 353 BPMN models produced by 29 different BPMN-based tools.

A series of commits from the BPMN MIWG specification were selected for their relevance and were processed in sequence for Hawk. A specially-designed `IGraphChangeListener` that checked that the contents of the Hawk index matched the contents of each file in each commit was installed during this process and reported positive results.

More details are available in deliverable D5.5 [17].

**4 Index Querying Using Epsilon**

This section introduces Epsilon and its Model Connectivity Layer (EMC) and presents implementation details of Hawk’s query layer integration with Epsilon. This allows queries expressed in the Epsilon Object Language (EOL) to be executed over a Hawk index.

**4.1 Epsilon**

The Epsilon platform [21] is an extensible family of languages for common model management tasks and includes tailored languages for tasks such as model-to-text transformation (EGL), model-to-model transformation (ETL), model re-factoring (EWL), comparison (ECL), validation (EVL), migration (Flock), merging (EML) and pattern matching (EPL). All task-specific languages in Epsilon build on top of a core expression language – the Epsilon Object Language (EOL) – to eliminate duplication and enhance consistency.

As seen in Figure 7, EOL – and as such all languages that build on top of it – is not bound to a particular metamodeling architecture or model persistence technology. Instead, an intermediate layer – the Epsilon Model Connectivity layer – was introduced to allow for seamless integration of any modeling back-end.

This layer of Epsilon uses a driver-based approach where integration with a particular modeling technology is achieved by implementing a **driver** that conforms to a Java interface (`IModel`) provided by EMC. For a more detailed discussion on EMC and the `IModel` interface, the reader can refer to Chapter 3 of [11].

### 4.2 Querying a Hawk Model Index Using the Epsilon Object Language

In D5.5 [17], the implementation of the important methods needed by an EMC driver to enable integration with Epsilon is summarised, as well as that of the derived attributes used by Hawk’s driver to improve its query performance.

#### 4.2.1 IModel interface method implementations

In order to use Epsilon’s EOL to query model indexes stored in Hawk, an implementation of the `IModel` interface is required. In Table 1 of [7] we present a description of various methods of interest in the `IModel` interface and a summary of their implementation details in Hawk.
Table 1: Methods suitable for Hawk-specific optimizations in the `IModel` interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allContents()</td>
<td>Collection&lt;??&gt;</td>
<td>Returns a collection containing all of the nodes contained in the index in the form of <code>GraphNodeWrappers</code></td>
</tr>
<tr>
<td>hasType(String type)</td>
<td>boolean</td>
<td>Returns whether the type <code>type</code> exists in the index by trying to find it through the Metamodel index of the store.</td>
</tr>
<tr>
<td>getAllOfType(String type)</td>
<td>Collection&lt;??&gt;</td>
<td>Returns a collection containing all of the objects of type <code>type</code> in the index by first invoking hasType(type) and, if successful, finding the type using the Metamodel index and then creating a collection of <code>GraphNodeWrappers</code> containing every element which has an ofType relationship to <code>type</code>.</td>
</tr>
<tr>
<td>getTypeOf(Object instance)</td>
<td>Object</td>
<td>Returns the type node of the element <code>instance</code> in the index by directly accessing the node <code>instance</code> (as this method is always passed a <code>GraphNodeWrapper</code> as the <code>instance</code>) and navigating its ofType relationship to get the type node. The returned object is a <code>GraphNodeWrapper</code>.</td>
</tr>
<tr>
<td>isOfType(Object instance, String type)</td>
<td>boolean</td>
<td>Returns whether the node <code>instance</code> in this model is of type <code>type</code> by first invoking hasType(type) and, if successful, invoking getTypeOf(instance) and performing a String comparison on the resulting names.</td>
</tr>
<tr>
<td>knowsAboutProperty(Object instance, String property)</td>
<td>boolean</td>
<td>Returns whether the element <code>instance</code> in this index can have the structural feature <code>property</code> by first invoking getTypeOf(instance) and then invoking the EMF method <code>getEStructuralFeature(type, property)</code>.</td>
</tr>
</tbody>
</table>

Note that any model element loaded into memory is an instance of the `GraphNodeWrapper` Java class. This is a lightweight object which contains only the unique identifier of the relevant model element in the store (the node ID for instance in a Neo4J NoSQL Graph database) as well as a reference to the Epsilon model it is part of. This object can be used to load the element’s attributes and relationships on demand. As discussed above, in the current prototype, we use EOL expressions to describe the derived attributes to be computed. Such EOL expressions are actually executed after the model insertion has been completed, using Hawk’s EMC driver to query the database.

### 4.2.2 Reverse reference navigation

In the spirit of EMF’s `eContainer` method which allows an `EObject` to get access to its container object, Hawk provides a mechanism for navigating a reference in reverse, i.e. from its target to its source. This feature can be used by prefixing the relevant reference with “`revRefNav_`”. For instance,
if “a” is a UML Class object, \(a\).\textit{superClass} would obtain its superclass and \(a\).\textit{revRefNav}\_\textit{superClass} would obtain its subclasses.

It is also possible to directly refer to \(a\).\textit{eContainer} to obtain the container of any model element. To implement this method, \textit{IHawkReference} was extended with a \textit{isContainment} reference that would allow Hawk to know if a reference between model elements had containment semantics. This flag is then used as an edge property in the underlying graph.

### 4.2.3 Scoped queries

Hawk supports the use of scoped queries with respect to their file of origin. Users can limit the query to the elements in a specific file or set of files by providing the relevant file inclusion pattern to Hawk. This allows Hawk, which is normally capable of returning global queries of the entire model, to limit its search to elements contained in only a specific subset of its contents. The syntax used is that of a comma-separated list of filename patterns to be included (where “\*” means “0+ arbitrary characters”): if empty, all files will be considered.

This allows virtual local queries to be performed, as if the rest of the model was not present. For example:

- On specific files: \textit{model1.xmi, model2.xmi}.
- On specific file types: \textit{*.xmi} or \textit{class\_diagram\_*\_uml}.
- On specific directories: \textit{/london\_bridge/schematics/\_*\_xmi}.

This filtering can also be performed at the repository level in a similar manner: only the files in the matched repositories will be considered for the query.

### 4.2.4 Meta-level queries

Hawk’s EMC driver also exposes the registered metamodels to the Epsilon Object Language, making it possible to perform queries on the available types and not only on the instances. Type nodes are wrapped into \textit{TypeNodeWrappers} and metamodel nodes are wrapped into \textit{MetamodelNodeWrappers}.

These expressions can be used as starting points for a meta-level query:

- \textit{Model.types} lists all the types registered in Hawk (EClass instances for EMF).
- \textit{Model.metamodels} lists all the metamodels registered in Hawk (EPackage instances for EMF).
- \textit{Model.getTypeOf(obj)} retrieves the type of the object \textit{obj}.

For a metamodel \textit{mm}, these attributes are available:

- \textit{mm.uri} is the namespace URI of the metamodel.
- \textit{mm.metamodelType} is the type of metamodel that was registered.
- \textit{mm.dependencies} lists the metamodels this metamodel depends on (usually at least the Ecore metamodel for EMF-based metamodels).
- \textit{mm.types} lists the types defined in this metamodel.
IfcWindow.all.select( window | window.OverallHeight >= 2.5 and window.OverallWidth >= 2.5).size().println();

Listing 11: Index Query for an IFC2x3 model.

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.collect(names|names.returnType.name.fullyQualifiedName);

Listing 12: Sample Index Query for a JDTAST (EMF) model.

- `mm.resource` retrieves the original string representation for this metamodel (the original `.ecore` file for EMF).

For a type `t`, these attributes are available:

- `t.metamodel` retrieves the metamodel that defines the type.
- `t.all` retrieves all instances of that type efficiently (includes subtypes).
- `t.name` retrieves the name of the type.
- `t.attributes` lists the attributes of the type, as slots (see below).
- `t.references` lists the references of the type, as slots.
- `t.features` lists the attributes and references of the type.

For a slot `sl`, these attributes are available:

- `sl.name`: name of the slot.
- `sl.type`: type of the value of the slot.
- `sl.isMany`: true if this is a multi-valued slot.
- `sl.isOrdered`: true if the values should follow some order.
- `sl.isAttribute`: true if this is an attribute slot.
- `sl.isReference`: true if this is a reference slot.
- `sl.isUnique`: true if the value for this slot should be unique within its model.

4.3 Sample Index Queries

Listing [11] shows a query over IFC2x3 models which prints the number of Windows (`IfcWindows`) in all indexed building models whose `OverallHeight` and `OverallWidth` are more than 2.5.

Listing [12] shows a query that uses the Java (JDTAST) metamodel used in the Program Comprehension case study of the GraBaTs 2009 contest [3]. In this metamodel, there are `TypeDeclaration` 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. This query can be used to find singleton classes (ones which declare a public and static method with return type the class itself) in the indexed models.

Additional sample queries have been listed in the parts dedicated to validation of Sections 3.2.2 and 3.2.3.

## 5 Integration of Additional Languages (D3.4)

As part of Work Package 3 (WP3), various other languages had to be integrated with Hawk. These languages included the IncQuery \[25\] incremental query language and two specializations of the ATL model transformation language \[8\]: CloudATL and ReactiveATL.

Since these languages were based on EMF, it was decided to develop an abstraction layer that would present Hawk indexes as read-only EMF-based models, which could be used as inputs for these tools without requiring any modifications. Later investigation suggested developing not one, but two such abstraction layers: one for Hawk indexes running in the same Java virtual machine as the EMF-based tool, and one for Hawk indexes running in a different VM or even a different computer.

Once these abstractions were developed, any EMF-based tool (including those developed within MONDO) could read Hawk indexes as models without any modifications. After this, several EMF-based tools and languages were modified to integrate more deeply with the advanced capabilities introduced by Hawk:

- The Epsilon Exeed tree-based model editor was specialized to allow browsing through large models efficiently.
- CloudATL was extended with the ability to consume Hawk indexes from the Hadoop jobs it launched.
- IncQuery was extended with an additional module that would enable it to query models without having to load them fully in memory.
- Beyond the original integration scenarios envisioned for WP3, a future integration scenario was enabled between the scalable notations for WP2 and Hawk, in which the scalable editors could list all the candidates for a reference without having to load all model fragments.

The rest of this section will provide more details on these various actions, starting with the local and remote EMF resource abstractions and continuing with the tool-specific integration scenarios.

### 5.1 Common interfaces for extended EMF resources

As mentioned above, there are two integration levels envisioned between Hawk and an EMF-based tool: one in which the tool sees the Hawk index as a standard *Resource* and does not take advantage of all its features, and one in which it takes into account its advanced features and has access to more functionality beyond the *Resource* Java interface.
In order to support this advanced level of integration, Hawk extends Resource with the new HawkResource interface, which is then implemented by the local and remote EMF resource abstractions in Sections 5.2 and 5.3. The new methods become practical with the graph-oriented storage solutions used by Hawk, and can dramatically speed up certain common operations in model-driven engineering contexts. These methods are:

- `hasChildren(EObject)` returns `true` if the underlying model element has children elements, or `false` otherwise.
- `fetchAttributes(Map<String, EObject>)` retrieves attributes for a set of model elements, if they have not been fetched yet.
- `fetchNode(HawkResource, String, boolean)` retrieves a model element by file and URI fragment, and may retrieve its attributes optionally as well.
- `fetchNode(String, boolean)` retrieves a model element by graph node ID, and may retrieve its attributes optionally as well.
- `fetchNodes(List<String>, boolean)` retrieves a list of nodes by graph node IDs, and may retrieve their attributes optionally as well.
- `fetchNodes(EClass, boolean)` retrieves all the instances of the provided type, and may retrieve their attributes optionally as well.
- `fetchValuesByEClassifier(EClassifier)` returns all the values of a certain data type in the model, including duplicates.
- `fetchValuesByEStructuralFeature(EStructuralFeature)` returns all the values that a certain feature within a certain type takes in the model, including duplicates.
- `fetchTypesWithEClassifier(EClassifier)` returns a (type, feature) mapping from all the types that contain a feature of the specified data type to these features.
- `getEObjectNodeID(EObject)` returns the graph node ID of a model element.
- `getRegisteredMetamodels()` returns a list with the URIs of the registered metamodels.
- `getRegisteredTypes(String)` returns a list with the names of the registered types.
- `addSyncEndListener(Runnable)` registers a new listener that will be invoked when a Hawk synchronization process has been completed. This allows certain computations to be done every time the contents of a Hawk index changes.
- `removeSyncEndListener(Runnable)` unregisters one of the previous listeners.
- `addChangeListener(HawkResourceChangeListener)` is an optional method that registers a listener that is invoked every time the HawkResource is updated with changes Hawk detects in the indexed models.
- `removeChangeListener(HawkResourceChangeListener)` is an optional method that registers one of those listeners.

Another common interface for all HawkResources is HawkResourceChangeListener (listed above). It allows third-party tools to register their interest in model changes detected by Hawk. This is especially necessary when combining the lazy loading and live update features of the Hawk EMF resource abstractions: in such a situation, the standard EMF notifications used to detect changes cannot tell apart lazy loads from live updates. In contrast, the HawkResourceChangeListener methods are only invoked for live updates. These methods are:

- `featureInserted(EObject, EStructuralFeature, Object)` is called when a new value for a feature was added to a model element.
• `featureDeleted(EObject, EStructuralFeature, Object)` is called when a value for a feature is removed from a model element.
• `instanceInserted(EClass, EObject)` is called when an instance of a certain type is added.
• `instanceDeleted(EClass, EObject)` is called when an instance of a certain type is removed.
• `dataTypeInserted(EClassifier, Object)` is called when a value of a certain data type is added.
• `dataTypeDeleted(EClassifier, Object)` is called when a value of a certain data type is deleted.

These two interfaces have superseded the Mondix approach proposed in D5.2 [16], as they achieve the same goal with a simpler approach. Users of this interface are encouraged to extend from the adapter class `HawkChangeAdapter` instead, which provides default empty implementations for all methods.

5.2 Hawk EMF resource abstraction for local indexes

The `LocalHawkResourceImpl` class in the `org.hawk.emfresource` plugin provides an implementation of `HawkResource` that exposes a Hawk index running in the same Java virtual machine as an EMF model.

File format

To use this EMF resource, users create a `.localhawkmodel` file pointing to the appropriate Hawk instance through an Eclipse wizard, and then open the file normally through the Eclipse “Package Explorer” or any of the usual means for an EMF model. The default editor registered for these `.localhawkmodel` files is the specialized version of Epsilon Exeed mentioned in Section 5.4.

In the current version, `.localhawkmodel` files follow a very simple format: they consist of a single line of text with the name of the Hawk instance to be exposed. The remote EMF resource abstraction in Section 5.3 has a more advanced file format which allows for setting a number of options.

Initial load

Once opened, the resource accesses the graph and retrieves only the nodes of the root model elements, which are not contained by any other model elements. These root elements can be retrieved immediately from a special-purpose index maintained by Hawk. This allows the resource to be opened more quickly than a traditional XMI-based resource, as most models only have one or a very few number of root elements.

Element loading

Model element nodes are converted into EMF `EObject`s through these steps:
1. The instance is created through the registered \textit{EFactory} for the \textit{EClass} referenced by metamodel URI and type name in the node.

Using an \textit{EFactory} instead of creating an \textit{EObject} and setting its \textit{EClass} is a must when using static metamodels, i.e. those that are implemented through Java code and not simply read from an .ecore file. Static metamodels normally assume that the \textit{EObject}s are of specific Java classes, to improve efficiency and simplify their implementation.

2. \textit{EAttributes} are set through the \texttt{AttributeUtils} utility class. If needed, this class can use \textit{EFactory} instances as well, for the same reasons as before.

3. \textit{EReferences} are treated differently, in order to ensure that additional model elements are only loaded when the user requests them. The bytecode of the \textit{EObject}s created by the \textit{EFactory} is instrumented using the Apache-licensed CGLIB library to intercept invocations of the \texttt{eGet} methods of the EMF API.\footnote{An early approach using EMF proxies was discarded as it did not support on-demand loading of attributes, and another early approach using the EMF \textit{EStore} API was discarded as it did not support static metamodels. On-demand attribute loading can significantly reduce the bandwidth needed for the initial load of certain models (Section 5.4).} These objects are created by a \texttt{LazyEObjectFactory}, while using a \texttt{BridgeClassLoader} that gives CGLIB access to the Java class of the \textit{EObject}.

The intercepted calls go through the \texttt{LazyResolver} class, which recovers the requested model elements from the graph. \texttt{LocalHawkResourceImpl} keeps track of pending lazy references through the \texttt{LazyResolver}, and ensures that model elements are only lazily loaded once.

\textbf{Containing files}

The created \textit{EObject} instances are then placed within a surrogate \texttt{HawkFileResourceImpl} resource with the URL of the indexed file. \texttt{HawkFileResourceImpl} instances maintain a bidirectional mapping between graph node IDs and EMF URI fragments, in order to support the EMF get\texttt{EObject(String)} method that requests a model element by URI fragment.

This is done in order to preserve the original object URLs as much as possible, enabling the future envisioned integration with DSL-tao (Section 5.7) and allowing users to easily see the containing file for any root model elements. These surrogate resources are created on demand in the same \texttt{ResourceSet} as the \texttt{LocalHawkResourceImpl} instance. Except for the bidirectional mapping mentioned above, most of the methods required by the \texttt{HawkResource} interface simply delegate on the main resource.

\textbf{Live updates}

In addition to loading and navigating through the model, some EMF-based tools also need to receive notifications of any changes that occur in the model. Some examples include the IncQuery incremental query language and the ReactiveATL transformation tool developed in WP3, or the Epsilon Exeed model editor that was extended in WP5.

Both of the Hawk EMF resource abstractions can produce these notifications. In the case of \texttt{LocalHawkResourceImpl}, this is done by installing the \texttt{LocalHawkResourceUpdater} implementation of \texttt{IChangeListener} into the \texttt{IModelIndexer} (see Figure 4 in page 9).
The model indexer will then notify the resource of any changes that happen in the graph, and the resource will update its own contents automatically. By doing these updates through the standard EMF APIs, the changes will be transparently notified to any EMF Adapters, such as those installed by IncQuery.

One important detail is that IGraphChangeListener implementations are assumed to provide their own transactional semantics, as a model synchronization operation may fail and may have to be rolled back. LocalHawkResourceUpdater keeps track of the new root elements and the loaded elements that have been updated or deleted in the current transaction and only propagates those changes when the synchronization operation is committed. The impact on memory and CPU usage is minimal, as it only requires storing a few pointers and performing some extra method calls during index updates.

### 5.3 Hawk EMF resource abstraction for remote indexes

The HawkResourceImpl class in the uk.ac.york.mondo.integration.hawk.emf plugin provides an implementation of HawkResource exposing a Hawk index running within the MONDO Server product described in D6.3 [19]. The index is accessed through the Apache Thrift-based API of the server (also described in D6.3). This resource abstraction provides many options for controlling the frequency and size of the messages to be exchanged.

#### File format and URLs

These resources can be created by opening a .hawkmodel file or a hawk+http(s):// URL through the standard Eclipse UI or EMF APIs. .hawkmodel files are plain Java property files that contain a list of name-value pairs with the options to be used. hawk+http(s):// URLs provide the address of the desired Hawk endpoint as their path and encode the name-value pairs as query parameters.

Users can create .hawkmodel files with an Eclipse-based editor (shown in Figure 8), which can also produce short and long versions of the equivalent hawk+http(s):// URLs. The short version does not include fields which have default values, and the long version includes all fields regardless of their value.

Whether using a .hawkmodel file or a URL, the name-value pairs are used to populate a HawkModelDescriptor object, which is passed to the HawkResourceImpl to initiate loading. The supported options are listed in Table 2 along with their names as Java properties or URL query parameters.

#### Initial load

The initial load of a HawkResourceImpl can behave in different ways depending on the selected loading mode. The available loading modes are listed in Table 3. These loading modes are backed by several operations in the Thrift API for retrieving entire models, retrieving root model elements and retrieving individual root model elements by graph node ID. The Thrift-based model element serialization is described in D5.6 [18].
Remote Hawk Descriptor

Open with Exceed Copy short URL to clipboard Copy long URL to clipboard

Instance
Access details for the remote Hawk instance.
Server URL: http://127.0.0.1:8080/thrift/hawk/tuple
Thrift protocol: TUPLE
Instance name: set0

Contents
Filters on the contents of the index to be read as a model
Repository URL: file:///C/Users/wv8596/Desktop/mondo-demos/models/set0/
File pattern(s): set0.xmi
Loading mode: GREEDY
Query_language:
Query:

Subscription
Configuration parameters for subscriptions to changes in the models indexed by Hawk.
Subscribe:
Client ID: wv8596
Durability: DEFAULT

Figure 8: Screenshot of the Eclipse-based .hawkmodel editor
Element loading

Model element nodes are converted into EMF EObject instances in a similar way to that of the local EMF resource abstraction:

1. EObjects are still created through the appropriate EFactory instances, for the same reasons.

2. EAttributes are set through the SlotDecodingUtils utility class, which decodes the Thrift representations of the attribute values into their expected forms in EMF. If using lazy attributes, the EObject will initially not have any attributes set: once the user requests an attribute, all the other attributes will be fetched as well.

3. Lazy EReferences are resolved on demand by using CGLIB instrumentation, as in the local EMF resource. The intercepted eGet invocations are resolved into requests for the Thrift API. The resource ensures that every model element is only requested once, by maintaining a map from graph node IDs to the deserialized model elements.

Model element deserialization requires keeping some internal state within the resource during loading:

- The last seen values for the repository, file path, metamodel URI and type name fields. This is needed since the Thrift ModelElement structs omit repeated occurrences of these values, as seen from a pre-order traversal of the tree.
- A list of all the created EObjects. This is needed to resolve position-based references, which use a more efficient encoding than ID-based references.
- A map from the ModelElement Thrift structs to their resulting EObjects. This map is needed to bridge the two stages of object creation: the first stage creates all the EObjects and sets their attributes, and the second stage fills in all the references.
<table>
<thead>
<tr>
<th>Java property</th>
<th>Query parameter</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><em>Connection details</em></td>
<td></td>
</tr>
<tr>
<td>hawk.url</td>
<td>(address)</td>
<td>URL of the Thrift endpoint for Hawk.</td>
<td><a href="http://127.0.0.1:8080/">http://127.0.0.1:8080/</a>...</td>
</tr>
<tr>
<td>hawk.thriftProtocol</td>
<td>tprotocol</td>
<td>Thrift protocol (e.g. TUPLE or JSON).</td>
<td>TUPLE</td>
</tr>
<tr>
<td>hawk.instance</td>
<td>instance</td>
<td>Name of the remote Hawk instance.</td>
<td>myhawk</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Exposed contents</em></td>
<td></td>
</tr>
<tr>
<td>hawk.repository</td>
<td>repository</td>
<td>Repository pattern to be exposed (supports “*” as wildcard).</td>
<td>*</td>
</tr>
<tr>
<td>hawk.files</td>
<td>filePatterns</td>
<td>Comma-separated list of file patterns to be exposed (supports “*” as wildcard).</td>
<td>*</td>
</tr>
<tr>
<td>hawk.loadingMode</td>
<td>loadingMode</td>
<td>Loading mode to be used (e.g. GREEDY or LAZY_CHILDREN).</td>
<td>GREEDY</td>
</tr>
<tr>
<td>hawk.queryLanguage</td>
<td>queryLanguage</td>
<td>If present, the language in which the query is written.</td>
<td>Not set.</td>
</tr>
<tr>
<td>hawk.query</td>
<td>query</td>
<td>If present, the initial load will only include the results of this query.</td>
<td>Not set.</td>
</tr>
<tr>
<td>hawk.split</td>
<td>split</td>
<td>True if surrogate file resources are desired, false otherwise.</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Live updates</em></td>
<td></td>
</tr>
<tr>
<td>hawk.subscribe</td>
<td>subscribe</td>
<td>True to enable live updates through Artemis, false to disable them.</td>
<td>false</td>
</tr>
<tr>
<td>hawk.subscriptionDurability</td>
<td>durability</td>
<td>Durability of the desired queue: default, temporary or durable.</td>
<td>DEFAULT</td>
</tr>
<tr>
<td>hawk.clientID</td>
<td>clientID</td>
<td>Unique ID that identifies an Artemis subscriptor on reconnections.</td>
<td>(local user name)</td>
</tr>
</tbody>
</table>

Table 2: Available options for the remote Hawk EMF resource abstraction
D3.4 – Integration with Model Management Languages Report
D5.4 – Heterogeneous Model Management Framework - Final Version

Figure 9: Steps involved in live updates for remote Hawk EMF resources

Live updates

In the local case, supporting live updates was simple to implement, as the producer of the notifications (Hawk) and the consumers (the resources) ran in the same Java virtual machine. In addition to providing the transactional semantics required by `IGraphChangeListener` implementations, the remote case requires serializing the changes into messages, delivering them in the correct order, handling intermittent connections, protecting slow consumers from being flooded by messages and dealing with low memory situations.

Figure 9 shows the main components of the MONDO Server product that implement remote live updates (more details in D6.3 [19]). A servlet embeds instances of Hawk and exposes them through an API, and these instances maintain indexes of one or more model repositories. The API for live updates relies on the serialization capabilities of the Apache Thrift libraries [2] and on message queues implemented by the Apache Artemis messaging system [1]. More specifically, the MONDO Server product embeds a copy of the Apache Artemis server. The Artemis server is configured in the following manner:

- The Core and STOMP/WebSockets protocols are enabled on the port indicated by the `hawk.artemis.port` Java system property (by default, 61616).
- The Artemis data directories for paging, bindings, journalling and storing large messages are created as standard OSGi data directories within the server.
- Messages are paged to disk when an address accumulates more than 100MB of data.
- Delivery is attempted again after 2 seconds, with a 1.5 backoff multiplier and a maximum delay of 30 seconds.

After the initial load has completed, the resource will enable live updates for the model by following these steps (also listed in Figure 9):

1. During loading, if the user has enabled the option in the `HawkModelDescriptor` (step 1 in the figure), the resource invokes the `watchModelChanges` operation of the Thrift API (step 2) implemented by the Hawk servlet from D6.3 [19].
<table>
<thead>
<tr>
<th>Name</th>
<th>Initial load</th>
<th>Lazy loads</th>
<th>Recommended usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td>Entire model.</td>
<td>None.</td>
<td>Only for small/medium models (below 200k elements). Mostly for training and testing.</td>
</tr>
<tr>
<td>Lazy attributes</td>
<td>Entire model without attributes.</td>
<td>On-demand attributes: when resolving e.attr, all attributes of e are prefetched.</td>
<td>For models that will be navigated exhaustively but only have a few attributes accessed.</td>
</tr>
<tr>
<td>Lazy children</td>
<td>Root elements.</td>
<td>On-demand references: when resolving e.ref, all references of e are prefetched.</td>
<td>Recommended for most cases.</td>
</tr>
<tr>
<td>Lazy references</td>
<td>Root elements.</td>
<td>On-demand references.</td>
<td>For large models in which few distinct references will be navigated, and most nodes will have their attributes accessed.</td>
</tr>
<tr>
<td>Lazy attributes and children</td>
<td>Root elements without attributes.</td>
<td>On-demand attributes and references, with prefetching of both.</td>
<td>For large models in which only a few elements have attributes of interest.</td>
</tr>
<tr>
<td>Lazy attributes and references</td>
<td>Root elements without attributes.</td>
<td>On-demand attributes and references, with prefetching of attributes.</td>
<td>For large models in which only a few elements have attributes of interest, and very few references will be navigated.</td>
</tr>
</tbody>
</table>

Table 3: Available loading modes for the remote EMF resource abstraction
The server will compute the appropriate Artemis address and queue name for the request: the address will be based on the requested contents only, and the queue name will also take into account the client identifier. Messages sent to an Artemis address are copied to all the queues associated with it. To readers familiar with the Java Message Service, an Artemis address can be understood as a JMS topic and an Artemis queue as a JMS queue.

The server will then create the queue (step 3) if its durability level is “default” or “durable” (“temporary” ones are created by clients). “Default” queues survive user reconnections but do not survive server restarts. “Durable” queues are persisted to disk and survive server restarts. “Temporary” queues are dropped as soon as the client disconnects.

Next, the server will install (if not already present) an instance of `ArtemisProducerGraphChangeListener` that will watch over the selected models and translate the change events into Thrift `HawkChangeEvent` unions, which will be sent to the previously computed Artemis address. Messages are only sent if the synchronization is successful, after the underlying transaction has been committed and all graph nodes involved have valid identifiers.

Finally, the server will send back a `Subscription` Thrift struct with the connection details to the created Artemis queue (step 4).

2. With all the preparations done in the server, the resource uses the received connection details to open a session in the provided Artemis queue (step 5) and sets up a `Consumer` from the `uk.ac.york.mondo.integration.artemis` plugin that will invoke a `HawkResourceMessageHandler` every time a message is received.

3. After someone makes a change in the models (step 6) and Hawk detects it, a sequence of Thrift `HawkChangeEvent`s is produced and added as messages to the queue (step 7).

4. Artemis delivers each message to the `HawkResourceMessageHandler` (step 8). The message is deserialized back into a `HawkChangeEvent`, which is used to update the contents of the resource, producing the appropriate EMF change notifications (step 9). The resource uses Java monitors to ensure mutual exclusion in the contents of the resource, as Artemis uses a pool of threads to process incoming messages.

5. When the resource is unloaded (step 10), the Artemis session is closed (step 11). Temporary queues will be deleted, default queues will remain until the server restarts, and durable queues will be preserved.

### 5.4 Integration with Epsilon Exeed: lazy browsing

Browsing through a model is the simplest usage scenario for the lazy loading and live update functionalities of the EMF resource abstractions. To test this scenario, `.localhawkmodel` and .localhawkmodel and

---

10 While new nodes in Neo4j have a persistent identifier as soon as they are created, OrientDB documents do not have permanent identifiers until the transaction completes successfully.

11 Note again that users that wish to tell apart lazy loads from live updates should install a `HawkResourceChangeListener` into the resource.
.hawkmodel files were opened normally from the Eclipse user interface with the standard generic
tree editor provided by EMF (the “Sample Reflective Ecore Model Editor”).

While it did display the models as expected, inspecting network traces collected with a packet log-
ger showed that it was loading the entire model every time, even when using the lazy modes. Further
inspection showed that these unwanted loads were due to several reasons: the default DiagnosticDec-
orator used to show error/warning emblems accessed the leaves of the tree to propagate information
to the roots, and computing whether an expand/collapse handle should be shown was done through
code of the form of `getChildren().isEmpty()`, which triggered a lazy load unnecessarily.

To allow MONDO partners to browse through models while taking advantage of the lazy loading
modes, a new model editor would have to be provided. Fortunately, YORK had previously devel-
oped Epsilon Exeed\(^{12}\) an enhanced version of the generic EMF tree editor that can show additional
information that is useful for querying and can customize model element icons and labels by anno-
tating the metamodels accordingly.

A new extension point was added to Epsilon Exeed: contributing plugins could modify the behavior
of the editor when used to view a specific implementation of the EMF Resource interface. Plugins
were then created to customize Exeed for the local and remote EMF resource implementations in the
following ways:

- The DiagnosticDecorator instance does not propagate information from the leaves to the roots
  anymore.
- Computing whether an expand/collapse handle should be shown is performed without trigger-
ing a lazy load.
- Since Hawk resources are read-only, the “Add Child” and “Add Sibling” context menus have
  been removed.
- An additional “Custom” submenu has been added to expose the methods in the HawkResource
  interface through the UI. Currently, “Fetch by EClass” is exposed as a dialog that asks for a
  metamodel and an EClass and retrieves all instances of the EClass in the model, regardless of
  their location.

The extended Exeed editor was then used to open a series of .hawkmodel files referring to an
indexed version of the set0.xmi file of the GraBaTs 2009 JDT case study (8962KB before com-
pression and 582KB after gzip -9 compression). Table 4 collects all the results with and without
gzip compression. The most efficient Thrift encodings (compact and tuple) surpass the regular XMI
encoding both before and after compression. Live updates require sending unique node IDs, which
reduces the compression ratios. However, using the lazy modes can make up for it drastically, espe-
cially with the lazy children mode: the initial load requires less than half a KB for it, since only the
root model element has to be sent.

5.5 Integration with CloudATL: consuming remote models

The CloudATL tool (developed in WP3) runs transformations written in the ATLAS Transformation
Language (ATL [8]) on a Hadoop cluster, using a special version of the ATL virtual machine called
ATL/MapReduce.

\(^{12}\)http://www.eclipse.org/epsilon/doc/articles/inspect-models-exeed/
Between WP3 and WP5, CloudATL was extended so it could use Hawk indexes as input models, in addition to regular XMI files. Since CloudATL is a regular Java application and is not based on Eclipse, it was not possible to simply include the same plugin that was used to extend regular Eclipse environments to support .hawkmodel files and hawk+http(s):// URLs.

Instead, the ATLMRUtils and RecordBuilder classes in CloudATL were modified to explicitly install UnsplitHawkResourceFactoryImpl into its own EMF resource factories. This is a modified version of the regular resource factory for the remote case which forces the “split by file” option to be always disabled, as ATL requires all source elements to be contained in the HawkResourceImpl itself and not the surrogate HawkFileResourceImpl resources.

After these modifications, CloudATL transformations can consume .hawkmodel files that were previously uploaded to the Hadoop File System. To avoid this step, it is also possible to consume hawk+http(s):// URLs, retrieving the model directly from the MONDO server without accessing HDFS.

### 5.6 Integration with IncQuery: lazy loading with incremental querying

The IncQuery incremental model querying language developed in WP3 normally operates by following these steps:

1. The query is transformed into a RETE network.
2. The model to be queried is fully loaded into memory.
3. The loaded model is traversed and indexed through an EMFQueryContext and an EMFScope into the tuple-based representations required by the RETE network.
4. The tuples are fed into the RETE network.
5. The activations of the RETE network produce a list of all the matches of the query.
IncQuery implements incremental query re-evaluation by watching over the model with EMF Adapters and translating EMF notifications into tuples which are fed into the RETE network, updating the existing result set.

**Naive integration**

With all this in mind, a naive integration of a local Hawk instance with IncQuery case would be simply a matter of using the `.localhawkmodel` file as the input model, instead of the actual XMI: the entire model would be recovered from the graph, and live updates would ensure the appropriate EMF notifications were produced.

For a remote Hawk index, this would be equivalent to using a `.hawkmodel` file in the greedy loading mode with live updates enabled. `.hawkmodel` files support filtering by repository, file and/or query, so at the very least it should be possible to limit the model to be loaded to the subset needed by the query itself. This naive integration does not use a `HawkResourceChangeListener`, so lazy loading modes would not be supported: IncQuery would not be able to tell apart EMF notifications produced by lazy loads from those coming from live updates.

The naive integration scenario was tested correctly on a remote Hawk index: IncQuery was able to consume it normally and produce the desired query results, without requiring any modifications. Nevertheless, while this was convenient for users wishing to query a remote model, it did not take advantage of the lazy loading modes provided by Hawk or the specialized methods in the expanded `HawkResource` interface.

**Optimized integration**

To make the most out of Hawk in IncQuery, an additional plugin was developed that contributed a new subclass of `IncQueryScope` called `HawkScope`. A `HawkScope` sets up IncQuery with a graph of `Hawk*` objects instead of their regular `EMF*` counterparts. These custom objects modify the behaviour of IncQuery in several key aspects:

- The model does not need to be fully loaded anymore: only the root elements and the parts of the model required by the query are retrieved, by using the specialized methods in the `HawkResource` interface.
- The model does not need to be indexed by IncQuery, as this work has been performed already by Hawk.
- Models modified outside the user’s environment can be incrementally queried: Hawk will watch over them and notify IncQuery through Apache Artemis (see Figure 9 on page 35).
- IncQuery uses notifications from a custom `HawkResourceChangeListener` implementation, instead of traditional EMF notifications. Lazy loading modes can be used normally again.

**Validation**

To validate this integration scenario, it was run on the custom benchmark used by IncQuery for its internal evaluation and validation. The main goal of this benchmark is measuring execution times
for graph-based querying, with emphasis on incremental re-evaluation of queries as models evolve. It has been used in various publications including [23, 24, 26]. It uses the metamodel shown in Figure 10 and systematically generates instance models of incrementally larger sizes, using various randomization points such as exact number of elements and cardinalities.

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

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

The train benchmark offers a collection of 24 test cases which test and evaluate the performance and correctness of different operations performed on a specific train model with respect to a set of well-formedness constraints. For example, the ConnectedSegments test case injects faulty sensors with more than five segments (Figure 11a) and then repairs some of these sensors (Figure 11b). Hawk has passed all of these test cases, providing additional confidence regarding the correctness of the contents of the index after initial and incremental updates, the results used through EOL queries and the operation of the remote EMF resource abstractions.

5.7 Future integration in scalable notations: efficient retrieval of candidates

Beyond the planned integrations with the languages in WP3, the scalable notations developed in WP2 provided an unexpected scenario in which the Hawk model indexer could also be useful. The fragmented models produced by the editors developed in WP2 presented a challenge when needing to list all candidate values for an **EReference**: finding these candidate values could potentially require loading all the fragments back into memory.

To avoid this, YORK and UAM collaborated to develop a proof of concept for an integration between Hawk and the DSL-tao and EMF-Splitter tools from WP2. Hawk would index the contents of the opened Eclipse workspace into a graph, and the WP2 editors would query this index to quickly find the candidates for **EReferences**.

An initial prototype for this integration has been developed, by extending Hawk in two ways:

- A new plugin was created, providing the **Workspace** model connector for indexing local Eclipse workspaces and reacting immediately to changes in the workspace.
- The local EMF resource abstraction shown in Section 5.2 was created, so DSL-tao would not have to re-implement the conversion of model element nodes to **EObject**s.

The improved version of Hawk was then tested for this scenario by following these steps:

1. In the same Epsilon Exeed editor (see Section 5.4), a model fragment created with one of the editors in WP2 and a `.localhawkmodel` file were loaded.

2. From one of the **EObject**s in the fragment, a reference to one of the **EObject**s in the `.localhawkmodel` was created.

3. The models were saved: the model fragment was updated, and the `.localhawkmodel` silently ignored the request to save.

---

14Hawk EMF resources are read-only views: we ignore the request instead of throwing an exception as we do not want to show unnecessary error messages to users.
4. The editor was closed and reopened with just the saved model fragment, and it could resolve the cross-file reference with no issues, as the `localhawkmodel` produced `EObject`s with the same URL as the actual `EObject` in the referenced fragment.

After this proof of concept was successful, UAM extended the editors for the scalable notation in WP2 with a new extension point for integrating in their user interfaces alternative indexing solutions for finding candidate values for references. During the evaluation stage, YORK will develop a plugin that will use this extension point to integrate Hawk into the WP2 editors.

6 Conclusions

This document has presented the final outcomes of Task 5.2 (“Heterogeneous model management”) and Task 3.6 (“Integration with scalable persistence mechanisms”). More specifically, it has presented a heterogeneous model management framework (Hawk) consisting of a set of uniform model/metamodel parsing interfaces and updated implementations of these interfaces for three different types of models: EMF/XMI, BIM/IFC and models persisted in the native Modelio format.

In terms of querying, a component has been developed that allows users to query heterogeneous model indexes using the EOL model-query language. In addition, two EMF resource abstractions (one for local graphs and one for remote graphs) have been developed to expose Hawk indexes and their changes as standard EMF models and EMF notifications, respectively. These supersede the original Mondix proposal for live updates from D5.2, and have been successfully used to integrate Hawk with the CloudATL model transformation language and the IncQuery model querying language from WP3. Beyond these planned integrations, Hawk was integrated with the Epsilon Exeed tree-based model browser, and a proof of concept for integrating with the scalable notations in WP2 was developed.
IFC model type detection algorithm

1. Try to open the file as a ZIP file. If it fails, continue with the next step. If it succeeds:
   (a) Find the first entry with one of the supported extensions, ignoring case: .ifc, .ifcxml, .ifc.txt, .ifcxml.txt, .ifc.zip, .ifczip.
   (b) Unpack the entry in memory (without writing to disk) and restart the process on this entry.

2. Read the first line of the file. If it is not “ISO-10303-21;”, continue with the next step. Otherwise, the file is in STEP format and these steps can be followed:
   (a) Obtain the first line that starts with “FILE_SCHEMA” before the first “ENDSEC;” line: if no such line exists, the model type is “unknown”. Otherwise, continue with the next step.
   (b) If the obtained line contains “IFC2X3”, the model type is “IFC2x3 STEP”. If it contains “IFC4”, the model type is “IFC4 STEP”. If it contains “IFC2x2”, the model type is “IFC2x2 XML” and an error is thrown indicating that the BIMserver parser does not support this version. Otherwise, the model type is “unknown”.

3. Open a streaming XML parser on the file (which only reads the file incrementally as requested), and look for the first “ifcXML” or “iso_10303_28” element:
   - If it is an “ifcXML” element, the model type is “IFC4 XML”.
   - If it is an an “iso_10303_28” element, look for the first “uos” element within it. If not found, the model type is “unknown”. If found, obtain the value of its “configuration” attribute:
     - If it is “ifc2x3”, the model type is “IFC2x3 XML”.
     - If it is “ifc2x2”, the model type is “IFC2x2 XML” and an error is thrown indicating that the BIMserver parser does not support this version.
     - Otherwise, the model type is “unknown”.

4. If at this point no decision has been reached, the model type is “unknown”.
Source Code

The source code for the software described in this report can be downloaded via Git using the following commands:

- `git clone https://github.com/mondo-project/mondo-hawk.git`
- `git clone https://github.com/mondo-project/mondo-integration.git`

The source code of the *mondo-hawk* project is automatically tested on every commit using the Travis continuous integration system\[^5\] producing an update site with the EPL-compatible components\[^{16}\].

The various components discussed in this report can be found in the following locations:

- Section 3.1, the model parsing interfaces are in `hawk.core` of *mondo-hawk*.
- Section 3.2.2, the IFC parser is in `hawk.ifc` of *mondo-hawk*.
- Section 3.2.3, the Modelio parser is in `hawk.modelio` of *mondo-hawk*.
- Section 4, the query interface is in `hawk.core` and the EOL `IQueryEngine` implementation in `hawk.epsilon` (both from *mondo-hawk*).
- Section 5, the `HawkResource` interface and the local EMF resource abstraction are in the `hawk.emfresource` (core) and `org.hawk.ui.emf` (UI) plugins of *mondo-hawk*. The remote EMF abstraction is in the `mondo-integration` repository, and is split across `integration.api` (API), `integration.hawk.emf` (*HawkResource* implementation) and `integration.hawk.emf.dt` (UI).

In order to build the software, an Eclipse development environment is required. Most of the dependencies for Hawk are included in the Eclipse target platform definition file in `hawk.targetplatform`, which can be resolved by Eclipse itself. However, several components require external dependencies that cannot be included in the target platform definition:

- `hawk.neo4j` and `hawk.orientdb` require some of the Neo4j and OrientDB libraries, respectively. These can be downloaded automatically by using the IvyDE Eclipse plugin\[^7\].
- `hawk.ifc` requires downloading a list of libraries from the BIMserver website, which are explicitly listed in a file within the repository.
- `hawk.modelio` requires downloading the source code of a specialized version of Modelio 3.2.1, available in a separate Git repository\[^{18}\].

---

\[^5\]: https://travis-ci.org/mondo-project/mondo-hawk
\[^6\]: http://mondo-project.github.io/mondo-hawk/updates/
\[^7\]: https://ant.apache.org/ivy/ivyde/
\[^8\]: https://github.com/mondo-project/hawk-modelio
References


