Systematic Recovery of MDE Technology Usage

Juri Di Rocco[1], Davide Di Ruscio[1], Johannes Härtel[2], Ludovico Iovino[3], Ralf Lämmel[2], Alfonso Pierantonio[1]

[1] Department of Information Engineering, Computer Science and Mathematics
Università degli Studi dell'Aquila - L'Aquila (Italy)
name.surname@univaq.it
[2] Faculty of CS, University of Koblenz-Landau (Germany)
{johanneshaertel|laemmel}@uni-koblenz.de
[3] Gran Sasso Science Institute - L'Aquila (Italy)
ludovico.iovino@gssi.it

Abstract. MDE projects may use various MDE technologies (e.g., for model transformation, model comparison, or model/code generation) and thus, contain various MDE artifacts (such as models, metamodels, and model transformations). The details of using the MDE technologies and the relationships between the MDE artifacts are typically not accessible at a higher level of abstraction, which makes it hard to understand, build, and test the MDE projects and thus, to reuse the contained MDE artifacts. In this paper, we present a megamodel-based reverse engineering methodology and an infrastructure MDEPROFILER for recovering details of using MDE technologies in MDE projects and modeling these details at a higher level of abstraction. We exemplify the approach for MDE projects that use ATL-based model transformations.

1 Introduction

The Model-Driven Engineering (MDE) community has made considerable progress in recent years with improving productivity and quality in software development. However, cost-efficient adoption of MDE is still a challenge [1]. Introspection about processes and usage of model-driven techniques and technologies may help here, if it enables a higher-level representation of tooling architectures, thereby enhancing understanding and reuse.

Research problem MDE projects may use various MDE technologies (e.g., for model transformation, model comparison, or model/code generation) and thus, they contain various artifacts (such as models, metamodels, and model transformations). The details of using MDE technologies and the relationships between the artifacts are typically not accessible at a higher level of abstraction, which makes it hard to understand, build, and test the projects and thus, to reuse the contained artifacts. This problem is arguably very relevant for model repositories [2,3] which, as a result of lacking access to a higher level of abstraction regarding usage of MDE technologies, end up focusing on aggregation of artifacts without attached 'architectural' information.

In principle, one could use a megamodeling approach, up to the point of executable megamodeling scripts [4], for managing MDE projects. The model elements of a megamodel are artifacts such as models, metamodels and transformations. A megamodel also contains relationships between artifacts, for example, conformance and transformation. Thus, megamodeling offers the possibility to specify relationships between artifacts and to navigate between them. For a megamodel to be practically useful though, it would need to address the technological heterogeneity of MDE projects which rely on, for example, mainstream build systems, scripting languages, and test frameworks.

Further, we must not limit ourselves to prescriptive megamodeling or forward engineering; we also need to be able to 'discover' megamodels and 'recover' their instances systematically, semi-automatically, and efficiently so that we can benefit from them without much extra developer effort. Thus, we face a problem similar to *software architecture reverse engineering* or *architecture recovery* [5,6] in that software projects may lack higher-level architectural descriptions. Recovery is to be leveraged when a suitable description has never existed or it is no longer 'in sync' with the actual code. In an MDE technological context, we may be interested in architectural knowledge such as model artifacts in a project, more specific types of models (e.g., metamodels), model-to-metamodel conformance, applications of model-management operations (e.g., model transformation, model/code generation, model merging, model weaving, model comparison, and model patching), evolution-related relationships, and some types of technological traces, for example, build scripts, launcher configurations, or tests.

Previous work by (some of) these authors The software language repository YAS manages a technologically heterogenous project by a suitable megamodel, as described in [7], but this work does not address 'mainstream MDE', neither does it address reverse engineering. Megamodeling is discussed for MDE technologies (including EMF, ATL, and Xtext) in [8], but reverse engineering is not addressed, despite being stated as a direction for future work. A rule-based approach to mining artifact relationships with an application to EMF is presented in [9], but no methodology for discovering megamodels is provided. All of this previous work invokes the term 'linguistic architecture' [10] as a form of megamodeling and a form of software architecture.

Contributions of the paper We present a megamodel-based reverse engineering methodology and an infrastructure MDEPROFILER for recovering details of using MDE technologies in MDE projects. We exemplify the approach for MDE projects that exercise ATL-based model transformations. Our experimental validation is limited to ATL and 'implied' technologies or languages such as Ecore, KM3, Ant, and launcher configurations, but our approach is completely 'generic'. Our methodology is designed to support the iterative process of discovering a pool of recovery heuristics for megamodel elements. We demonstrate the methodology with the ATL Zoo.[4]

Road-map of the paper Section 2 describes our methodology for recovering MDE-technology usage. Section 3 describes our infrastructure for recovery. Section 4 evaluates our approach by means of a case study for the ATL Zoo. Section 5 discusses related work. Section 6 concludes the paper.

[4] `https://www.eclipse.org/atl/atlTransformations/`

2 Recovery Methodology

Figure 1 summarizes key aspects of our methodology. Any number of MDE projects (possibly also adding new ones over time) are analyzed semi-automatically to recover megamodels representing MDE-usage information. Heuristics are used to locate artifacts of interests (e.g., models) and artifacts that encode relationships (e.g., build scripts with model transformation applications). The recovered megamodels are essentially graphs with artifacts of interest as nodes and relationships as edges. Simple measures are computed for the megamodels. In particular, 'dangling' nodes are determined, as they are considered indicators of missing relationships. Domain knowledge and technology documentation are leveraged to manually refine the applied heuristics and to conceive new ones until all the artifacts of the analyzed MDE projects are modeled together with the corresponding relationships. This recovery process is intrinsically incremental. In the sequel, we discuss artifacts in MDE projects, relationships between them, and heuristics for relationship inference in more detail.

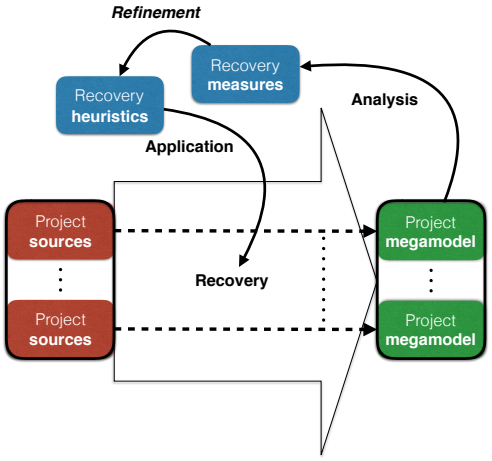


Fig. 1. Megamodel-based reverse engineering.

2.1 Artifacts in a MDE Project

As shown in Fig. 2 and described below, several kinds of artifacts are considered when applying MDE. *Available artifacts* make up the system in terms of its source code and other resources that are available typically through version control or download. *All artifacts* includes artifacts that may be *not at all or not directly available*. For instance, an artifact may only be obtainable by system building or testing. Also, an artifact may only be transient (e.g., as a runtime object during the execution of a testcase). Further, an artifact may only be obtainable by some well-defined computational step, for instance the application of a code generator — with or without this application being exercised by build management or testing. Yet further, an artifact may be unavailable, but its existence, at least, in the past, is known simply because there are traces of it (i.e., references to it) in the available artifacts. *Artifacts of interest* are those (available or not) that are obviously of interest for recovering technology usage. In the case of ATL-based model transformation, artifacts of interest are clearly the ATL transformations themselves, but also source and target models for transformations as well as metamodels for conformance. *Artifacts with traces* are those (available) artifacts (of interest or not) in which we may locate traces to artifacts (mainly references). Subject to a classification of the

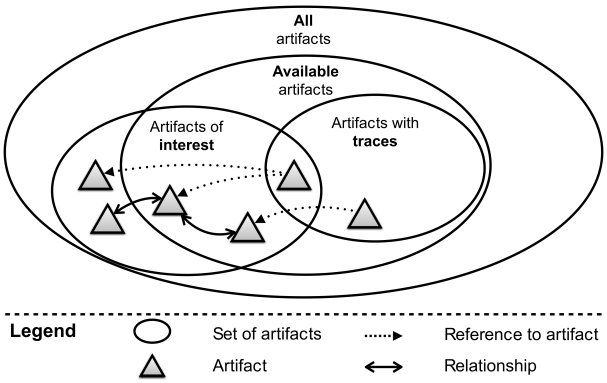


Fig. 2. Artifacts in a MDE project.

artifacts with traces, these artifacts may be interpreted as (encoding) relationships between artifacts.

The overall assumption is that we may identify artifacts of interest by examining algorithmically the available artifacts and we may identify relationships between artifacts by examining, again, algorithmically available artifacts on the grounds of technology-specific patterns for traces.

2.2 Relationships to be Recovered

Figure 3 identifies 'abstract' artifacts of interest with relationships for the running example of ATL. In particular, there are source and target models, the corresponding metamodels (MMs), the actual ATL model transformation (MT), and the application thereof. We also show relationships between these artifacts that need to be recovered. Relationships between artifacts, e.g., conformance and transformation application in the example, can be identified in different ways:

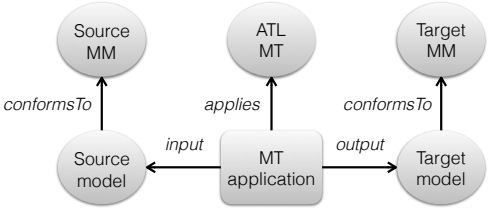


Fig. 3. 'Abstract' artifacts and relationships for ATL usage.

Trace-based identification Based on the type of referring artifact (e.g., an ANT file), based also on the details of reference (e.g., the argument position of an ATL transformation execution), one may identify a relationship (e.g., a model to serve as the 'source' of a model transformation).

Computational identification By considering a more or less standardized, technology-specific functionality (e.g., the operation for Ecore-based conformance checking) on given candidate artifacts (e.g., a model artifact and a metamodel artifact), one may identify a relationship (e.g., conformance).

Mining-based identification Based on a more 'ad-hoc' application of technology-specific functionality (e.g., a comparison of vocabulary extracted from various artifacts) on given candidate artifacts, one may identify a relationship (e.g., similarity).

2.3 Heuristics for Recovery

Heuristics for identifying artifacts of interest and finding traces for relationships are based on the following techniques mostly inspired by existing work on reverse engineering and megamodeling.

Filename heuristics Many types of artifacts may be precisely detected on the grounds of filenames or extensions thereof [11]. For instance, the '.atl' extension identifies an ATL model transformation — especially within an MDE project. Clearly, filenames may not always be sufficient; one may also need to consult the content of files for the purpose of artifact classification. For instance, EMF models may be stored in '.xmi' files, but other extensions are also used.

Watermark heuristics Some types of artifacts may be precisely detected by looking for specific content patterns ('watermarks') in files [11,12]. For instance, a syntax definition for the EMFText technology would be a '.cs' file that contains the string 'syntaxdef' [12]. (The extension '.cs' alone would be imprecise, if we assume that C# files could also be in the same project.)

Parser heuristics Some types of artifacts may be precisely detected by just trying to parse the artifact by a standard component for the type of interest. For instance, an XML file could be precisely detected, by just invoking any XML parser, e.g., a DOM-based one, on the file in a non-lax mode. A filename or watermark heuristic can be used as a precondition, if costs of parsing are a concern [13].

Component heuristics Some types of artifacts may be precisely detected and some types of suspected relationships may be precisely verified by reusing the technology of interest, or rather a component thereof [13,8]. For instance, a suspected conformance relationship may be verified by the available component (operation) for Ecore-based conformance checking, as discussed in Section 2.2.

Extractor heuristics Customized fact extractors [13,14,15] may be used to identify traces in given artifacts, thereby helping with recovery of relationships. For instance, a heuristics for ANT files may extract instances of common patterns of using ANT for applying model transformations.

Analyser heuristics Ultimately, more advanced software analyses may be used to detect or verify relationships. For instance, one may infer source and target metamodels (or approximations thereof) from model transformations [16], thereby preparing the detection of potential source or target models on the grounds of attempted conformance checking.

Figure 4 arranges some of the heuristics that were developed in the case study of Section 4. The root node is 'abstract'; it does not correspond to any actual heuristic.

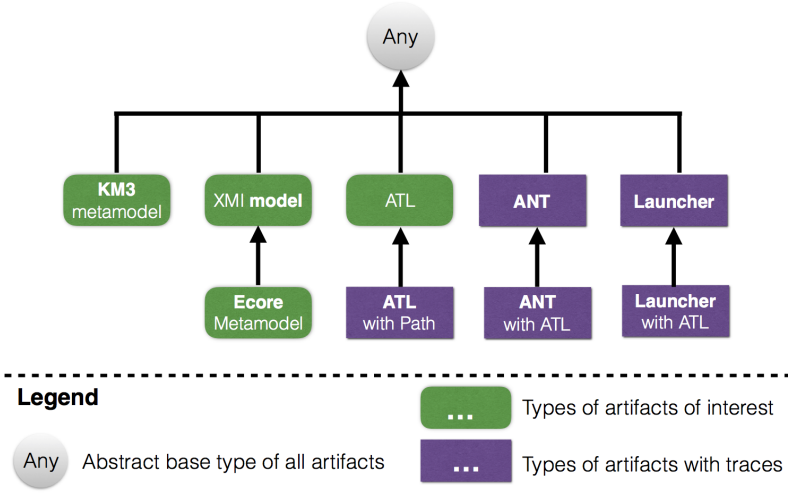


Fig. 4. Artifacts involved in the recovery of ATL usage.

The rounded (green) shapes correspond to heuristics for detecting available artifacts of interests. The angular (purple) shapes correspond to heuristics for artifacts with potential traces. The heuristics are arranged in a specialization hierarchy to express that a sub-heuristic should only be tried once the super-heuristic was confirmed. For instance, we first try to find all models and then we filter out all metamodels among them.

The key principle of the methodology is that heuristics like those in Fig. 4 are introduced in an iterative process on the grounds of measuring connectiveness of the recovered graph and leveraging domain knowledge (regarding MDE technologies) for identifying opportunities for relationship recovery by additional heuristics.

3 The Recovery Infrastructure

In this section, the recovery infrastructure supporting the methodology presented in the previous section is described. As shown in Fig. 5, the implemented recovery machinery consists of three main components, namely `RepositoryConnector`, `HeuristicsManager`, and `MegamodelVisualizer`.

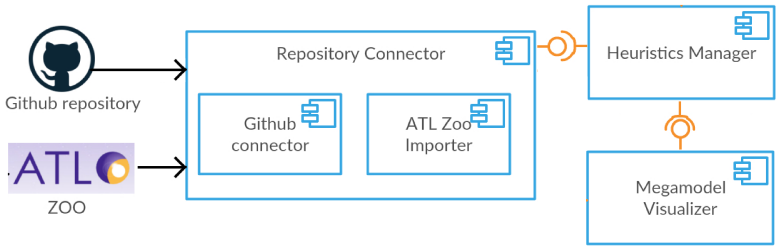


Fig. 5. Components of the recovery architecture.

The `RepositoryConnector` connects to data sources that export reusable MDE projects, which thus can be locally downloaded for subsequent analysis. Currently, the recovery infrastructure can import data from the ATL Zoo and from GitHub repositories (see Section 3.1). `RepositoryConnector` is extensible in that it provides developers with interfaces that can be implemented for adding new connectors.

The `HeuristicsManager` component is in charge of applying the available heuristics on all the projects locally downloaded by `RepositoryConnector`. The outcome of the recovery process consists of models conforming to a specifically conceived metamodel as presented in Section 3.2. The outcome of `HeuristicsManager` can be consumed in different ways — including the possibility of graphically visualizing it in order to give an overview of the analyzed projects and to support the understanding of the contained artifacts. The `MegamodelVisualizer` component presented in Section 3.3 takes recovery models as input and generates a graphical representation of them.

3.1 Repository Connector

In order to enable the analysis of MDE projects, our infrastructure downloads the projects. Currently, the infrastructure can import projects from the ATL Zoo and from GitHub repositories. The ATL Zoo is a widely used repository of model transformations, which have been the subject of several empirical works over the last few years. Unfortunately, the repository does not provide a dedicated API to easily export the available projects. Thus, HTML scraping is the only viable way to programmatically download the data available in the repository. The GitHub connector exploits the Git API[5] for locally cloning a project of interest identified by its `owner` and `name` attributes.

3.2 Heuristics Manager

Once data has been downloaded by means of the available connectors, the actual recovery process starts. The outcome of the process is a model conforming to a specifically conceived *recovery metamodel*. The heuristics currently available are presented later in this section.

The recovery metamodel As mentioned earlier, the model generated by the recovery process is a *graph* consisting of *nodes* and *edges*. For each artifact that can be identified by the available heuristics, the recovery approach generates a corresponding target node. The recovery process also detects relationships among artifacts. Detected relationships are represented as edges among previously recovered nodes. For instance, a model transformation consuming models conforming to a source metamodel and generating models conforming to a target metamodel give rise to a sub-graph consisting of nodes and edges as follows. One node would represent the analyzed transformation. Two edges would link the transformation with two further nodes representing the source and target metamodels.

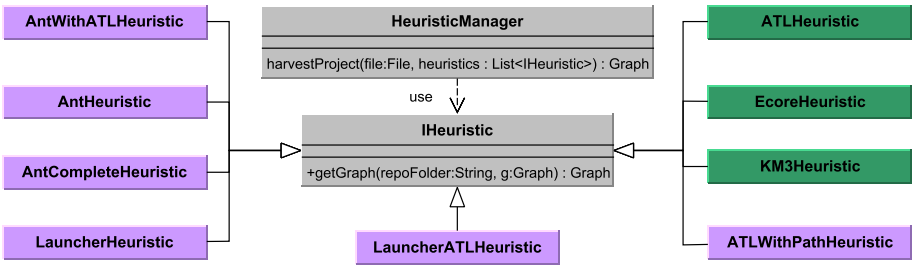


Fig. 6. Class diagram showing an overview of the Heuristic Manager.

Recovery heuristics Figure 6 shows a class diagram representing the hierarchical organization of the heuristics currently available in the `HeuristicsManager` component shown in Fig. 5. Each heuristic implements the `Heuristic` interface or extends an available implementation. In Fig. 6, the elements `ATLHeuristic`, `EcoreHeuristic`, and `KM3Heuristic` are in green color in order to be consistent with what is discussed in the previous section. These heuristics identify artifacts of interest, i.e., ATL transformations and metamodels specified either in KM3 or Ecore. Heuristics that are shown in Fig. 6 with the violet color represents heuristics that have been implemented in order to recover relationships among transformations, models, and metamodels.

Listing 1. Fragment of ATLHeuristic

```
1  package it.univaq.MDEProfiler.heuristic;
2  ...
3  public class ATLHeuristic implements IHeuristic {
4    private String extension = ".atl";
5    private String nodeKind = "NodeType.ATL";
6    @Override
7    public Graph getGraph(String repoFolder, Graph g){
8      File repoFolderF = new File(repoFolder);
9      List<File> fList = FileUtils.getFilesByEndingValue(repoFolderF,extension);
10     for (File file : fList) {
11       boolean guard = g.getNodes().stream()
12         .anyMatch(s -> s.getUri().equals(file.getAbsolutePath()));
13       if(!guard) {
14         Node n = GraphFactory.eINSTANCE.createNode();
15         n.setDerivedOrNotExists(false);
16         n.getType().add(nodeKind);
17         n.setUri(file.getAbsolutePath());
18         n.setName(file.getName());
19         g.getNodes().add(n);
20       }
21     }
22     return g;
23   }
24 }
```

Listing 1 shows a fragment of the Java implementation of `ATLHeuristic`. Essentially, in each project, the heuristic searches for files having the .atl extension (see line 4), and for each of them a new node typed `NodeType.ATL` is generated in the target recovery model (see line 5 and lines 13-20). Similarly, `KM3Heuristic` and

[5] `https://developer.github.com/v3/`

`EcoreHeuristic` search for .km3 and .ecore files, respectively and generate target `NodeType.KM3` and `NodeType.Ecore` nodes accordingly.

The recovery of relationships among generated nodes requires more elaborated analyses that should consider additional artifacts like ANT scripts and launcher files. For instance, Listing 2 shows the launch file configuration for the ATL transformation *Families2Persons*[6]. Lines 6-15 contain precious information about the input and target elements of the ATL *Families2Persons* transformation, which if considered alone does not contain such details.

Listing 2. A sample ATL launch file

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <launchConfiguration ..>
3    <stringAttribute key="ATL File Name"
4    value="/Families2Persons/Families2Persons.atl"/>
5    ...
6    <mapAttribute key="Path">
7      <mapEntry key="Families"
8        value="/Families2Persons/Families.ecore"/>
9      <mapEntry key="IN"
10       value="/Families2Persons/sample-Families.xmi"/>
11     <mapEntry key="OUT"
12       value="/Families2Persons/sample-Persons.xmi"/>
13     <mapEntry key="Persons"
14       value="/Families2Persons/Persons.ecore"/>
15   </mapAttribute>
16 </launchConfiguration>
```

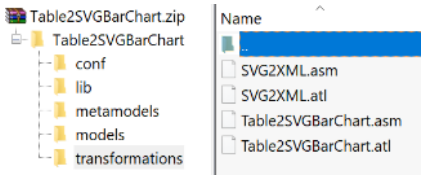
The analysis of ATL launch configuration files like the one shown in Listing 2 is implemented by the `LauncherATLHeuristic` shown in Fig. 6. Due to space limitation, this paper does not give more details about the implementation of the currently available heuristics. Interested reader can refer to the Github project related to this work[7].

3.3 Megamodel Visualizer

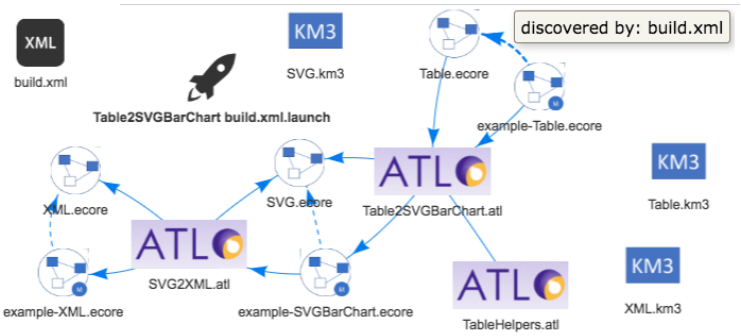
The recovered model generated for each project can be processed by other services, for example, to graphically represent the automatically recovered project as shown in Fig. 7.b. By looking at such a model, users can get a clear understanding about how the different elements are connected. By contrast, Fig. 7.a shows the folders contained in the package of the *Table2SVGBarChart* project[8], as users could explore the project by means of a file explorer and view the content of files to understand how different artifacts are related. We contend that the visualized megamodel helps much better with understanding.

The *MegamodelVisualizer* component shown in Fig. 5 is in charge of generating diagrams like the one shown in Fig. 7.b by means of an Acceleo[9]-based generator; it

[6] `https://www.eclipse.org/atl/atlTransformations/#Families2Persons`
[7] `https://github.com/MDEGroup/MDEProfile`
[8] `https://www.eclipse.org/atl/atlTransformations/#Table2SVGBarChart`
[9] `https://www.eclipse.org/acceleo/`



(a) Content of the project as it is available from the ATL Zoo



(b) Graphically representation of the recovered model

Fig. 7. The Table2SVGBarChart project.

takes a recovery model as input and generates HTML5+Javascript code. The generated code uses the Visjs[10] Javascript library and it can handle large amounts of dynamic data while enabling manipulation, representation, and interaction. For instance, in the diagram shown in Fig. 7.b, the artifact *XML.ecore* is visually associated with the Ecore type and the link with the artifact *example-XML.ecore* highlights that the latter is a model conforming to the former. Moreover, the ATL transformation *SVG2XML.atl* takes as input the *XML.ecore* node as metamodel and the *example-XML.ecore* element as model. The output consists of the *example-SVGBarChart.ecore* model conforming to the *SVG.ecore* metamodel. The node *build.xml* contributes the discovery of the represented relationship as shown by the hovering label *discovered by*.

4 Case Study

This section discusses the application of the proposed approach to the ATL Zoo consisting of ≈100 model transformation projects. The case study was performed in an iterative process in order to gradually add heuristics for new types of nodes and edges. Initially, we implemented some heuristics to identify 'obvious' artifact types of interest. Subsequently, we went through some iterations to add heuristics to recover relationships among previously discovered nodes. The case study addresses the following research questions:

- **RQ1:** What is the accuracy of recovered models?
- **RQ2:** How much effort is saved by automated recovery?

[10] `http://visjs.org`

Table 1. Case-study results

Iteration	Applied Heuristics	#Nodes	#Edges	#Dangling Nodes
1	EH	327	0	327
2	EH, AH	587	0	587
3	EH, AH, KH	795	0	795
4	EH, AH, KH, LH	887	0	887
5	EH, AH, KH, LH, ANH	1001	0	1001
6	EH, AH, KH, LH, ANH, APH	1001	37	965
7	EH, AH, KH, LH, ANH, APH, LTH	1041	236	887
8	EH, AH, KH, LH, ANH, APH, LTH, ANATLH	1133	533	745

Legend: EH: EcoreHeuristic, AH: ATLHeuristic, KH: KM3Heuristic, LH: LauncherHeuristic
ANH: ANTHeuristic, APH: ATLWithPathHeuristic, LTH: LauncherATLHeuristic
ANATLH: ANTWithATLHeuristic

Evaluation measures We use *precision* and *recall* measures as follows:

$$precision = \frac{Corr_a}{All_a} \quad (1) \qquad recall = \frac{Corr_a}{All_m} \quad (2)$$

where $Corr_a$ is the *correct number* of elements recovered by the approach, All_a is the total number of elements *automatically* produced by the approach, and All_m is the expected total number of elements as produced by a *manual* harvesting phase.

Results Table 1 shows representative results related to each iteration of the performed case study. In the first five iterations, we gradually added heuristics to discover Ecore, ATL, KM3, and ANT files. All the artifacts of interest were dangling (see the `#Edges` and `#DanglingNodes` columns). This means that we were able to increasingly discover new types of elements even though they were added in the recovery model as nodes without edges. The addition of heuristics for analyzing ATL launcher file configurations and ANT scripts for ATL automation led to a turning point. That is, even though new nodes were discovered, the number of dangling ones was decreased.

Figure 8 graphically represents the effect of applying the heuristics by focusing on the discovered and dangling nodes. The chart shows how considering specific files and properties leads to the discovery of new relationships. Starting at iteration 6, new nodes were discovered with a consequent reduction of dangling ones.

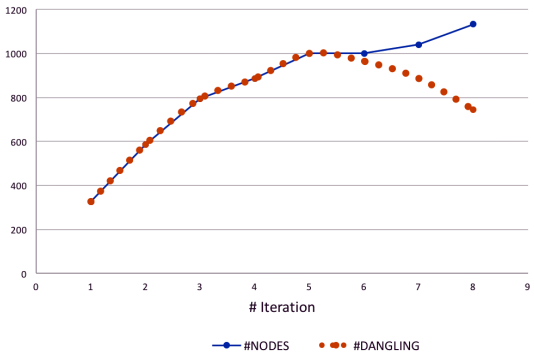


Fig. 8. Nodes recovered during the case study.

Reducing the number of dangling nodes is a challenging task, which requires the implementation of new heuristics able to cover new node types and relationships by deducing additional information from the available artifacts. For instance, by looking at the dangling nodes at the end of the last iteration shown in Table 1 we noticed that many of them are of KM3 type and many ATL transformations are defined on Ecore metamodels that

can be automatically generated from the available KM3 specifications. For instance, in the case shown in Fig. 7.b there are three dangling nodes of type KM3 (i.e., *XML.km3*, *Table.km3*, and *SVG.km3*). By applying the ATL transformation *KM32Ecore* [17] on such KM3 specifications, we noticed that the obtained Ecore metamodels are those already available in the project (i.e., XML.ecore, Table.ecore, and SVG.ecore). By applying such a heuristic, the three KM3 nodes shown in Fig. 7.b would be removed from the list of dangling nodes. The ATL Zoo contains 73 projects that have similar cases. If all of them are managed, as previously discussed, the number of dangling nodes would decrease from 745 to 347.

To evaluate the accuracy of the approach and thus, to answer *RQ1*, we manually analysed 40 projects (randomly) downloaded from the ATL Zoo. In particular, a senior modeler manually inspected such projects (without knowing in advance the results of the tools) and recovered the nodes and relations of the corresponding megamodels[11].

The case study's accuracy can be increased by means of adding heuristics. For instance, the analysed projects contain TCS specifications [18], which are currently not covered by MDEPROFILER and this is reflected by the *precision* and *recall* measures.

	Nodes	**Relations**
Precision	0.913	0.896
Recall	0.942	0.636

Table 2. Precision and recall of recovery.

To answer *RQ2*, the 40 projects considered to produce the data in Table 2 have been analysed by means of MDEPROFILER executed on an Intel Core i5 machine with 8GB of RAM. The analysis took ≈10 seconds, whereas the senior modeler needed 2 full-time working days to perform the analysis on the same data set.

5 Related Work

We begin with a discussion of heuristics used for recovery purposes in the domains of a) architecture recovery, b) traceability recovery, and c) analysis of software, technology or language usage. Afterwards, we discuss related work on megamodels in model-management systems.

Heuristics for architecture recovery Bowman et al. [19] compare three recovered architectures: a conceptual architecture based on the documentation, a concrete architecture that is derived from the actual system, and an ownership architecture extracted from version control. By examining the overlap of edges, they check whether one architecture correlates with another. Concrete, ownership and conceptual architecture recovery can be considered as a kind of heuristic. In contrast, our work combines the output of heuristics and refines the set of used heuristics through an iterative process. While Bowman et al. considers fundamentally different sources, in [20] a very fine-grained and specific set of heuristics on code-package structures is employed to guide exploration of system architecture. Our work also facilitates fine-grained exploration, by means of an extensible heuristics-based mechanism. In [21], source code is represented as a graph of, e.g.,

[11] A replication package consisting of the MDEPROFILER tool, the analysed projects, and of the obtained results is available for download at https://github.com/MDEGroup/MDEProfile

variables, types, or import relations. Here, heuristics are used in the form of patterns that are matched on this graph. These patterns contain placeholders for abstract components and connectors. An approximate instantiation on the source graph produces the resulting architecture. The methodology comes close to ours in that it facilitates domain knowledge in an iterative and interactive process to define the patterns. Our approach recovers megamodels of actual systems based on file-type recognition. This motivates our need for flexible heuristics that we implement in plain Java. In [22], the authors compare a set of alternatives to group the system using hierarchical clustering and conclude on their characteristics (e.g., one way of clustering is good for detecting utility functions). Depending on which similarity definition is chosen for clustering, this method can be seen as very general and domain-independent heuristics for grouping and connecting nodes. Architecture recovery of web applications facilitated by different extractors is pursued in [23] with a form of extractors comparable to our heuristics. The extractors also provide relationships for the web application.

Heuristics for traceability recovery Traceability recovery concentrates on mining edges between artifacts. Here, the usage of language-agnostic heuristics is very common, since trace links often reside between artifacts in different languages including natural language. For instance, in [24], links are recovered by computing the cosine similarity between the artifact term vectors. The recovered trace links connect Java and functional requirements as well as C++ and manual pages. Alternatively, in [25], sequential pattern mining is applied on commits to connect any type of artifact in a repository co-occurring in a change. We see such types of generic heuristics as a promising extension to our approach, especially to uncover unknown domain-specific heuristics. In this paper we concentrate on ATL-specific recovery.

Heuristics for software, technology, and language usage In [12], the usage of Eclipse-based MDE technologies in projects hosted on GitHub is analyzed by counting the files that are strongly related to technology usage. Another language-usage analysis of repositories, without being focused on MDE, is described in [26]. The authors also use file extensions as a heuristic to detect languages. We use file extensions only as the simplest heuristic. API usage in projects, as a very specific kind of software usage, is analyzed extensively in related work (e.g., [27,28,29]). Different features or metrics are used for characterizing API usage, for example, whether or not a component uses a given API or whether or not the component extends or simply reuses the API.

Megamodeling and executable model management Megamodels, as introduced in [30], are concerned with models as first-class entities. Megamodels are often used in executable model management systems to organize tasks on models, e.g., the application of transformations, querying, merging, and constraint checking. For instance, in [31], an explorative framework for working with models is described that follows the megamodeling principles. Alternatively, in [4], a layer on top of heterogeneous repositories is presented to get uniform model-based access to the system by writing model operations in a DSL. In [32], graphical and interactive support is described; this work is close to our model visualizer. There is no related work on megamodels where heuristics are used for identification of model elements and recovery of relationships. In some of our previous work on megamodeling [13,8,9], we considered heuristics, but without a methodology for their discovery along an iterative process.

6 Conclusion and Future Work

MDE projects are typically shared without any machine-readable description. Projects are given as packages consisting of files, possibly organized in folders, that modelers have to manually scan in order to figure out how the different project artifacts are related. Thus, understanding the artifacts contained in MDE projects and their relationships can be a strenuous and error-prone activity.

In this paper, we presented an approach based on megamodels which permits to automatically recover the structure of MDE projects represented as typed nodes and relationships among them. The approach is implemented as the recovery infrastructure MDEPROFILER. The approach has been applied in a case study on the widely used ATL Zoo consisting of ≈100 model transformation projects. In future work, we plan to apply the approach to MDE projects retrieved from elsewhere, e.g., GitHub, and to implement additional heuristics, as needed in order to minimize the number of dangling nodes and improve the overall accuracy of the approach. We are also working on extending the portfolio of MDE technologies beyond the current focus on ATL.

References

1. Tomassetti, F., Torchiano, M., Tiso, A., Ricca, F., Reggio, G.: Maturity of software modelling and model driven engineering: A survey in the Italian industry. In: Proc. EASE. (2012) 91–100
2. Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Model Repositories: Will They Become Reality? In: Proc. CloudMDE@MoDELS 2015. Volume 1563 of CEUR Workshop Procs. (2016) 37–42
3. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Collaborative Repositories in Model-Driven Engineering. IEEE Software **32** (2015) 28–34
4. Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., Cabot, J.: MoScript: A DSL for Querying and Manipulating Model Repositories. In: Proc. SLE 2011. Volume 6940 of LNCS., Springer (2012) 180–200
5. Stringfellow, C., Amory, C.D., Potnuri, D., Andrews, A.A., Georg, M.: Comparison of software architecture reverse engineering methods. Information & Software Technology **48** (2006) 484–497
6. Krikhaar, R.L.: Reverse Architecting Approach for Complex Systems. In: Proc. ICSM, IEEE (1997) 4–11
7. Lämmel, R.: Relationship Maintenance in Software Language Repositories. The Art, Science, and Engineering of Programming Journal **1** (2017) 27 pages.
8. Härtel, J., Härtel, L., Heinz, M., Lämmel, R., Varanovich, A.: Interconnected Linguistic Architecture. The Art, Science, and Engineering of Programming Journal **1** (2017) 27 pages.
9. Härtel, J., Heinz, M., Lämmel, R.: EMF Patterns of Usage on GitHub. In: Proc. ECMFA. LNCS, Springer (2018) To appear.
10. Favre, J., Lämmel, R., Varanovich, A.: Modeling the Linguistic Architecture of Software Products. In: Proc. MODELS. Volume 7590 of LNCS., Springer (2012) 151–167
11. Favre, J., Lämmel, R., Leinberger, M., Schmorleiz, T., Varanovich, A.: Linking Documentation and Source Code in a Software Chrestomathy. In: Proc. WCRE, IEEE (2012) 335–344
12. Kolovos, D.S., Matragkas, N.D., Korkontzelos, I., Ananiadou, S., Paige, R.F.: Assessing the Use of Eclipse MDE Technologies in Open-Source Software Projects. In: Proc. OSS4MDEMODELS. Volume 1541 of CEUR Workshop Procs. (2015) 20–29

13. Lämmel, R., Varanovich, A.: Interpretation of Linguistic Architecture. In: Proc. ECMFA. Volume 8569 of LNCS., Springer (2014) 67–82
14. Murphy, G.C., Notkin, D.: Lightweight Lexical Source Model Extraction. ACM Trans. Softw. Eng. Methodol. **5** (1996) 262–292
15. Ferenc, R., Siket, I., Gyimóthy, T.: Extracting Facts from Open Source Software. In: Proc. ICSM, IEEE (2004) 60–69
16. de Lara, J., Di Rocco, J., Di Ruscio, D., Guerra, E., Iovino, L., Pierantonio, A., Cuadrado, J.S.: Reusing Model Transformations Through Typing Requirements Models. In: Proc. FASE. Volume 10202 of LNCS., Springer (2017) 264–282
17. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL Frameworks. In: Companion to the 21st ACM SIGPLAN OOPSLA '06, ACM (2006) 602–616
18. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Proc. GPCE, ACM (2006) 249–254
19. Bowman, I.T., Holt, R.C.: Software architecture recovery using Conway's law. In: Proc. CASCON, IBM (1998) 6
20. Lungu, M., Lanza, M., Gîrba, T.: Package Patterns for Visual Architecture Recovery. In: Proc. CSMR, IEEE (2006) 185–196
21. Sartipi, K., Kontogiannis, K.: On Modeling Software Architecture Recovery as Graph Matching. In: Proc. ICSM, IEEE (2003) 224–234
22. Maqbool, O., Babri, H.A.: Hierarchical Clustering for Software Architecture Recovery. IEEE Trans. Software Eng. **33** (2007) 759–780
23. Hassan, A.E., Holt, R.C.: Architecture recovery of web applications. In: Proc. ICSE, ACM (2002) 349–359
24. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D.: Information Retrieval Models for Recovering Traceability Links between Code and Documentation. In: ICSM, IEEE (2000) 40–49
25. Kagdi, H.H., Maletic, J.I., Sharif, B.: Mining Software Repositories for Traceability Links. In: ICPC, IEEE (2007) 145–154
26. Karus, S., Gall, H.C.: A study of language usage evolution in open source software. In: Proc. MSR, ACM (2011) 13–22
27. Lämmel, R., Pek, E., Starek, J.: Large-scale, AST-based API-usage analysis of open-source Java projects. In: SAC, ACM (2011) 1317–1324
28. Lämmel, R., Linke, R., Pek, E., Varanovich, A.: A Framework Profile of .NET. In: Proc. WCRE, IEEE (2011) 141–150
29. Roover, C.D., Lämmel, R., Pek, E.: Multi-dimensional exploration of API usage. In: Proc. ICPC, IEEE (2013) 152–161
30. Bézivin, J., Jouault, F., Valduriez, P.: On the need for Megamodels. In: Proc. of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop. (2004)
31. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In: European MDA Workshops MDAFA 2003 and MDAFA 2004, Revised Selected Papers. Volume 3599 of LNCS., Springer (2005) 33–46
32. Sandro, A.D., Salay, R., Famelis, M., Kokaly, S., Chechik, M.: MMINT: A Graphical Tool for Interactive Model Management. In: Proc. MoDELS 2015 Demo and Poster Session. Volume 1554 of CEUR Workshop Procs. (2016) 16–19