Noname manuscript No.
(will be inserted by the editor)

Understanding MDE Projects: Megamodels to the Rescue for Architecture Recovery

Juri Di Rocco · Davide Di Ruscio · Johannes Härtel · Ludovico Iovino · Ralf Lämmel · Alfonso Pierantonio

Received: date / Accepted: date

Abstract Conventional wisdom on Model-Driven Engineering (MDE) suggests that this software discipline is key to achieve superior automation, whether it be refactoring, simulation, or code generation. However, the diversity of employed languages and technologies blurs the picture making it difficult to analyze existing MDE-based projects in order to retrieve architectural information to foster a better understanding about the rationale behind them. Thus, the ability of carefully analyzing projects to identify their components and their interrelationships is key to obtain representations at a higher level of abstraction that can support reuse processes. In this paper, a megamodel-based approach to the reverse engineering of model-driven projects is proposed in order to leverage the representation of the involved technologies and assets. An automated recovery technique implemented by the MDEPROFILER infrastructure is presented and illustrated by analyzing community projects in terms of basic MDE artifacts (such as models and metamodels) and the usage of common technologies such as model transformations and code generators.

Keywords Megamodeling, Reverse Engineering, Architecture Recovery, MDE, Code Generator, Model Transformation

1 Introduction

The Model-Driven Engineering (MDE) [64] community has made significant progress with enhanced productivity and quality software development. However, cost-efficient

J. Di Rocco, D. Di Ruscio and A.Pierantonio
University of L'Aquila (Italy)
E-mail: firstname.lastname@univaq.it

L. Iovino
Gran Sasso Science Institute (Italy)
E-mail: ludovico.iovino@gssi.it

R. Lämmel and J. Härtel
University of Koblenz-Landau (Germany)
E-mail: laemmel-or-johanneshaertel@uni-koblenz.de

adoption of such software discipline is still a challenge [68]. An introspective analysis of typical processes and usage of model-driven techniques and technologies, which leverages the representation of tooling architectures and formalizes component inter-relationships, is key to better understanding and increased reuse.

Research problem MDE projects make use of a wide range of technologies (e.g., for model transformation, model comparison, or model/code generation) and thus, they contain inherently different 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 analyze, build, and test the projects and thus, to reuse the contained artifacts. Arguably, this problem is of paramount relevance for model repositories [8,21] which, as a result of lacking access to a higher level of abstraction regarding usage of MDE technologies, end up focusing on just aggregation of artifacts without attached 'architectural' information.

In principle, one could use a megamodeling approach, up to the point of executable megamodeling scripts [43], for managing MDE projects. The model elements of a megamodel are artifacts such as models, metamodels and transformations. A megamodel also contains (typed) relationships between artifacts, for example, conformance and transformation relationships. 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* [67,46] 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.

Contributions of the paper In this paper, we present a megamodel-based reverse engineering methodology and its supporting infrastructure MDEPROFILER. The approach is agnostic of the specific technologies and enables harvesting detailed information about their employment in model-driven projects. The proposal is illustrated by analyzing a corpus of ATL-based model transformations and Acceleo code generators taken from the ATL Zoo[1], and by taking into account related tools, such as

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

Ecore, KM3, Ant, and launcher configurations. The discovery process uses heuristics for detecting and connecting megamodel elements. The discovery process is iterative in so far that one starts from basic types of megamodel elements and then performs iterations for classifying an increasing number of artifacts in projects and connecting them in the megamodels. In other words, the original contribution of this paper is the adoption and combination of a number of heuristics for the recovery of megamodels, i.e., a synthesis of the areas of reverse engineering, megamodeling, and architecture recovery.

This is an extended version of our original contribution in [20]. Specifically, in this paper the discussion about the related work is substantially extended, additional heuristics are introduced to improve the obtained results in terms of reduced number of dangling nodes in the recovered models. Moreover, two research questions (RQs) are defined and answered by means of the performed experiments: RQ1 is about the accuracy of recovered models, whereas RQ2 is related to the effort, which is saved by employing the proposed approach.

Road-map of the paper Section 2 discusses related work. Section 3 contextualizes the proposed approach with an application scenario. Section 4 describes our methodology for recovering MDE-technology usage. Section 5 describes our infrastructure for recovery. Section 6 evaluates our approach by means of a case study for the ATL Zoo. Section 7 concludes the paper.

2 Related Work

This section discusses relevant works that are related to the recovery approach described in the next section. The main novelties of the proposed recovery technique with respect to approaches that are already available in other research fields and application domains can be summarized as follows:

- *megamodel-based*: by adhering to the MDE principle that everything is a model, the recovery approach produces megamodels consisting of typed relationships between discovered artifacts;
- *extensible*: the recovery approach relies on the availability of heuristics each contributing to the identification of specific artifact types. Additional heuristics can be added to enable the recovery of new types of artifacts and relationships.

Previous work by (some of) the authors The software language repository YAS [49] leverages megamodeling to manage many language processors that are diverse in terms of implementation languages and involved technologies. However, YAS does not cover MDE and model transformation technologies such as those covered by this paper. More importantly, YAS does not involve any form of reverse engineering for obtaining the megamodel; YAS depends on authoring and maintaining the megamodel by the contributors of the repository while megamodels serve building and testing. Megamodeling is discussed for MDE technologies (including EMF, ATL, and Xtext) in [31], but reverse engineering is not leveraged, 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 [32], but no methodology for discovering megamodels is provided. All of this previous work invokes the term 'linguistic architecture' [26] as a form of megamodeling and a form of software architecture; see yet more related work on the axiomatization of a linguistic architecture [34], its interpretation [52], the linking of documentation and source code [25], and tool support for the renarration of linguistic architecture for educational purposes [53].

Heuristics for architecture recovery Bowman et al. [15] 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 [55] 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 [63], Sartipi et al. represent source code as a graph of, for instance, 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 [57], 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, representing software modules.

Architecture recovery of web applications facilitated by different extractors is pursued in [33] with a form of extractors comparable to our heuristics. This work describes a set of tools which parse and extract relations between the various components of a web application. The extracted components and relations can be visualized using a specialized viewer, very similarly to the visualizer we propose.

In [1], the recovery of components from object-oriented source code is described as a step facilitating the migration of code. Mapping the diversity of technology application that appears in MDE projects to a megamodel is related to this recovery but our approach has to handle several languages and formats – as opposed to just working with object-oriented source code as input.

In [17], the recovery of a system's internal architecture is refined by a semi-automatic process that considers documentation and expert recommendations. In our approach, documentation artifacts can be seen as potential targets for heuristics.

In [56], an architecture recovery tool is proposed that can process versions of the system. Like our approach, the authors focus on a recovery process driven by exploration and visualization; however, currently we do not consider different versions of MDE projects. We see the analysis of versions as a promising way for finding additional traces of artifacts and for performing additional measurements, for instance, the erosion of MDE technologies in projects.

In [18], a method for noise detection is proposed as a pre-processing step for program comprehension. Noise in software systems is produced by classes that are intensively utilized either system-wise, for instance omnipresent classes. The idea of noise detection is transferable to the domain of MDE. In our future work, we plan to identify MDE artifacts that are widely used in a system but do not relate to the real application scenario (e.g., usage of the Ecore metamodel). Wille at al. in [69] present guidelines and a generic implementation that both ease adaptation of a previously presented variability mining algorithm for new languages. This work also integrates a clustering approach as a pre-processing step to the mining. Babur et al. in [4] apply generic model analytics to compare the feature models in a case study repository. The final goal is testing the genericity and extensibility of the approach for new model types and datasets.

In [47], an approach is presented that uses hierarchical clustering for the reverse engineering of a complex system. The resulting clusters are visualized. Following up work in [48] adds semantic links between clusters; both focus on depicting 'semantic hot-spots' of the system, for instance, a connector between a system's core and scripting library. Such semantic hot-spots are related to the occurrence of MDE technology as its application is essential in understanding of a system.

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 [2], 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 [38], sequential pattern mining is applied on commits to connect any type of artifact in a repository co-occurring in a change. In [3], the evolution of traceability links is subject to a topic model, that is the artifact outcome of a process combining traceability with machine learning techniques; finally used to visualize and describe the system in different ways – as done in our work. 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 a running example based on Acceleo and ATL-specific recovery.

In [42], a tool provides an experimental environment for solving traceability tasks. This is done by instantiating, configuring and connecting components in a workspace. Predefined and user-defined components are collected in a library that facilitates shared and reusable knowledge. Our library of heuristics is also intended for sharing and reuse within the MDE scope. As of writing, we do not yet provide a flexible way of combining and configuring heuristics.

The motivation of [65] reflects ours in that the correspondence between models should be visible when transformations are applied. The approach uses formal rules to maintain relations between models; the technique can be applied in batch and incremental mode.

Heuristics for software, technology, and language usage In [44], 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 [41]. 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., [51,50,60]). 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.

In [35], the extraction of metric is described using open source parsers targeting projects where several languages occur. Clearly, this is the case for MDE in that our implementation of heuristics benefits for already available infrastructure (e.g., JDT).

Megamodeling and executable model management Megamodels, as introduced in [13], are concerned with models as first-class entities. Megamodels are often used in executable model management systems to organize tasks on models, for instance, the application of transformations, querying, merging, and constraint checking. For instance, in [12], an explorative framework for working with models is described that follows the megamodeling principles. Alternatively, in [43], 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 [62], 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 [52,31,32], we considered heuristics, but without a methodology for their discovery along an iterative process.

In [45], consistency checking is discussed focusing on complex systems of multi-models. The approach reduces the cost of checking by the localization of models. Our description of model interrelations in terms of a graph can be seen as some sort of localization.

In [61], relational data migration is examined with respect to applicable MDE techniques. If reverse engineering is needed as a first step towards a relational data migration, our approach may also provide the underlying infrastructure for such recovery.

In [11], the author proposes RAS++, a common representation language to be used in the core of a Knowledge Base for Model Driven Engineering. This common representation language is derived based on similarities and differences in MDE literature. Megamodel-based techniques are highlighted as important related work.

In [62], a model management tool (MMINT) providing a graphical and interactive environment for model management is discussed. Key features of MMINT are an interactive and automated user interface intended to reduce the complexity when managing models. The tool supports the user interactively, whereas our tool automatically derives relationships.

In [66], the relations between build systems and megamodels are discussed. The build system can be used in combination with a megamodel to restore desired consistency relationships between models (in case of modification). The work focuses on optimal restoration of consistency, for instance, a transformation does not have to be reapplied if the related models did not change. Our approach detects transformations but does not provide a build system due to several issues arising with arbitrary repositories.

Repository Mining In [14,40,39], the promises and perils of mining Github are discussed. For instance, a proper interpretation of the results is only possible when the specific characteristics of a repository are considered. In that manner, our research is affected by the fact the MDE projects on Github are often of academic nature. Other perils or challenges are discussed below.

When analyzing repositories, very scalable techniques are needed. In [22,23,24], a highly scalable computation infrastructure for analyzing repositories is described that compiles to a distributed map-reduce framework. Our approach is not easy to distribute as MDE technologies are often bound to a file system. We see promising future work in virtualization of the file system enabling the scaling and evolution aware MDE analysis. In [30], a technique for attaching file ownership to developers is discussed. Such ownership can clearly be transferred on MDE artifacts. Ownership of artifacts remains future work. In [16], the inner source code collaboration is measured in terms of patches (code contributions) flowing across boundaries of organizational units. MDE technology also crosses such boundaries, for instance, transformations may be developed and used by different teams. Our proposed recovery approach can be employed to better understand inner MDE technology collaboration.

Repository history provides detailed chronological data when working with several repository versions. In [70], version histories are analyzed to provide developers with some sort of guidance when changing code. Our aim is to continue by focusing on the developer as a part of the MDE process; evaluating the very complex usage of MDE technology.

In [29], the misalignment between MDE in academia and industry motivates a method to evaluate the quality of modeling languages that are used in combination. Our work can be used to measure MDE in the wild (on Github) and thereby provides some ground truth on its acceptance. For analyzing MDE technology used in combination, our approach needs to improve in terms of technologies covered. The list of MDE technologies examined in [44] can provide a useful guideline for such diversification.

3 Application Scenario

Existing repositories for modelling artifacts are very diverse: they typically expose functionalities that range from persistent storage to complex modelling environments where teams can collaboratively develop models and execute transformations. The

Repository for Model Driven Development [28] (ReMoDD) project is one of the first attempts of developing a community-driven repository. A system that permits developers, students, instructors, and researchers to make their artifacts publicly available to be inspected, studied, and potentially reused. While such repositories represents a useful route to the sharing of knowledge in communities and organizations, they are limited in terms of providing reuse opportunities because they are relatively unstructured or manually classified.

In order to leverage the profitability and scope of repositories containing knowledge objects, more advanced techniques, such as similarity- (e.g., [6]) and quality-based analysis and filtering (e.g., [7]), should be employed. For instance, the MDE-Forge [5] platform offers an unmanned classification mechanism based on clustering techniques [6] and a discovery mechanism for identifying chains by composing the transformations existing in the repository [9].

Unstructured repositories do not provide advanced classification techniques and non-basic searching functionality. Because artifacts are stored in a file-system-based storage and classification is based on merely syntactical methods (like manual tagging) the accuracy is limited and modelers need to browse, download, and inspect models in order to gain insight.

In Figure 1, a scenario is reported to better illustrate and discuss the limitations of unstructured repositories and to motivate the contributions made by the proposed megamodel-based architecture recovery approach. When dealing with an unstructured repository, the user's ability of looking for artifacts is, on the whole, limited to *full-text search*, *browsing* and, when possible, to the possibility of (internally or externally) *exploring* (*inspecting*) the artifacts. Arguably, this is tedious and error-prone especially if the repository contains a large number of items. In fact, search often relies on simplistic solutions that make use of metadata limited to basic information such as artifact name, textual descriptions, and customized label-based tagging. The outcome is typically a flat collection of seemingly unrelated elements, that fails to convey to the user the structural information about the interrelationships within the repository as, for example, models that are interconnected because of a consistency relationships maintained by a transformation.

Architecture recovery and megamodels to the rescue Megamodeling [13] represents a useful technique to formally characterize the underlying structure of a repository in terms of artifacts categories and their interrelationships. In essence, a megamodel can be viewed as analogous to a schema in a database system, i.e., a representation of the repository structure in terms of the categories of artifacts alongside with their interrelationships. Searching for a specific artifact by specifying its *neighbourhood*, i.e., the network of connections the searched artifact must conform to, allows the modeler to restrict the search space according to her needs and purpose [8]. For example, the user could apply a direct search considering artifacts-specific attributes (or references) and inspect the artifacts that satisfy the search criteria. In a structured repository, a *clustering* function can offer another view of the stored artifacts by grouping together artifacts that are considered *similar* according to a distance measure that is based on structural and lexical similarities. This view can be further refined with a proper *visualization* in which the typing, nature, and relationship between the artifacts are

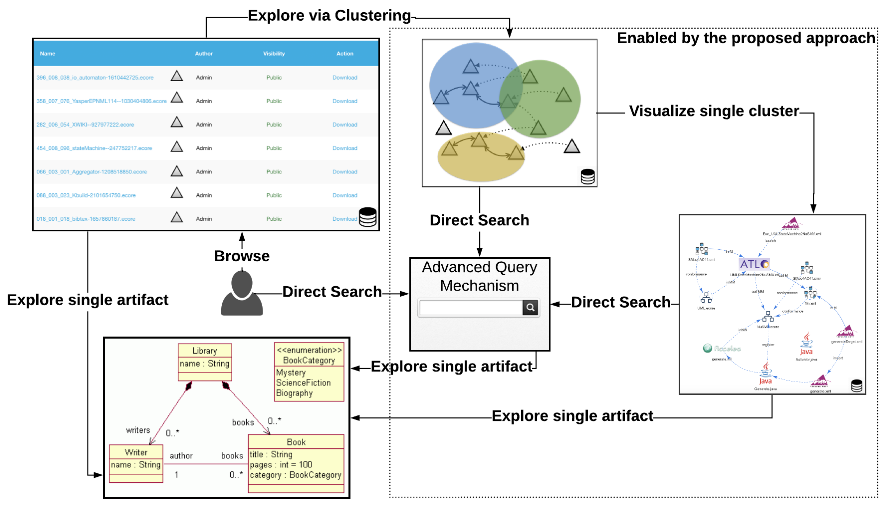


Fig. 1: Application scenario: understanding MDE repositories to prepare reuse.

exposed and the user can decide to proceed in the selection or refining the view by applying a direct search.

The approach proposed in this paper brings suitable megamodels to the table and it also provides an architecture recovery methodology so that structuring repositories does not rely on manual classification and otherwise manual efforts, but the structure is discovered by an extensible set of discovery heuristics inspired by related work in reverse engineering.

In the remainder of this section, we briefly describe some functionality, as pointed out in the figure and as enabled by the approach proposed in this paper:

– search for an artifact,
– clustering of the available model-based artifacts, and
– detailed megamodel-aware visualization.

Searching Modeling Artifacts The availability of efficient and accurate ways to retrieve artifacts in large model repositories is crucial. Thus, relying on sound and well-formed models for discovering and reusing existing artifacts is key to preserving productivity benefits related to model-based processes [59]. The heterogeneity and the multitude of the modeling artifacts stored in a repository require query mechanisms based on a fine-grained level of understanding the repository. For instance, in order to locate an artifact, it might be useful to be able to predicate over artifact types, metamodels, domain types, maturity levels, and metamodel elements, such as classes and structural features, as well as repository-wide attributes [10]. One of the needed prerequisites to enable an efficient way of searching through repositories is having a wide range of supported modeling artifacts. Moreover megamodel-awareness [20] in the repositories contributes to the efficiency of this functionality. Considering the relations among different kinds of artifacts, where relations enable *joins* for traversing the repository is key to success. For instance, consider an illustrative search problem: given a metamodel mm, find all metamodels supported by existing editors that are source metamodels of a transformation that generates models conforming to mm. Using the result of the search, one would be able to create models (instances) of mm rather than relying on existing artifacts in the repository. This kind of search clearly relies on artifact typing (classification) and structure in terms artifact relationships, as they can be represented by a megamodel.

Clustering Modeling Artifacts Most of the potential benefits of having model repositories remain unexploited especially when hundreds or even thousands of modeling artifacts have to be managed. In particular, organizing and browsing models in the available repository are crucial functionalities enabled by an efficient way of typing the stored artifacts. If automatic categorization of the stored artifacts is not supported, this can make the interaction with the repository complex. An efficient way to cluster modeling artifacts permits to automatically organize unstructured repositories and provide the users with overviews of the available artifacts. Clustering is an unsupervised procedure, which automatically organizes artifacts into clusters, where mutually similar artifacts are grouped together depending on a proximity measure the definition of which can be given according to specific search and browsing requirements. These requirements are strongly based on artifact typing and they may also relate to artifacts' relationships (e.g., their existence or their frequency). Thus, megamodel-based architecture recovery for MDE projects also helps clustering.

Visualizing Modeling Artifacts When a subset of artifacts is identified, the user can proceed with a detailed view of the artifacts, in which all the details are reported. By looking at such a representation, users can get a clear understanding about how the different elements are connected and how different artifacts are related. This functionality is explored in detail in Section 5.3.

Explore the Modeling Artifact The individual artifact can be downloaded or inspected directly in the repository.

4 Recovery Methodology

Figure 2 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

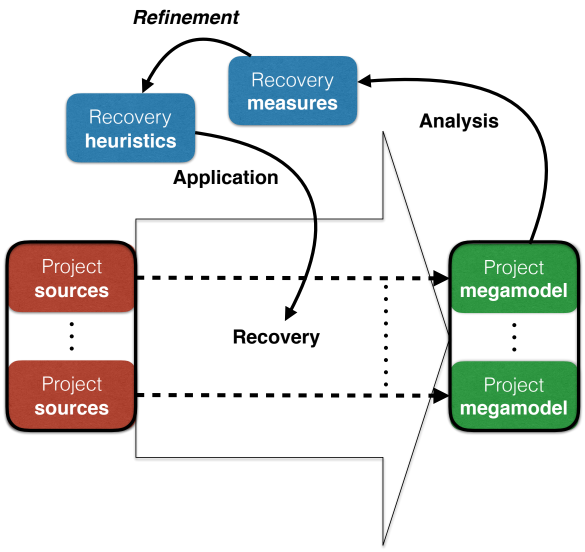


Fig. 2: Megamodel-based reverse engineering.

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.

4.1 Artifacts in a MDE Project

As shown in Fig. 3 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. In particular, an artifact may only be *transient*, for instance, a run-time object during the execution of a test case or otherwise the result of computational step (e.g., due the application of a transformation or a code generator). An artifact may also 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. For instance, an ANT script might contain a path to not-existing files that would be created, if the script is executed, in the sense of the output model of a transformation. *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 transforma-

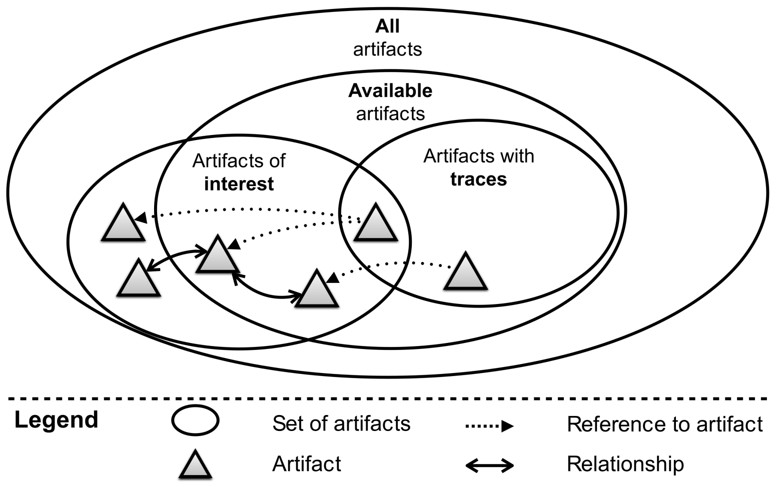


Fig. 3: Artifacts in a MDE project.

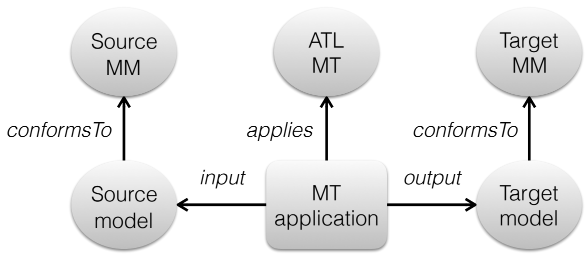
tions as well as metamodels for conformance. *Artifacts with traces* are those (available) artifacts (of interest or not) in which we may locate traces to artifacts (mainly references). Subject to a classification of the artifacts with traces, these artifacts may be interpreted as (encoding) relationships between artifacts.

Finally, the recovery approach may also involve *virtual artifacts*; by this we mean that these artifacts are not really thought to be part of the repository (available or not), but they are computed, much in the sense of transient artifacts, but only for the purpose of discovering artifact types and relationships.

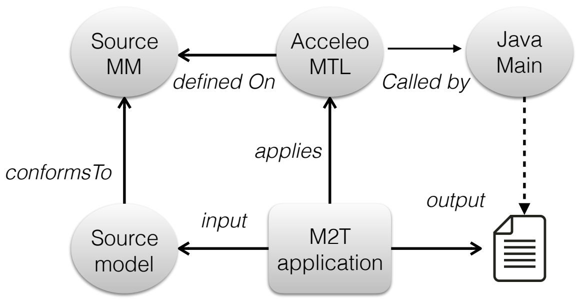
The overall assumption is that we may identify artifacts of interest by examining algorithmically the available artifacts and we may identify relationships between artifacts by examining, again, algorithmically available artifacts on the grounds of technology-specific patterns for traces; we may introduce (in rare cases) virtual artifacts along the way.

4.2 Relationships to be Recovered

Figure 4 identifies 'abstract' artifacts of interest with relationships for the running example of ATL and Acceleo. In particular, in Figure 4a there are source and target models, the corresponding metamodels (MMs), the actual ATL model transformation (MT), and the application thereof. In Figure 4b, the source model is the input, conforming to the source metamodel, and the Acceleo module (M2T) is executed to get the output, which can be any textual format file or code, depending on the target platform. An Acceleo module is usually called by a corresponding main Java file containing references to the module specification.



(a) ATL



(b) Acceleo

Fig. 4: ‘Abstract’ artifacts and relationships for ATL vs Acceleo usage

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:

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). Also converting an artifact into a different format or technological space can give rise to discovered relationship among artifacts, e.g., injecting from a concrete syntax to a model.

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).

4.3 Heuristics for Recovery

We will discuss now heuristics for identifying artifacts of interest and finding traces for relationships. These heuristics adopt techniques that have been applied elsewhere in a more classic reverse engineering context or in the validation of prescriptive megamodels or yet other contexts of software engineering; we provide related work pointers on the way. Our original contribution is the adoption and combination of a number heuristics for the recovery of megamodels, i.e., a synthesis of the areas of reverse engineering, megamodeling, and architecture recovery.

Filename heuristics Many types of artifacts may be precisely detected on the grounds of filenames or extensions thereof [25]. For instance, the '.atl' extension identifies an ATL model transformation — especially within an MDE project, and the '.mtl' extension stands for Acceleo module — from Acceleo transformation language. Clearly, filenames may not always be sufficient, for instance mtl is also used for files used by 3D object editing applications; 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 [25,44] or by means of techniques resembling the magic number-based detection in the UNIX *file* command[2]. For instance, a syntax definition for the EMFText technology would be a '.cs' file that contains the string 'syntaxdef' [44]. (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 [52].

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 [52,31]. For instance, a suspected conformance relationship may be verified by the available component (operation) for Ecore-based conformance checking, as discussed in Section 4.2.

Extractor heuristics Customized fact extractors [52,58,27] may be used to identify traces in given artifacts, thereby helping with recovery of relationships. For instance, a heuristic for ANT files may extract instances of common patterns of using ANT for applying model transformations. Such extraction may involve virtual artifacts at times.

[2] `https://pubs.opengroup.org/onlinepubs/9699919799/utilities/file.html`

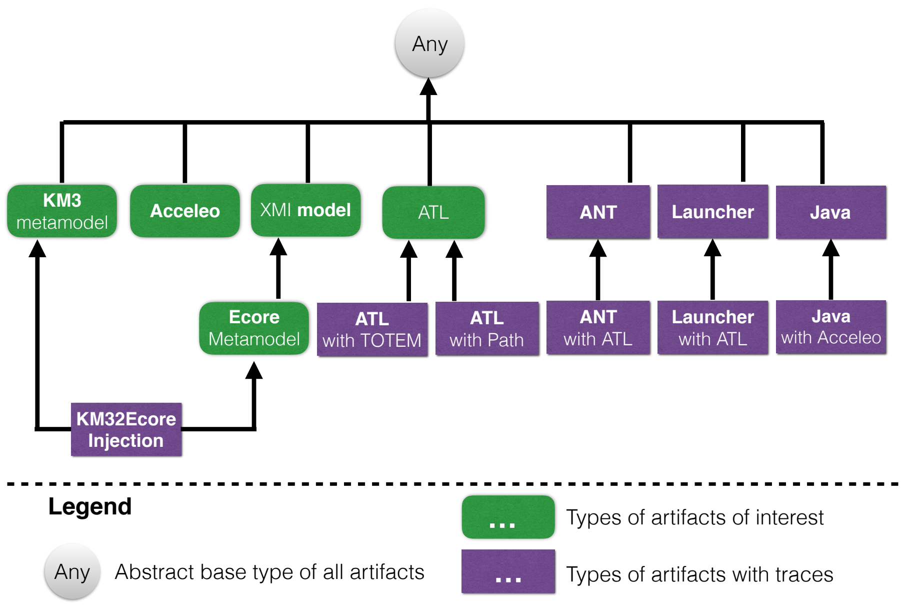


Fig. 5: Artifacts involved in the recovery of the considered artifacts.

Analyzer 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 [19], thereby preparing the detection of potential source or target models on the grounds of attempted conformance checking. Such analyses may involve virtual artifacts at times.

For purposes of illustration, Figure 5 arranges some of the heuristics that were developed in the case study of Section 6, also used to create the visualization for the running example in section 5.3. The root node is ‘abstract’; it does not correspond to any actual heuristic. 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. 5 are introduced in an iterative process on the grounds of measuring connectivity of the recovered graph and leveraging domain knowledge (regarding MDE technologies) for identifying opportunities for relationship recovery by additional heuristics.

5 The Recovery Infrastructure

The ability of analyzing projects (and systems in general) to identify their components and their interrelationships is key to obtaining representations at a higher level of abstraction that can support recovery processes. Then, applying automated practices to the maintenance and enhancement of existing projects lies in the capability

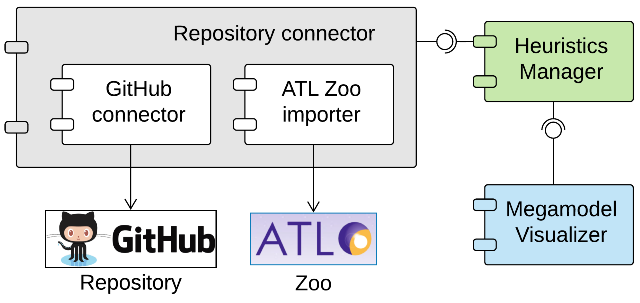


Fig. 6: Components of the recovery architecture.

of employing reverse-engineering approaches as described in the previous section. In this section, the recovery infrastructure supporting the methodology presented in the previous section is presented. As shown in Fig. 6, the recovery procedure consists of three main components, i.e.:

– `RepositoryConnector`,
– `HeuristicsManager`, and
– `MegamodelVisualizer`.

The `RepositoryConnector` component associates data sources that export reusable MDE projects, which can then be locally downloaded for subsequent analysis. Currently, the recovery infrastructure can import data from the ATL Zoo and GitHub repositories (see Section 5.1). `RepositoryConnector` is extensible and provides developers with interfaces that can be implemented for adding new connectors. The `HeuristicsManager` component is responsible for applying the available heuristics on all the projects that have been locally downloaded by the `RepositoryConnector`. The outcome of the recovery process consists of models conforming to a specifically conceived metamodel as presented in Section 5.2. The outcome of `HeuristicsManager` can be consumed in different ways — including the possibility of graphically presenting it in order to give a more intuitive overview of the analyzed projects and to support the analysis and understanding of the contained artifacts. The `MegamodelVisualizer` component presented in Section 5.3 takes recovery models as input and generates a graphical representation of them.

5.1 Repository Connector

This component retrieves projects from online repositories. Currently, it can import projects from the ATL Zoo and GitHub repositories.Extraction from ATL Zoo required coverage of ATL transformations, XMI models, metamodels and other supplementary files. Extration from GitHub repositories made us additionally cover, for instance, Acceleo projects. The ATL Zoo is a well-known repository of model transformations, which have been the subject of several empirical works over the last few

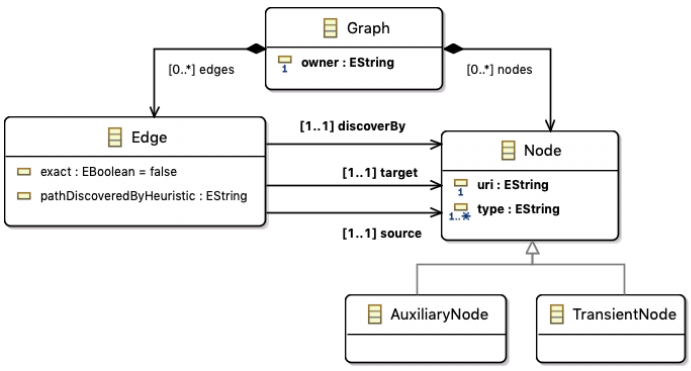


Fig. 7: Recovery metamodel

years. Unfortunately, ATL Zoo 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[3] for locally cloning a project of interest identified by its `owner` and `name` attributes. Additional repositories can be considered by extending the connector by means of its extension API.

5.2 Heuristics Manager

Once data have been retrieved by means of the existing connectors, the actual recovery process starts. The outcome of the process is a model conforming to a specifically devised *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*. The corresponding metamodel is presented in Fig. 7. For each artifact that can be identified by the heuristics, the recovery approach generates a corresponding target node.

The generated node can be a concrete *Node*, if the artifact corresponds to a physical file in the project; it can be a *TransientNode* if it is derived from an existing artifact, but it is not part of the project before the recovery process. The instantiation of this type of nodes is normally deferred to later stages when the information becomes available, e.g., an output model of a transformation that is declared in an ANT script, but it will be produced only if the transformation is executed. Finally, a node can be typed *AuxiliaryNode* if it is temporally created during the recovery process in the sense of *virtual* nodes of Section 4.1. Thus, auxiliary nodes are not actually part of the analyzed project and consequently are not relevant for the megamodel structure being created. A clarifying example is shown later.

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

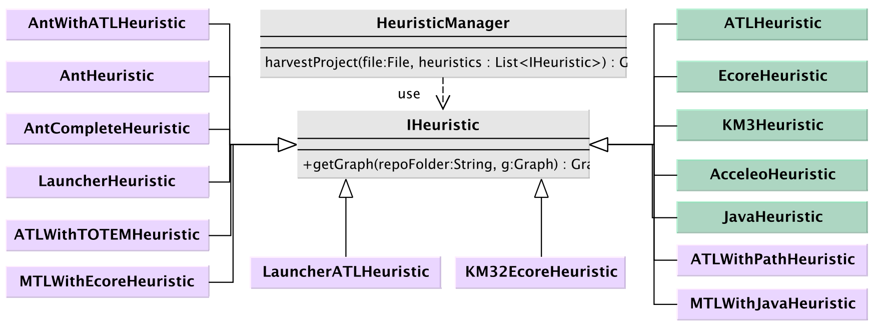


Fig. 8: Class diagram showing an overview of the Heuristic Manager.

An important aspect of the recovery process is the ability of detecting and representing relationships among artifacts. The 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, and two edges would link the transformation with two further nodes representing the source and target metamodels.

Recovery heuristics Figure 8 shows a class diagram representing the hierarchical organization of the heuristics currently available in the `HeuristicsManager` component shown in Fig. 6. Each heuristic implements the `Heuristic` interface or extends an available implementation. In Fig. 8, the elements `ATLHeuristic`, `EcoreHeuristic`, `KM3Heuristic`, `AcceleoHeuristic`, and `JavaHeuristic` are shown in green color in order to be consistent with the organization of Fig. 5.

Listing 1: Fragment of ATLHeuristic

```
package it.univaq.MDEProfiler.heuristic;
...
public class ATLHeuristic implements IHeuristic {
  private String extension = ".atl";
  private String nodeKind = "NodeType.ATL";
  @Override
  public Graph getGraph(String repoFolder, Graph g){
    File repoFolderF = new File(repoFolder);
    List<File> fList = FileUtils.getFilesByEndingValue(repoFolderF,extension);
    for (File file : fList) {
      boolean guard = g.getNodes().stream()
        .anyMatch(s -> s.getUri().equals(file.getAbsolutePath()));
      if(!guard) {
        Node n = GraphFactory.eINSTANCE.createNode();
        n.setDerivedOrNotExists(false);
        n.getType().add(nodeKind);
        n.setUri(file.getAbsolutePath());
        n.setName(file.getName());
        g.getNodes().add(n);
      }
    }
    return g;
  }
}
```

These heuristics identify artifacts of interest, i.e., ATL transformations and metamodels specified either in KM3 or Ecore, or Acceleo modules with the corresponding source metamodel. Heuristics that are shown in Fig. 8 with the violet color represents heuristics that have been implemented in order to recover relationships among transformations, Acceleo artifacts, models, and metamodels.

Listing 1 shows a fragment of the Java implementation of *ATLHeuristic*. Essentially, in each project the heuristic searches for files with extension .atl (see line 5), and for each of them a new node typed `NodeType.ATL` is generated in the target recovery model (see lines 13-20). The heuristic can be easily adapted to cover other kinds of artifacts by properly specifying the file *extension* to be considered (e.g., .km3, .mtl, .ecore) and the corresponding *node kind*.

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 a fragment of the ANT file launcher contained in the ATL project *UMLStateMachine2NuSMV*[4] and named `Exe_UMLStateMachine2-NuSMV.xml`. Lines 16-19 contain precious information about the input and target elements of the ATL *UMLStateMachine2NuSMV* transformation, which if considered alone, does not contain such details. In particular, in this case a transient node will be generated for the output model *SMac4AC41.nusmvmodel* (see lines 4 and 22), since it is a node that will be instantiated only once the transformation gets executed. The analysis of ATL launch configuration files, as the one shown in Listing 2, is implemented by the `AntWithATLHeuristic` given in Fig. 8.

Listing 2: Fragment of the ANT file launching the *UMLStateMachine2NuSMV* transformation

```
1
2  <project name="ATL2Metrics" default="extract" basedir=".">
3    <!-- Set paths -->
4    <property name="inputname" value="SMac4AC41"></property>
5    ...
6    <target name="extract">
7      <atl.loadModel modelHandler="EMF" name="UML" metamodel="%EMF"
8                  nsURI="http://www.eclipse.org/uml2/4.0.0/UML">
9      </atl.loadModel>
10     <!-- Define metamodels/models-->
11     <atl.loadModel modelHandler="EMF" name="UMLModel"
12             metamodel="UML" path="${inputfile}">
13     </atl.loadModel>
14     <atl.loadModel metamodel="%EMF" name="NuSMV" path="${mmdir}/NuSMV.ecore">
15     </atl.loadModel>
16     <!-- Execute transformation -->
17     <atl.launch path="UMLStateMachine2NuSMV.asm" refining="false">
18       <inModel name="IN" model="UMLModel"></inModel>
19       <outModel name="OUT" model="out" metamodel="NuSMV"></outModel>
20     </atl.launch>
21     <!-- Generate output model -->
22     <atl.saveModel model="out" path="${outputdir}/${inputname}.nusmvmodel">
23     </atl.saveModel>
24   </target>
25 </project>
```

To improve the coverage of ATL projects, the additional heuristic named `ATLWith-TOTEMHeuristic` has been developed by relying on the techniques presented

[4] Project imported from `https://github.com/kiyo07/UMLStateMachine2NuSMV`

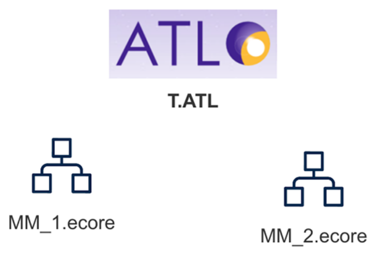
Listing 3: Fragment of *ATLWithTOTEMHeuristic* for discovering metamodels related to a given transformation

```
1  public Graph getGraph(String repoFolder, Graph g) {
2    for (Node atlNode : g.getNodes().stream().filter(z -> z.getType()
3        .contains(FileUtils.ATL)).collect(Collectors.toList())) {
4      ReduceRequirementMetamodels rrmm = new ReduceRequirementMetamodels();
5      AuxiliaryNode drmIN = GraphFactory.eINSTANCE.createNode();
6      drmIN.setDerivedOrNotExists(false);
7      drmIN.getType().add(FileUtils.DRM);
8      drmIN.setUri(rrmm.generateRMM(atlNode.getUri()));
9      g.getNodes().add(drmIN);
10
11     ...
12     for (Node n1 : g.getNodes().stream().
13             filter(z -> z.getType().contains(FileUtils.ecoreKind)).
14             collect(Collectors.toList()))
15       if (rrmm.checkDRMConformance(drmIN.getUri(), n1.getUri())) {
16         Edge edge = GraphFactory.eINSTANCE.createEdge();
17         edge.setSource(atlNode);
18         edge.setTarget(n1);
19         edge.setName(FileUtils.sourceDRM);
20         g.getEdges().add(edge);
21         ...
22       }
23     g.getNodes().remove(drmIN);
24     ...
25   }
26   return g;
27 }
```

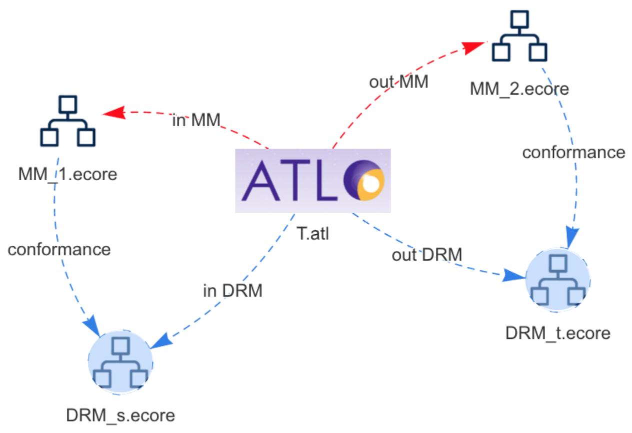
in [20] and on the TOTEM tool [54], which is built on a method able to extract a typing requirements model (TRM) from an ATL transformation. A TRM describes the requirements that the transformation needs from the source and target meta-models in order to obtain a transformation with a syntactically correct typing. A TRM is extracted from the transformation and it generates domain typing requirements models (DRMs), describing the requirements for the source and target meta-models. A TRM is a good example of a virtual artifact, as of Section 4.1; it is computed merely to facilitate relationship recovery between transformations, metamodels, and models. Additional relationships can be derived by checking the conformance between existing metamodel nodes in the graph and the *auxiliary* nodes related to the generated DRMs. If the conformance checking succeeds then an additional relationship among the transformation input of the TRM and the metamodel subject of the conformance check can be created.

Listing 3 shows an excerpt of `ATLWithTOTEMHeuristic`: for each transformation (line 3) in the project, the heuristic generates the source and target DRMs, added as *AuxiliaryNode*s to the graph (lines 4-9). For each metamodel (line 14) in the graph the heuristic checks the conformance with the generated DRMs (line 15). If the conformance check is positive, the heuristic adds a link between the analyzed transformation and the metamodel (lines 16-20).

Figure 9 reports an explanatory application of *ATLWithTOTEMHeuristic*. In particular, Figure 9a shows the *T.atl* node and two unrelated metamodels, highlighting that the two metamodel nodes are dangling. By applying the heuristic described



(a) Before ATLWithTOTEM Heuristic



(b) Applying ATLWithTOTEM Heuristic

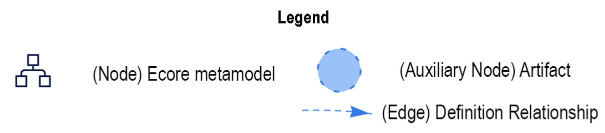


Fig. 9: Explanatory application of *ATLWithTOTEMHeuristic*

above, two *AuxiliaryNode*s are produced, namely DRM_s and DRM_t from the transformation *T.atl* (see Fig. 9b). If the conformance check is positive (as in this example) the two dangling nodes MM_1 and MM_2 can be linked to the transformation (as highlighted in red in Figure 9.b). Thus, the auxiliary nodes and their incoming relations are subsequently removed.

To refine further the precision of the proposed infrastructure another heuristic has been implemented, called `KM32EcoreHeuristic`. This heuristic is based on an injection from a KM3 metamodel to Ecore with the intent of further reducing the number of dangling nodes. KM3 (Kernel MetaMetaModel) [36] is a DSL for describing metamodels with a specific textual notation that should enhance the agility and precision in defining metamodels. Many projects still use KM3 as metamodeling language, with the result of making the ATL transformations runnable only if these models are converted to Ecore. Thus, for such projects dangling nodes would be generated because of the alternative KM3 versions of Ecore metamodels. By using

`KM32EcoreHeuristic`, it is possible to link a KM3 dangling node to the Ecore metamodel resulting from the injection, and then if the resulting injected metamodel overlaps with one of the existing metamodels then a new relation can be created and labelled as *inject*. The implementation of `KM32EcoreHeuristic` is based on the *KM32Ecore injector* tool[5] and on EMFCompare[6]. The result of the application of this heuristic on an explanatory example is shown in Fig. 10.

The initial scenario is represented in Fig. 10a), where two km3 nodes are dangling and a transformation has two relations to in and out metamodels. Applying the injection from km3 to Ecore in Fig. 10b resulted that the node *injMM*_1 (auxiliary) and *MM*_1 are overlapping, as result of the comparison (same for the other two nodes). For this reason, the result reported in Fig. 10c leveraged a new type of relationship, *inject*, thereby representing that *MM_1* and *injMM_1* represent the same metamodel in different formats.

The support for Acceleo modules has been introduced by implementing two dedicated heuristics i.e., `MTLWithJavaHeuristic` and `MTLWithEcoreHeuristic`. The former creates relationships among Acceleo templates and the corresponding Java files responsible of their execution. The latter creates relationships among Acceleo templates and the Ecore metamodels typing them. A fragment of `MTLWithJava-Heuristic` is shown in Listing 4. `MTLWithEcoreHeuristic` is not shown due to the sake of brevity. However, it essentially checks if the Ecore metamodel referred by the considered Acceleo template exists in the analyzed project; if yes a dedicated relationship is created in the megamodel being produced.

For brevity, this paper does not give more details about the implementation of all the currently available heuristics. Readers are referred to the Github[7] repository to download and play with the tool supporting the proposed approach. To give some numbers related to MDEPROFILER, Table 1 shows the number of heuristics and the lines of code of the tool as proposed in [20] and as developed for this paper.

	#Heuristic Implementations	**Loc**
MDEPROFILER as proposed [20]	11	880
MDEPROFILER proposed in this paper	20	1446

Table 1: Heuristic implementations

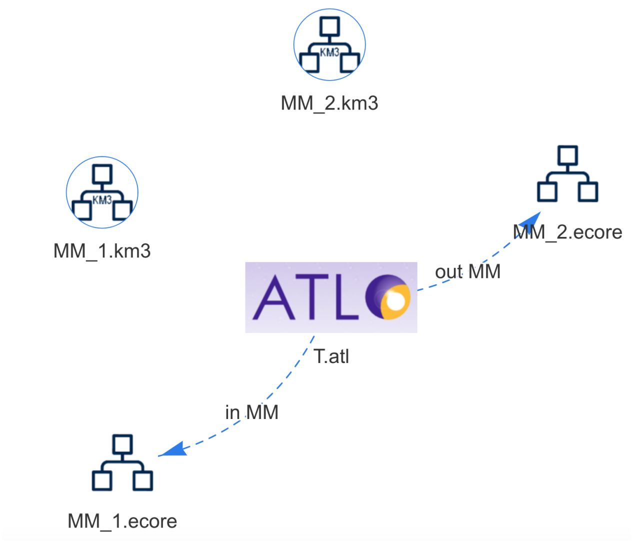
5.3 Megamodel Visualizer

The recovered model generated for the input projects can be processed by other services, for example, to graphically represent projects imported by `https://github.`

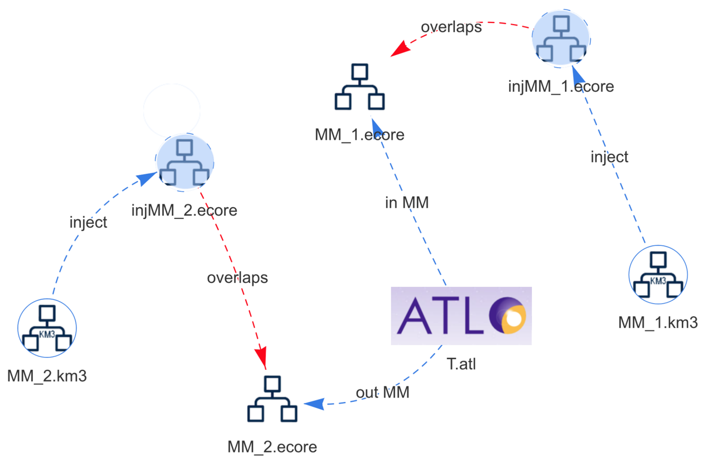
[5] Km3 to Ecore injector project: `https://github.com/atlanmod/EMFTVM-D/tree/master/deprecated/org.atl.eclipse.km3/src/org/atl/eclipse/km3`

[6] EMFCompare:`https://www.eclipse.org/emf/compare/`

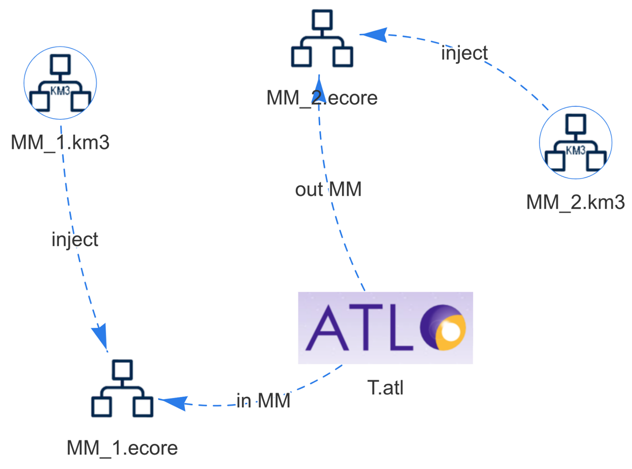
[7] Our project developed to support the methodology: `https://github.com/MDEGroup/MDEProfiler`



(a) Before the application of *KM32EcoreHeuristic*



(b) Application of *KM32EcoreHeuristic*



(c) After the application of *KM32EcoreHeuristic*

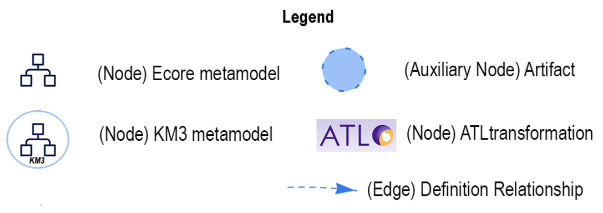


Fig. 10: Application of *KM32EcoreHeuristic*

Listing 4: Fragment of *MTLWithJavaHeuristic* for discovering Java links to Acceleo template

```java
package it.univaq.MDEProfiler.heuristic;
public class MTLWithJavaHeuristic implements IHeuristic {
  @Override
  public Graph getGraph(String repoFolder, Graph g) {
    this.g = g;
    for (Node n : g.getNodes().stream().
                filter(z -> z.getType().contains(FileUtils.MTLKind)).
            collect(Collectors.toList()))
        for (Node n1 : g.getNodes().stream().
                    filter(z -> z.getType().contains(FileUtils.JavaKind)).
                collect(Collectors.toList()))
          if(getMTL(n1.getUri()).equals(n.getName()))
            g.getEdges().add(createEdge(n, n1););
    return g;
  }
  /** Get MTL name from file Java **/
  private String getMTL(String path) throws FileNotFoundException {
      InputStream inputStream = new FileInputStream(path);
          CompilationUnit cu = JavaParser.parse(inputStream);
          List<com.github.javaparser.ast.Node> node_list = cu.getChildNodes();
          String result = "";
          int i = 0;
          while(true){
            com.github.javaparser.ast.Node node = node_list.get(i);
            String main_string = node.toString();
              if(main_string.indexOf("MODULE_FILE_NAME") !=-1 &&
                      main_string.indexOf("TEMPLATE_NAMES") !=-1){
                        ...
                  result = new String(main_string));
              }
            i = i + 1;
          }
      return result;
  }
}
```

com/kiyo07/UMLStateMachine2NuSMV and automatically recovered as shown in Fig. 11b. The same visualization has been used to show the examples in section 4.3. By looking at such a model, users can get a clear understanding about how the different elements are connected. By contrast, Fig. 11a shows the folders contained in the package of the considered projects, as users could explore the projects 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. In this case the first project contains an Acceleo template, and the last one an ATL transformation. Additional artifacts are stored in the other folders, e.g., ANT scripts, models and metamodels. The *MegamodelVisualizer* component shown in Fig. 6 is in charge of generating diagrams like the one shown in Fig. 11b by means of an Acceleo[8]-based generator; it takes a recovery model as input and generates HTML5+Javascript code. The generated code uses the Visjs[9] 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. 11b, the artifact

[8] https://www.eclipse.org/acceleo/
[9] http://visjs.org

`NuSMV.ecore` is visually associated with the Ecore type (see Legend) and the link with the artifact `SMac4AC41.smv` highlights that the latter is a model conforming to the former.

Moreover, the ATL transformation `UMLStateMachine2NuSMV.atl` takes as input the `SMac4AC41.uml` node as model and the `UML.ecore` *TransientNode* element as metamodel. This last element is part of the artifacts, since it can be discovered by the `Exe_UMLStateMachine2NuSMV.xml` ANT file, but there is no concrete file to be discovered in the projects (since in this case it is referred by the nsURI of the metamodel). The output consists of the `SMac4AC41.nusmvmodel` model conforming to the `NuSMV.ecore` metamodel. The node `generate.xml` as in the previously discussed example, contributes to the discovery of the represented relationship between *Generate.java* and the `NuSMV.ecore` metamodel as shown by the hovering label *discovered by*. The `generate.mtl` Acceleo template has been discovered by the Acceleo-specific heuristics previously presented. In particular, the node related to `generate.mtl` is pointed by `Generate.java`, which specifies in the variable `MODULE_FILE_NAME` (see Listing 5) the Acceleo template to be invoked, as detailed in the description of Listing 4.

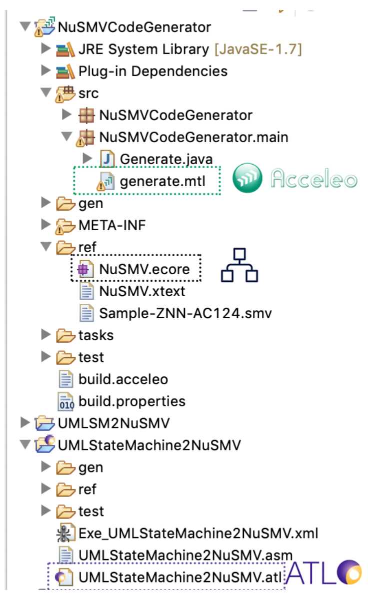
Listing 5: Fragment of the file *Generate.java*

```
...
public static final String MODULE_FILE_NAME = "/NuSMVCodeGenerator/main/generate";
...
```

Finally, as shown in Fig. 11b, the considered project contains also two ANT scripts, i.e, `generateTarget.xml` importing `generate.xml`, to automate the execution of the ATL tranformation, which takes `file.xmi` as input model, conforming to `NuSMV.ecore`. The node related to `file.xmi` is a transient node, since it is not physically stored in the project, but it is specified in the ANT script, and it will be generated by the tool chain. Very likely this node is corresponding to the `SMac4AC41.smv` model, but it is still not discoverable by the current implemented heuristics, leading to the future plan to implement a new heuristic able to find correspondences among transient and concrete nodes, like in the case of these two models. Using model to code transformations for generating a visualization is a fast way to provide modelers with a simple though effective visualization of the produced models. However, such a generative step can be replaced by the implementation of a more sophisticated diagrammatic editor, which would allow the user to interact with the recovered models. We intent to pursue this as a future work.

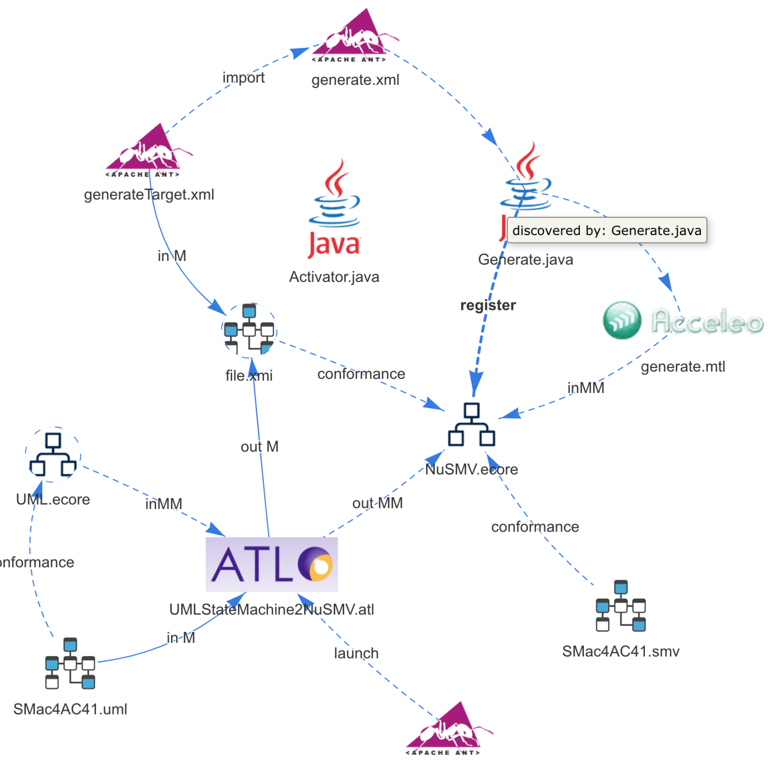
6 Experimental evaluation

This section discusses the evaluation of the proposed approach by considering two datasets that have been defined by starting from an initial set of ≈100 ATL projects retrieved from the ATL Zoo. In particular, such set has been pruned by discarding those projects containing transformations that were not syntactically correct. The resulting dataset, named ATL_L hereafter, consists of 85 transformation projects and it has been used to evaluate the approach in terms of dangling nodes that the available



```
▼ NuSMVCodeGenerator
    ▶ JRE System Library [JavaSE-1.7]
    ▶ Plug-in Dependencies
    ▼ src
        ▶ NuSMVCodeGenerator
        ▼ NuSMVCodeGenerator.main
            ▶ Generate.java
              generate.mtl
    ▶ gen
    ▶ META-INF
    ▼ ref
        NuSMV.ecore
        NuSMV.xtext
        Sample-ZNN-AC124.smv
    ▶ tasks
    ▶ test
      build.acceleo
      build.properties
▶ UMLSM2NuSMV
▼ UMLStateMachine2NuSMV
    ▶ gen
    ▶ ref
    ▶ test
      Exe_UMLStateMachine2NuSMV.xml
      UMLStateMachine2NuSMV.asm
      UMLStateMachine2NuSMV.atl
```

(a) Imported projects shown with Project Explorer



(b) Recovered graph shown with the Visualizer

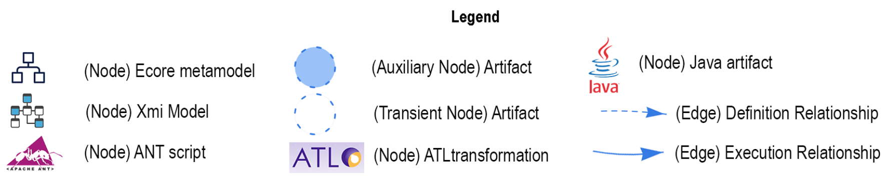

Legend
(Node) Ecore metamodel
(Node) Xmi Model
(Node) ANT script
(Auxiliary Node) Artifact
(Transient Node) Artifact
(Node) ATLtransformation
(Node) Java artifact
(Edge) Definition Relationship
(Edge) Execution Relationship

Fig. 11: Project Explorer vs Visualizer

Table 2: Evaluation results

Iteration	Applied Heuristics	#Nodes	#Edges	#Dangling Nodes
1	EH	324	0	324
2	EH, AH	546	0	546
3	EH, AH, KH	735	0	735
4	EH, AH, KH, LH	817	0	817
5	EH, AH, KH, LH, ANH	916	0	916
6	EH, AH, KH, LH, ANH, APH	916	37	880
7	EH, AH, KH, LH, ANH, APH, LTH	948	212	844
8	EH, AH, KH, LH, ANH, APH, LTH, ANATLH	1105	831	709
9	EH, AH, KH, LH, ANH, APH, LTH, ANATLH, JH	1210	831	709
10	EH, AH, KH, LH, ANH, APH, LTH, ANATLH, JH, TOTEMH	1210	1039	626
11	EH, AH, KH, LH, ANH, APH, LTH, ANATLH, JH, TOTEMH, KM3ECOREH	1210	1112	456

Legend: EH: EcoreHeuristic, AH: ATLHeuristic, KH: KM3Heuristic,
LH: LauncherHeuristic, ANH: ANTHeuristic, APH: ATLWithPathHeuristic,
LTH: LauncherATLHeuristic, ANATLH: ANTWithATLHeuristic,
JH: JavaHeuristic, TOTEMH: ATLWithTOTEMHeuristic,
KM3ECOREH: KM32ECOREHeuristic

heuristics are able to remove. In particular, the evaluation 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. To evaluate the accuracy of the approach, we randomly extracted a smaller dataset consisting of 40 transformation projects (named ATL_S hereafter) out of ATL_L with the aim of measuring precision and recall. Overall, the performed evaluation addresses the following research questions:

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

Results Table 2 shows representative results related to each iteration of the performed evaluation on the dataset ATL_L. 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. After the first 8 iterations we were able to reduce the number of dangling nodes to 709. Introducing additional heuristics corresponding to the last three iterations considerably decreased that number from 709 to 456, i.e., from 58.59% to 37.68% of the total number of discovered nodes.

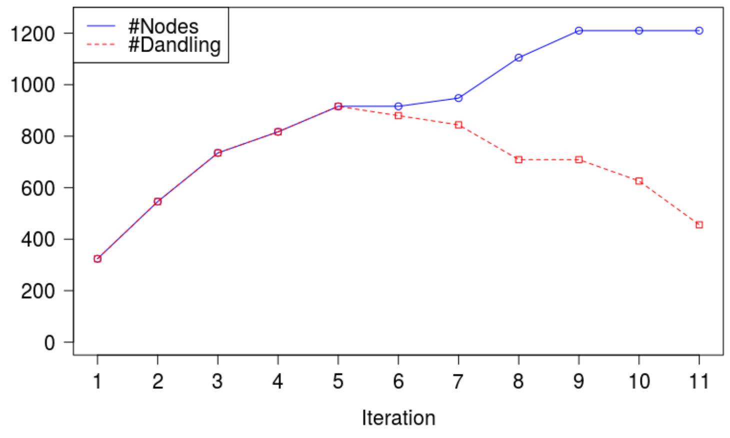


Fig. 12: Nodes recovered during the evaluation

Figure 12 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.

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

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

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.

To evaluate the accuracy of the approach and thus, to answer *RQ1*, we manually analysed the dataset ATL_S. In particular, a senior modeler manually inspected such projects (without knowing in advance the results of the tools) and recovered the nodes and relations in All_m of the corresponding megamodels[10].

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.

The overall accuracy can be increased by means of adding heuristics. For instance, the analysed projects contain TCS specifications [37], which are currently not

[10] 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/MDEProfiler

	Nodes	**Relations**
Precision	0.925	0.908
Recall	0.942	0.726

Table 3: Precision and recall of recovery.

covered by MDEPROFILER and this is reflected by the *precision* and *recall* measures. To answer *RQ2*, the dataset ATL_S has been analysed by means of MDEPROFILER executed on an Intel Core i5 machine with 8GB of RAM. The analysis took about 15 seconds, whereas the senior modeler needed 1.5 full-time working days to perform the analysis on the same data set. The resulting precision and recall are shown in Table 3. It is important to remark that even though effort reduction has been measured in terms of precision and recall, additional aspects could be also considered. For instance, the cognitive efforts that are needed to understand the produced modeling artifact network could be also taken into account in the evaluation. However, we consider this aspect out of the scope of this paper.

7 Conclusion and Future Work

MDE projects are typically shared without any higher-level descriptions serving understanding of classification, data flow, conformance, and other properties and relationships of the involved artifacts. Much of such classification information and relationships are encoded in some idiosyncratic manner or lost in persisting projects in a repository or otherwise subject to analysis or inference. That is, projects are given as packages consisting of files, possibly organized in folders that modelers have to manually explore 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, thereby severely limiting reuse of MDE projects.

In this paper, we presented an approach based on megamodels and inspired by the notion of architecture recovery which enables the model-based recovery of 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 on the widely used ATL Zoo consisting of about 100 model transformation projects.

In future work, we plan to apply the approach to other corpora of projects, for instance, a corpus with Acceleo projects. We also plan to implement additional heuristics, as needed in order to minimize further the number of dangling nodes in megamodels and to improve the overall accuracy of the approach. We are also working on extending the portfolio of MDE technologies beyond the current coverage of ATL and Acceleo by, for example, including textual concrete syntax definitions and related models. Furthermore, we expect to make the discovery methodology more systematic, for example, in terms of tracking not just all known nodes and the dangling nodes, but also paying full attention to our ability of explaining the relevance, if any, of any artifact in a given (MDE) project, perhaps even before assigning a node type. In this direction, it would be interesting to explore, inspired by the KM3-based heuristic, if other similarity-based heuristics might help the modeler to identify versioning in metamodeling, and then enable coupled-evolution management in the discovered megamodels. It is important to notice that the design of the proposed recovery approach caters for the introduction and execution of additional heuristics without disrupting the overall eco-system. In particular, we would not want to refactor MDE projects or enforce new best practices for the benefit of megamodel-based recovery of artifact typing and relationships. In this manner, we hope to increase the probability of community projects for the development of additional heuristics to be integrated eventually into a reusable infrastructure.

References

1. Z. Alshara, A. Seriai, C. Tibermacine, H. Bouziane, C. Dony, and A. Shatnawi. Materializing Architecture Recovered from Object-Oriented Source Code in Component-Based Languages. In *Proc. ECSA*, volume 9839 of *LNCS*, pages 309–325. Springer, 2016.
2. G. Antoniol, G. Canfora, G. Casazza, and A. D. Lucia. Information Retrieval Models for Recovering Traceability Links between Code and Documentation. In *ICSM*, pages 40–49. IEEE, 2000.
3. H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *ICSE (1)*, pages 95–104. ACM, 2010.
4. Ö. Babur, L. Cleophas, and M. van den Brand. Model analytics for feature models: case studies for S.P.L.O.T. repository. In *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018.*, pages 787–792, 2018.
5. F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio. Mdeforge: an extensible web-based modeling platform. In *CloudMDE@MoDELS*, pages 66–75, 2014.
6. F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. *Automated Clustering of Metamodel Repositories*, pages 342–358. Springer International Publishing, 2016.
7. F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. A customizable approach for the automated quality assessment of modelling artifacts. In *Quality of Information and Communications Technology (QUATIC), 2016 10th International Conference on the*, pages 88–93. IEEE, 2016.
8. F. Basciani, J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Model Repositories: Will They Become Reality? In *Proc. CloudMDE@MoDELS 2015*, volume 1563 of *CEUR Workshop Procs*, pages 37–42, 2016.
9. F. Basciani, D. Di Ruscio, L. Iovino, and A. Pierantonio. Automated chaining of model transformations with incompatible metamodels. In *MODELS*, pages 602–618, 2014.
10. F. Basciani, J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio. Exploring model repositories by means of megamodel-aware search operators. In *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018.*, pages 793–798, 2018.
11. F. P. Basso. A proposal for a common representation language for mde artifacts and settings. In *STAF Doctoral Symposium*, 2015.
12. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. 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*, pages 33–46. Springer, 2005.
13. J. Bézivin, F. Jouault, and P. Valduriez. On the need for Megamodels. In *Proc. of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop*, 2004.
14. C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devanbu. The promises and perils of mining git. In *MSR*, pages 1–10. IEEE Computer Society, 2009.

15. I. T. Bowman and R. C. Holt. Software architecture recovery using Conway's law. In *Proc. CASCON*, page 6. IBM, 1998.
16. M. Capraro, M. Dorner, and D. Riehle. The patch-flow method for measuring inner source collaboration. In *MSR*, pages 515–525. ACM, 2018.
17. S. Chardigny and A. Seriai. Software Architecture Recovery Process Based on Object-Oriented Source Code and Documentation. In *Proc. ECSA*, volume 6285 of *LNCS*, pages 409–416. Springer, 2010.
18. E. Constantinou, G. Kakarontzas, and I. Stamelos. An automated approach for noise identification to assist software architecture recovery techniques. *JSS Journal*, 107:142–157, 2015.
19. J. de Lara, J. Di Rocco, D. Di Ruscio, E. Guerra, L. Iovino, A. Pierantonio, and J. S. Cuadrado. Reusing Model Transformations Through Typing Requirements Models. In *Proc. FASE*, volume 10202 of *LNCS*, pages 264–282. Springer, 2017.
20. J. Di Rocco, D. Di Ruscio, J. Härtel, L. Iovino, R. Lämmel, and A. Pierantonio. Systematic recovery of mde technology usage. In A. Rensink and J. Sánchez Cuadrado, editors, *Theory and Practice of Model Transformation*, pages 110–126, Cham, 2018. Springer International Publishing.
21. J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Collaborative Repositories in Model-Driven Engineering. *IEEE Software*, 32(3):28–34, 2015.
22. R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431. IEEE Computer Society, 2013.
23. R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34, 2015.
24. R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *ICSE*, pages 779–790. ACM, 2014.
25. J. Favre, R. Lämmel, M. Leinberger, T. Schmorleiz, and A. Varanovich. Linking Documentation and Source Code in a Software Chrestomathy. In *Proc. WCRE*, pages 335–344. IEEE, 2012.
26. J. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proc. MODELS*, volume 7590 of *LNCS*, pages 151–167. Springer, 2012.
27. R. Ferenc, I. Siket, and T. Gyimóthy. Extracting Facts from Open Source Software. In *Proc. ICSM*, pages 60–69. IEEE, 2004.
28. R. France, J. Bieman, and B. H. C. Cheng. Repository for model driven development (remodd). In T. Kühne, editor, *Models in Software Engineering*, pages 311–317, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
29. F. D. Giraldo, S. España, W. J. Giraldo, and O. Pastor. Evaluating the quality of a set of modelling languages used in combination: A method and a tool. *Inf. Syst.*, 77:48–70, 2018.
30. T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *IWPSE*, pages 113–122. IEEE Computer Society, 2005.
31. J. Härtel, L. Härtel, M. Heinz, R. Lämmel, and A. Varanovich. Interconnected Linguistic Architecture. *The Art, Science, and Engineering of Programming Journal*, 1, 2017. 27 pages.
32. J. Härtel, M. Heinz, and R. Lämmel. Emf patterns of usage on GitHub. In *Proc. ECMFA*, LNCS. Springer, 2018. To appear.
33. A. E. Hassan and R. C. Holt. Architecture recovery of web applications. In *Proc. ICSE*, pages 349–359. ACM, 2002.
34. M. Heinz, R. Lämmel, and A. Varanovich. Axioms of linguistic architecture. In *Proc. MODELSWARD*, pages 478–486. SCITEPRESS, 2017.
35. A. Janes, D. Piatov, A. Sillitti, and G. Succi. How to calculate software metrics for multiple languages using open source parsers. In *OSS*, volume 404 of *IFIP Advances in Information and Communication Technology*, pages 264–270. Springer, 2013.
36. F. Jouault and J. Bézivin. Km3: A dsl for metamodel specification. In R. Gorrieri and H. Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems*, pages 171–185, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
37. F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proc. GPCE*, pages 249–254. ACM, 2006.
38. H. H. Kagdi, J. I. Maletic, and B. Sharif. Mining Software Repositories for Traceability Links. In *ICPC*, pages 145–154. IEEE, 2007.
39. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. E. Damian. The promises and perils of mining github. In *MSR*, pages 92–101. ACM, 2014.
40. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. E. Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, 2016.

41. S. Karus and H. C. Gall. A study of language usage evolution in open source software. In *Proc. MSR*, pages 13–22. ACM, 2011.
42. E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. I. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hossein, and D. Hearn. TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *Proc. ICSE*, pages 1375–1378. IEEE, 2012.
43. W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot. MoScript: A DSL for Querying and Manipulating Model Repositories. In *Proc. SLE 2011*, volume 6940 of *LNCS*, pages 180–200. Springer, 2012.
44. D. S. Kolovos, N. D. Matragkas, I. Korkontzelos, S. Ananiadou, and R. F. Paige. Assessing the Use of Eclipse MDE Technologies in Open-Source Software Projects. In *Proc. OSS4MDEMODELS*, volume 1541 of *CEUR Workshop Procs*, pages 20–29, 2015.
45. H. König and Z. Diskin. Efficient consistency checking of interrelated models. In *ECMFA*, pages 161–178, 2017.
46. R. L. Krikhaar. Reverse Architecting Approach for Complex Systems. In *Proc. ICSM*, pages 4–11. IEEE, 1997.
47. A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *WCRE*, pages 133–142. IEEE Computer Society, 2005.
48. A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information & Software Technology*, 49(3):230–243, 2007.
49. R. Lämmel. Relationship maintenance in software language repositories. *The Art, Science, and Engineering of Programming Journal*, 1, 2017. 27 pages.
50. R. Lämmel, R. Linke, E. Pek, and A. Varanovich. A framework profile of .NET. In *Proc. WCRE*, pages 141–150. IEEE, 2011.
51. R. Lämmel, E. Pek, and J. Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *SAC*, pages 1317–1324. ACM, 2011.
52. R. Lämmel and A. Varanovich. Interpretation of Linguistic Architecture. In *Proc. ECMFA*, volume 8569 of *LNCS*, pages 67–82. Springer, 2014.
53. R. Lämmel and V. Zaytsev. Language support for megamodel renarration. In *XM@MoDELS*, volume 1089 of *CEUR Workshop Proceedings*, pages 36–45. CEUR-WS.org, 2013.
54. J. Lara, J. Di Rocco, D. Di Ruscio, E. Guerra, L. Iovino, A. Pierantonio, and J. S. Cuadrado. Reusing model transformations through typing requirements models. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering - Volume 10202*, pages 264–282, New York, NY, USA, 2017. Springer-Verlag New York, Inc.
55. M. Lungu, M. Lanza, and T. Gîrba. Package Patterns for Visual Architecture Recovery. In *Proc. CSMR*, pages 185–196. IEEE, 2006.
56. M. Lungu, M. Lanza, and O. Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwarenaut. *Sci. Comput. Program.*, 79:204–223, 2014.
57. O. Maqbool and H. A. Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Trans. Software Eng.*, 33(11):759–780, 2007.
58. G. C. Murphy and D. Notkin. Lightweight Lexical Source Model Extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, 1996.
59. J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio. Collaborative repositories in model-driven engineering [software technology]. *IEEE Software*, 32(3):28–34, May 2015.
60. C. D. Roover, R. Lämmel, and E. Pek. Multi-dimensional exploration of API usage. In *Proc. ICPC*, pages 152–161. IEEE, 2013.
61. F. J. B. Ruiz, J. G. Molina, and O. D. García. On the application of model-driven engineering in data reengineering. *Inf. Syst.*, 72:136–160, 2017.
62. A. D. Sandro, R. Salay, M. Famelis, S. Kokaly, and M. Chechik. MMINT: A Graphical Tool for Interactive Model Management. In *Proc. MoDELS 2015 Demo and Poster Session*, volume 1554 of *CEUR Workshop Procs*, pages 16–19, 2016.
63. K. Sartipi and K. Kontogiannis. On Modeling Software Architecture Recovery as Graph Matching. In *Proc. ICSM*, pages 224–234. IEEE, 2003.
64. D. C. Schmidt. Model-Driven Engineering. 39(2), 2006.
65. A. Seibel, R. Hebig, and H. Giese. Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance. In *Software and Systems Traceability.*, pages 215–240. Springer, 2012.
66. P. Stevens. Towards sound, optimal, and flexible building from megamodels. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pages 301–311, 2018.

67. C. Stringfellow, C. D. Amory, D. Potnuri, A. A. Andrews, and M. Georg. Comparison of software architecture reverse engineering methods. *Information & Software Technology*, 48(7):484–497, 2006.
68. F. Tomassetti, M. Torchiano, A. Tiso, F. Ricca, and G. Reggio. Maturity of software modelling and model driven engineering: A survey in the italian industry. In *Proc. EASE*, pages 91–100, may 2012.
69. D. Wille, Ö. Babur, L. Cleophas, C. Seidl, M. van den Brand, and I. Schaefer. Improving custom-tailored variability mining using outlier and cluster detection. *Science of Computer Programming*, 163:62 – 84, 2018.
70. T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572. IEEE Computer Society, 2004.