Preserving and Synchronising Hand-Written Text in Model-to-Text Transformation

Sultan Owaytiq Almutairi

Doctor of Philosophy

University of York
Computer Science

December 2022

Abstract

Model-Driven Engineering (MDE) is an approach to software development that has been suggested as a possible alternative to more traditional programming based approaches to the problem of developing software that can effectively cope with the complexity of modern systems. MDE encourages the use of modeling languages as a means of providing an abstract description of systems and offers methods for automatically generating a variety of development artifacts, such as documentation and code, from the source models. In the process of developing a complex system, several stakeholders are often involved. These stakeholders make use of a variety of tools in order to alter the development artifacts, source models and generated code. Model-to-Text transformations (M2T) are used for the generation from models of any kind of textual artefact, such as documentation, source code, requirements specifications, and manuals. The focus of this thesis is on combining text generated with such M2T transformations and hand-written text. In particular, the thesis makes two contributions: (1) an approach for adding hand-written lines of code anywhere in generated files, and for preserving them upon re-generation and (2) an approach for automatically synchronising changes made to embedded code fragments at the generated code level, with the models from which these fragments originate. The two proposed approaches have been implemented on top of an existing model-to-text transformation language (the Epsilon Generation Languages) to evaluate their completeness, applicability, and performance.

Contents

List of Tables

List of Figures

Listings

List of Algorithms

Acknowledgements

In the first place, I would like to express my endless gratitude to my supervisors Prof. Dimitris Kolovos and Dr. Thanos Zolotas for their incredible and continuous support and valuable guidance throughout the five years for this project. This thesis could not have been realised without their support. I will be forever indebted to them for everything they have given me.

I would also like to thank my assessor Prof. Ibrahim Habli and external examiner Prof. Artur Boronat, for providing me with feedback and advice.

My heartfelt gratitude and appreciation go to my beloved mother, sisters, and brothers for their unending love and support on a daily basis. They are constantly present to solve any problems that happen during my life.

A big thanks goes to my wife, for fruitful discussions and advice on my work but also for her patience and continuous support during this period. Thanks to you, our daughter, and our son, for understanding my absences and shortcomings throughout this period.

I would like to thank my colleagues in the Automated Software Engineering group, Dr. Horacio Hoyos, Dr. Beatriz Sanchez, Dr. Konstantinos Barmpis, Dr. Alfa Yohannis, Dr. Faisal Alhwikem, Dr. Simos Gerasimou, Dr. Sina Madani, Sorour Jahanbin and Qurat ul ain Ali, for their daily support. Thanks to my colleagues and friends in the department Dr. Sultan Alahmari, Dr. Saud Yonbawi, Dr. Naif Alasmari, Dr. Abdullah Albalawi, for the interesting discussions that we have shared. I would also like to thank Dr. Ibrahim Alotaibi, Mr. Mohamad Alassaf and Mr. Abdullah Alotaibi for making me feel settled and happy in the UK, and for consistently offering good advice.

Finally, thanks to Shaqra University, which granted me this scholarship and supported me throughout this period.

Declaration

I declare that all the work of this thesis is the result of my own research which I carried out between February 2018 and December 2022. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References. The following publication has been written by the PhD candidate.

S.Almutairi, A. Zolotas, D. Kolovos “Towards Round-Trip Engineering of Code Fragments Embedded in Models”, in Proceedings of the 16th Workshop on Models and Evolution (ME) 2022 - MODELS ’22

Chapter 1

Introduction

This chapter discusses the significance and scientific challenges of the integration of integration of hand-written and generated text and the synchronisation process between source models and their target-generated artefacts in Model-To-Text Transformations (M2T), in the context of Model-Driven Engineering (MDE). While the integration of hand-written text and the synchronisation process are essential and have been the subject of several research contributions, there are still many challenges that can be improved. These challenges are described in more detail in the following section.

Chapter Structure. Section 1.1 provides a brief description of the research area of this thesis. Section 1.2 presents the research challenges. Section 1.3 provides a brief description of the contributions of this thesis. Lastly, the structure of this thesis is presented in Section 1.4.

1.1 Context

Model-Driven Engineering (MDE), which advocates for automation and provides developers with the ability to work with artefacts that are near to their area of expertise [126]. The core concept underlying MDE is to employ models that capture the underlying system complexity in a specific context to simplify its understanding and manipulation. Models were traditionally represented as drawings of systems views, and their purpose was to direct the process of development.

Nevertheless in MDE processes, models are alive entities that are at the centre of the development process [126]. They are responsible for capturing information in an organised manner so that it can be processed automatically. Models are capable of being compared, queried, validated, and transformed into other models or different artefacts (e.g., documents, configuration artefacts and code). All of these automated tasks save time that would have been spent on manual and often error-prone tasks. For instance, code generation can decrease the time it takes to write code, the errors of programming, and code reviews.

Model-to-Text (M2T) transformations are used for the generation of any kind of textual artefact from models, such as documentation, source code, requirements specifications, and manuals [123, 108]. In the automated software development field, code generation is of paramount importance, and is a core concept in MDE [157, 62]. Software quality and programmers' productivity can improve significantly through correct code and proper comments. Automatic code generation can alleviate problems, which are typically costly, error-prone, and time-consuming [157]. According to Balzer et.al. [13] there are many advantages of code generation. It aims to mitigate the need for tedious work, and to minimise maintenance costs by raising the quality of code and helping limit programming errors [25]. Developers can save time as they need to write fewer lines of code. However, as models should be at a high level of abstraction, they cannot capture all the system information [53]. Thus, developers require to integrate the missing information into generated artefacts.

Round-trip engineering (RTE) is the ability to automatically preserve the consistency of various changing software artefacts in software development environments/tools [63]. RTE is one aspect of MDE, because the target generated code and the source model are interrelated; altering the generated code will affect the source model and vice versa. RTE consists of forward and reverse engineering. Forward engineering is the process of converting conceptual models into source code, whilst reverse engineering is the process of converting source code into conceptual models [104]. In software development, generating code from source models and then performing round-trip engineering are critical processes. Throughout the development process, it is vital that software-related artifacts such as models and their source code remain in sync [104].

Dealing with integrating hand-written text is one of the main challenges in M2T transformations. Greifenberg et al. [53] discussed eight mechanisms to integrate generated and hand-written text for OOP languages such as generation gap and delegation mechanisms. Some of these mechanisms require a clear separation between hand-written text and auto-generated text. Some benefits arise in separating hand-written text and auto-generated text, such as mitigation of generated code becoming polluted, and enabling the editing of just a separate file for hand-written changes. However, a drawback is system information being spread between two or more places, and possible developer confusion from the extra files. Other mechanism proposals involve merging hand-written text merged into generated code using constructs such as protected regions. For example, the protected regions mechanism works by inserting hand-written text inside tagged blocks within generated files, eliminating the need for additional files. However, using such a mechanism the generated code will be polluted with hand-written text and the comment/tags for protected regions, as each protected region will have two extra lines: the beginning and the end comments of the regions. Another major limitation of using a protected region mechanism is that if the developer wishes to change any part of the class, such as adding a new variable or a new field, they may be unable unless an appropriately positioned protected region is present. Although, using these existing mechanisms does provide benefits to developers they still have some restrictions. A mechanism that solves these limitations of existing mechanisms and gives developers the freedom to include their hand-written text anywhere in the generated artefacts, would be valuable.

The second challenge is that Round-Trip Engineering (RTE) process is not supported in M2T transformations. Demeyer et al. [37] describe RTE as the integration of design diagrams and source code, as well as modelling and implementation. The purpose of RTE is thus to ensure seamless interaction between the design and execution stages. Because the target-generated artefacts and the source model are interrelated, altering the target-generated artefacts will affect the source model and vice versa. Typically, when generated artefacts, that were originally generated from source models, are modified by developers, consistency between these models and their generated artefacts is lost. Supporting the process of maintaining consistency between two different representations is a complex task. Therefore, the majority of developed tools provide relatively limited support.

This thesis proposes two novel approaches: the first is to facilitate the process of integrating hand-written and generated text and the second is to support the synchronization process in M2T transformations. The first challenge is that although there are some mechanisms that support the integration of hand-written text, some require a clear separation between the generated and hand-written text, or integrate them within the generating code but using specific tags each time to prevent them of being overwrite during the regeneration. To meet the first challenge, an approach has been proposed to facilitate the process of integrating hand-written text in the generated artefacts directly. This approach is based on giving developers the freedom to integrate hand-written lines anywhere in the generated code and preserving these lines upon regeneration without the need for protected regions. The second challenge is that while there is a substantial abstraction gap between models and their generated artefacts: Models are at higher level of abstraction than the generated artefacts [63]. Since of the gap, it is difficult to apply the currently available synchronization mechanisms because there is no straightforward mechanism to reflect changes made in generated artefacts back to the source model. Both proposed approaches have been evaluated and the results show that the process of integrating hand-written text into M2T transformations can be facilitated as well as maintaining synchronization between the source models and the generated artefacts. In addition, the proposed approaches have been implemented on top of an existing M2T language called Epsilon Generation Language (EGL) [124].

1.2 Research Challenges

While models are the centre artefact for designing systems in MDE, they cannot capture the whole system alone due to the high level of abstraction [22, 64, 53]. Thus, developers are required to complete the missing information at the code level (generated artefacts). The problems are stated as follows:

> *When integrating hand-written text is required, it is not ideal to use common existing approaches as some require a clear separation between the generated and hand-written text, or integrate them within the generating code but using specific tags each time to prevent them of being overwrite during the regeneration. It is also difficult to synchronize the source model with such integration at the code level due to the abstraction gap between source model elements, and the target generated artefacts.*

- **Research Challenge 1 (RC-1): Integrating hand-written text approach.** Existing approaches for integrating hand-written and generated text in M2T transformations require deciding in advance where hand-written text can be contributed.

- **Research Challenge 2 (RC-2) : Code-synchronize approach.** To achieve full code generation, models often need to embed code fragments. Editing such fragments at the model level poses usability challenges (e.g. lack of code assistance, syntax highlighting, error detection) while editing them at the level of the generated code requires manual synchronisation (i.e. copying and pasting modified fragments back to the model), which is labour-intensive and error-prone.

1.3 Thesis Contributions

In this thesis two novel contributions for addressing the problems of extendibility and synchronisation in M2T transformations are proposed:

- **First contribution.** A technique for adding hand-written lines of code anywhere in generated files, and for preserving them upon regeneration.
- **Second contribution.** A technique for automatically synchronising changes made to embedded code fragments at the generated code level, with the models from which these fragments originate.

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

Chapter 2 reviews literature that relates to the concepts behind this research project. More specifically, Section 2.1 presents the key components of MDE such as meta-model, models, modelling languages, and model management activities. Section 2.2 presents an overview of model transformations with more focus on model-to-text transformations (M2T). Existing mechanisms for integrating hand-written text and highlighting their advantages and disadvantages are presented in Section 2.3. Section 2.4 presents an overview of round-trip engineering (RTE) in MDE and also highlights the existing synchronisation mechanisms. Finally, Section 2.5 summarises this chapter.

Chapter 3 presents an overview of the main research problems. More specifically, Section 3.1 highlights the limitations of using existing integration approaches. Section 3.2 presents an example of these limitations. Section 3.3 contends that there are drawbacks and limitations in using the existing integrated approaches. Lastly, the hypothesis and objectives of this research are presented in Section 3.4.

Chapter 4 presents the first contribution of this work, a technique for embedding hand-written text into the generated files in M2T transformations. This approach includes solutions for integrating hand-written text within generated artefacts in Model-to-Text Transformation. More specifically, an overview of the proposed approach is given in Section 4.1. The implementation of the proposed merging approach is presented in Section 4.2. Section 4.3 presents an evaluation of the proposed approach, mainly its correctness and performance. The results of the evaluation are also presented. Section 4.4 presents the limitations of the proposed approach are discussed. The alternative solutions for the proposed approach are discussed in Section 4.5. Finally, Section 4.6 summarises this chapter.

Chapter 5 presents the second contribution of this work, a technique for supporting round-trip engineering to facilitate the automated synchronisation between models, and the textual artefacts generated from them, via template-based M2T transformation. More specifically, an overview of the proposed approach is given in Section 5.1. The implementation of the synchronisation approach is presented in Section 5.2. Section 5.3 presents an evaluation of the proposed approach, mainly its correctness and performance. The results of the evaluation are also presented. Section 5.4 discuses the limitations of the proposed synchronisation approach. Finally, Section 5.5 summarises this chapter.

Chapter 6 provides a summary of the contributions proposed by the thesis to fill the identified gap. It also provides directions for future research in this work.

Chapter 2

Literature Review

This chapter presents an overview of MDE; the key principles, practices, and the tools that are necessary to understand this thesis. It also presents a critical review of existing research in the area of model transformations - more specifically in Model-to-Text Transformations (M2T).

Chapter Structure. Section 2.1 presents background on MDE and its key technologies. Section 2.2 presents background on model transformations focusing on M2T transformations. Section 2.3 presents integrating handwritten text in M2T transformations. It also highlights and compares the existing approaches. Section 2.4 presents background on Round-Trip Engineering (RTE). It also presents the existing approaches that support RTE in MDE. Section 2.5 summarises the chapter.

2.1 Model-Driven Engineering

This section presents an overview of Model-Driven Engineering (MDE); the key principles, practices, and the tools that are necessary to understand this thesis.

MDE terminology can be quite conflated among the various, unfortunate synonyms used. To mitigate this confusion, Brambilla [21] presents the different acronyms used in the field. These acronyms within MDE pertain to different levels of broad to narrow meanings for the application of the particular approach. Figure 2.1 depicts the relationships between the acronyms used to describe the modelling approaches.

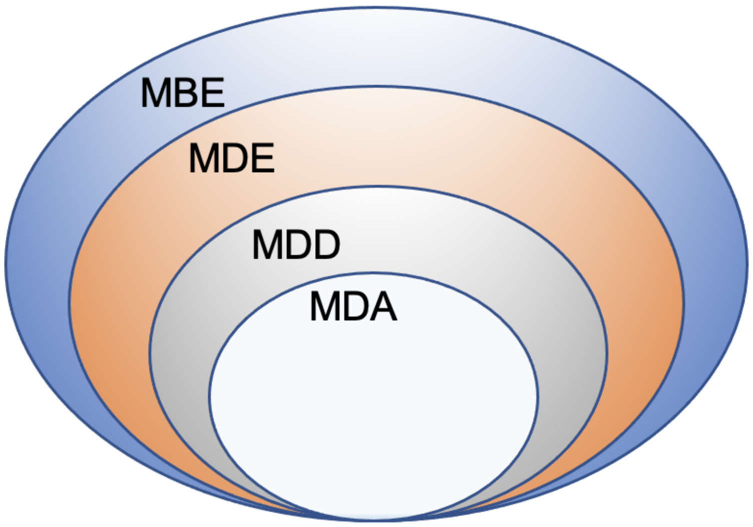


Figure 2.1: The relationship among the various MD* acronyms [21].

The development paradigm, which typically makes models its primary development process artifact is called Model-Driven Development (MDD). In this paradigm, the implementation is often generated (semi)automatically from the source models.

The Object Management Group (OMG) has proposed a Model-Driven Architecture (MDA), which is considered a particular version of MDD and includes the use of OMG standards. As a result, MDA can be considered as a subset of MDD, where OMG has standardised the languages used for transformation and modelling.

However, MDE can also be considered as a superset of MDD, as it extends beyond activities solely based in development. This reaches into other tasks using models in comprehensive software engineering, such as the system undergoing model-based evolution, or a legacy system being reverse engineered through model-driven activities. The term Model-Based Engineering (MBE) is a less strict form of Model-Driven Engineering, because that MBE process involves important roles undertaken by software models, though the development is not strictly centred around (or driven by) models as key artifacts.

2.1.1 MDE Principles

Model-Driven Engineering (MDE) is a specific software engineering strategy, which encompasses the use of models as the principal artefacts in the course of the process [16]. Typical software engineering practises approach problem solving primarily through a focus on the implementation details and the architecture between areas of behaviour. MDE focuses on the use of models as the primary method of problem solving, unlike typical engineering practises, which use models more for communication between architects, developers and stakeholders [84]. This characterisation of the difference between software engineering disciplines is in line with Men's [102] definition of MDE, describing its reliance on models as first-class entities which accommodate software maintenance and evolution through model transformations.

The appeal of MDE is founded on a recognised and successful software engineering principle of using higher levels of design specification abstraction [131, 103, 11]. This separates system specification from the implementation details and brings benefits such as improved software quality, and reduced development time due to the increased automation of repetitive activities such as code generation and system verification [132, 8]. To understand these benefits, it is necessary to examine the underlying terminologies and principles of MDE, which consists of core concepts like models, meta-models, and surrounding concepts like exchange formats, modelling languages and modelling platforms.

2.1.2 Meta-models and Models

This section presents the core principles behind MDE, including system, model, meta-model, and their relationships.

Meta-models and models are elementary units of MDE. The models provide problem simplification and system abstraction, and the meta-models provide modelling notation through abstract syntax [21]. The domain concepts can be captured based on elements within a meta-model, with models used to express systems in these specific domains [107].

In MDE, "A model is a simplification of a system built with an intended goal in mind" [17]. Models are used as an abstraction of the actual environment being developed for, including reality itself, or the rest of a system being engineered, or both. Typically, a model is comprised of three aspects: concepts, relationships, and structure [27]. Concepts describe the domain characteristics being modelled. Relationships describe how concepts are related to one another. Additional characteristics that limit how the concepts of domain can be combined to create a valid model are referred to as structure [27].

Another description of a model is that it abstracts the real system in such a way that it can be studied and queried to answer productive design questions [18]. Consequently, a model's utility will be characterised by the reliability that the stakeholders feel the model provides in answering such questions, in reaching and maintaining the goals of a system [102]. With this definition, it follows that the primary purpose of a model is to help reduce the complexity of a concrete system through abstraction. It is important that the implementation details of a system are out-of-mind during high-level design, which MDE helps during the design process through representing complex problems in simpler, but still practical terms. A primary advantage in modelling is the ease brought to the understanding of a problem domain, between both non-technical and technical stakeholders [34]. Scientific fields will often use models and modelling to their advantage as a concept, such as in economics, physics, and other mathematical disciplines.

According to Bezivin et. al. "A meta-model is the explicit specification of an abstraction (a simplification)" [17]. Meta-models are used to represent the models as concrete implementations of other, more abstract models. The semantics and constraints detailed in meta-models are associated with concepts within the domain [130]. A meta-model will ensure that models conform to a common set of elements, akin to a contract of capabilities, which allows models to be treated with their specific implementation details hidden. Figure 2.3 illustrates a model M, that conforms to the meta-model in Figure 2.2. In this example, all elements of M initially depend on being defined in MM. A model can implement elements from multiple meta-models at once, with each meta-model defining just part of a contract as a way to use and manipulate the model in different situations [107].

2.1.3 Modelling Languages

The previous section explored model and meta-model concepts; this section focuses on the definition of modelling language characteristics such as meta-models.

Brambilla has defined a modelling language as "a tool that lets designers specify the models for their systems" [21]. The definition of a modelling language also includes its concrete syntax, not just its abstract syntax, with both contributing their semantics to the definition [79]. Typically, modelling languages include the three following aspects:

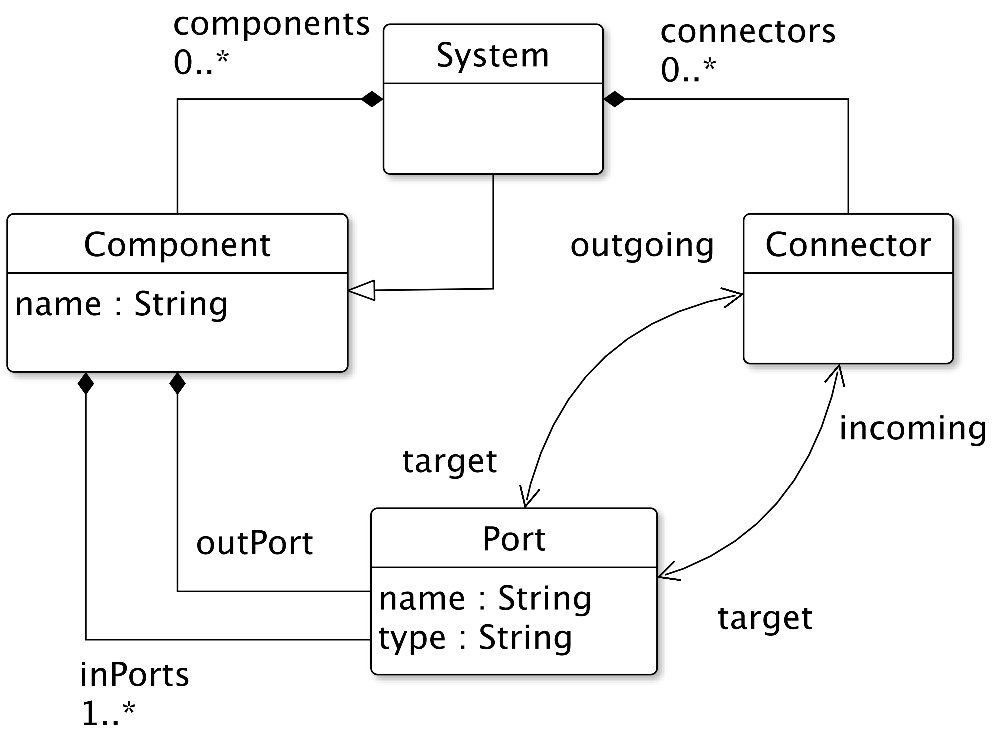


Figure 2.2: Meta-model of the Component-Connector DSL.

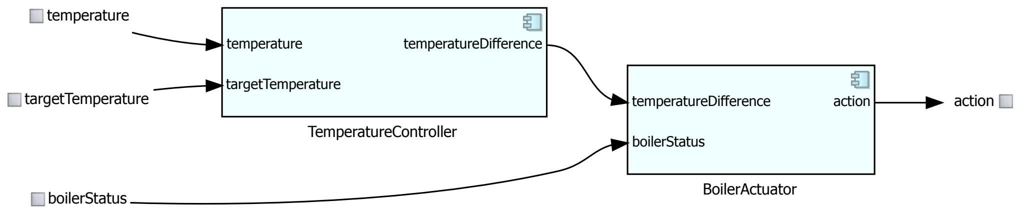


Figure 2.3: BoilerController model that conforms to the meta-model in Figure 2.2.

The Semantics. These identify the domain, the meaning of modelling concepts, and their combinations [21]. Consider an example of a typical modelling construct: the tree. Between languages, the semantics of a tree will likely differ. These modelling language semantics might be precisely specified with a formal reference semantics language such as Z [137], or alternatively employ a natural language to achieve the same in a semi-formal manner [143].

The Abstract Syntax. A language has its described concepts defined by abstract syntax. For example the concepts of datatypes, packages, and classes. There is an independence between the concrete syntax and the way concepts are represented [143]. The implementation of software, such as a compiler, typically makes use of Abstract Syntax Trees (ASTs) to encode and represent the program's abstract syntax, even if the same language has concrete syntax that is purely graphical or textual. Abstract syntax also describes the grammatical rules and structure of the language, this being the permitted connections and constructs between them [21]. A meta-model is generally used to specify abstract syntax in modelling languages [79].

The Concrete Syntax. Model construction that has conformity with the language has its notation provided by concrete syntax [143]. As an example, a simple group of line-connected boxes may be the concrete representation of a model; or by using particularly organised tables, forms, and matrices [79]. Communication is enabled through concrete syntax standardisation. Different forms of concrete syntax can be used to aid better consumption and analysis. This includes XML Metadata Interchange (XMI), for machine readability and model distribution, or the Unified Modelling Language (UML), which has a concrete syntax better suited for human readability.

Classification of Modeling Languages

The separation of modelling languages has no conclusive rule, although there are some proposed classifications [160]. A particular classification has languages separated into the categories of domain-specific and general-purpose.

Domain-Specific Languages (DSLs). Requirements for particular companies, contexts, or domains can be catered for in these languages [21]. They are typically found to be easier to use and be more productive than languages for general purpose applications [79]. This is because concepts are used from the actual problem domain and as a result the level of abstraction is raised. As they are built for use in specific domains, they are not as portable as general-purpose languages but instead assist productivity in that domain [79]. They also help the comprehension of written code particularly by domain experts, and therefore in the communication of those problems [47].

Structured Query Language (SQL) is a prominent example of a DSL created specifically for database manipulation [36]. Goal Structured Notation (GSN) [138] is another example of a DSL used for argument structuring and in representing relationships between the evidence of assurance cases and the arguments that help support the case. Business Process Model and Notation (BPMN) is a graphical DSL for business process representation [109]. VHSIC Hardware Description Language (VHDL) operates in the domain of electronic systems, and so is intended to model those systems [70]. And the Web Modelling Language (WebML) is intended for use in the specification of navigation features, composition, and content, as a graphical DSL [28].

General-Purpose Languages (GPLs): For modelling purposes, and to be applicable to any domain, GPLs are used as agnostic notations and constructs are provided, which are meant to be universal. UML [111], SSADM [10], SysML [110], IDEF [71], and MERISE [12] are examples of GPLs. The application domain of the Unified Modelling Language (UML) [111] is so broad, even though its intent is object-oriented software-based systems implementation, design, and analysis, that it could be viewed as a general-purpose language on its own [21]. Other GPLs could also be classified in this way, for example MERISE, which is focussed on information systems.

Languages can also be classified into graphical/visual, textual, or a hybrid of both, and are commonly used as another type of language classification [128].

Textual. As is expected by its name, textual languages are composed of plain text, encoded as written words, punctuation, and indentation. Popular examples of textual languages are Java [47] and HTML [14]. Textual languages make up the majority of the prominent programming languages. Editors for textual languages can be generated with tools such as EMFText [61] and Xtext [40] to have basis in the principles of MDE.

Graphical/Visual. As opposed to textual languages, graphical languages, express themselves with icons, shapes, and possible connections between them. A distinctive example of a graphical language is UML [6]. Akin to textual languages, there are tools available to generate graphical model editors, such as Eugenia [90], AToM3 [94], Graphiti, and the Sirius [148] framework which extends the GMF.

Hybrid. A combination of graphical and textual syntax languages is known as a hybrid language. The system in development can be described in different aspects using the notations available [115].

2.1.4 Meta-Object Facility (MOF)

The Meta Object Facility (MOF) is an architecture specified by the Object Management Group (OMG) for modelling language construction [56]. Figure 2.4 illustrates the architecture. M3 is the highest level, and it comprises of meta-modelling languages, or languages that may be used to construct meta-models similar to those found in the M2 level. For example, the UML Meta-model [58] is an OMG proposed meta-model, used to define models that appear on the M1 level, such as Activity or Class diagrams. The M0 level, which is the lowest level, is composed of a set of elements which reflect real world domain entities that should be modelled. Some examples of these elements include video player and reservation system.

The layered design of MOF is analogous to how computer programming languages are defined, in layers: a language used for language definition, such as EBNF [50]; a programming language, such as Java [112]; code written in said language, Book.java for example, and real-world objects as the lowest layer [21]. Models in MOF are saved using the XML Metadata Interchange (XMI) format [55], which is an OMG standard that permits compatibility amongst modelling suites that are MOF-based. Aside from the structural constraints enforced by meta-models, the MOF standard does not allow for the introduction (making up meta-model semantics) of constraints. To express more complicated constraints in MOF, the Object Constraint Language (OCL) [57] is employed [107].

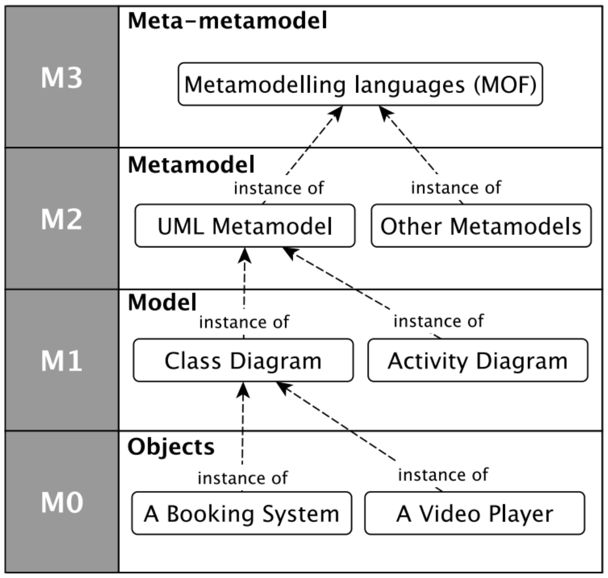


Figure 2.4: The four layers of meta-modelling infrastructures [160].

2.1.5 Model Management Tasks

The previous sections explored the basic concepts of models and modelling languages; now the focus is placed on how these models can be used meaningfully in the process of development. Activities that process, analyse, or evolve programmatically, or make use of models in any way, shall be referred to as model management tasks.

Model validation. Certain external constraints must often be satisfied within a model. Inconsistent or incomplete models (where information is missing) can give rise to problems [86, 93]. These inconsistencies can arise due to syntax, where they are not in conformance with their meta-model, or due to semantics, such as where semantic constraints are unsatisfied [41]. Intra-model consistency relates to semantic and syntactic consistencies, such that they constitute a single model's properties [41]. Inter-model consistency relates to the same information being captured in conflicting ways between different models, in some cases between views of one system [41]. Inconsistency must be discovered and subsequently addressed, due to the crucial nature of models within MDE, in order to avoid an inconsistency finding its way into deployed systems, having proliferated through transformation chains. Model constraints can be expressed in languages such as the Object Constraint Language (OCL) [57], Epsilon Validation Language (EVL) [89], and ATL [75] (as seen in [74]).

Model comparison. The comparison of models covers a variety of tasks related to models and is a vital activity [158]. Calculating model differences and verifying that the outcome of model transformation is as intended are just a few examples. Due to their structure, comparing models is difficult [158]; models cannot be compared solely on internal XMI representation, since two representations of one model might have several discrepancies, for example different element orderings or IDs [144]. Some methods of comparison are as follows: strategies which are signature based (such as in [42])), in which the identities of elements are generated at the moment of comparison; approaches which are graph-based (such as [144]), in which models are considered as graphs with typed attributes; or the use of language specifically for such a task, defining the exact rules for matching between elements (such as in [159]).

Model Querying. Models tend to store practical and useful information. As a result, being able to query models to obtain data stored in them is desirable. It proves valuable, as it provides easier detection of model flaws through information filtering and aggregation, making it a centre component of every model management task [29]. Model querying can be carried out using languages such as OCL [57], Epsilon Object Language (EOL) [88], and ATL [74]. OCL is in the functional paradigm, devoid of side-effects and imperative characteristics, and is most popular for model querying and validation [99].

Other. Model management tasks include model transformations, where models can transform to other models or text (discussed in more detail in next Section 2.2) and model merging, which is referred to the process of integrating two or more models into one.

2.2 Model Transformations

Model transformations are programs that transform models into other representations (e.g., documentation, other models and source code) [26]. They are considered the cornerstone of MDE, as they offer the essential mechanisms to manipulate and transform models [145]. Transformation theory and tools are essential for model operations such as merging, refactoring, code generation, weaving, etc [133]. Various purposes are served by the application of model transformations in MDE, such as model quality enhancement, automating software evolution, platform-independent models (expressed as platform-specific models), reverse engineering models, and identifying software patterns, etc [83]. Another example is the generation of models on different levels of abstraction and the automation of model development tasks [32].

Specifications for model transformations are typically defined by a set of transformation rules. Source meta-model types are declaratively specified through these rules, correspondingly mapped to target meta-model types. OCL (Object Constraint Language) can make transformation languages more expressive, enabling model properties to be formally specified in expressive formats [119]. OCL-like expressions can help to define mappings between target and source model elements. Multiple rule definitions can be contained within a transformation specification. The order of execution of rules can be controlled through mechanisms in many transformation languages. This is called rule scheduling. This can be implicit or explicit [92]. Implicit rule scheduling enables the automatic realisation of the relationship between different rules, whereas explicit scheduling enables rule execution orders to be manually specified.

There are three different types of model transformations, categorised by the types of target the transformation produces: Model-to-Model transformation refers to the transformation from one or more source models to one or more target models. In M2M transformations, the process of transformation is determined by a set of transformation rules. Each rule focuses on the way in which a set of elements in the source models can be transformed into a set of elements in the target models. Model-to-Text (M2T) transformations are used for the generation of models of any kind of textual artefacts, such as documentation, source code, requirements specifications, and manuals [123, 108]. Text-to-model (T2M) transformations are able to take given texts and extract models from them. Typically, in order to construct models appropriately, T2M transformations require sophisticated reverse-engineering technologies [21].

Since this research focuses on the development of merging and consistency processes in model-to-text transformations, in the following sections, this type of transformation is presented in more detail.

2.2.1 Model-to-Text Transformations (M2T)

Model-to-Text (M2T) transformations are used for the generation of models of any kind of textual artefact, such as documentation, source code, requirements specifications, and manuals [123, 108]. Figure 2.5 shows an example of M2T transformation. M2T transformations are used to generate text as opposed to structured models, with the generated text being independent of the target language; most often it no longer conforms to any meta-model. As M2T transformations are used for unstructured data, the requirements are completely different from the requirements for M2M transformations.

Some source code generators produce code for target programming languages by using a model API in existing general-purpose languages. These can be characterised by some examples such as Simulink Coder, which produces C code, and the Ecore-to-Java transformation provided by EMF [126]. However, these general-purpose approaches, which are alternatives to M2T transformations, suffer problems in that dynamic and static code is obfuscated and difficult to read, or make sense of the structure of the final output [21]. M2T mitigates these problems by lending itself toward configuration through a template-based approach. In this approach, the structure of the output, where dynamically generated parts should go, are explicitly represented and clearly indicated in the templates.

Examples of some M2T transformation language offerings are Acceleo [44], Epsilon Generation Language (EGL) [123], and Xpand [45]. M2T languages including MOFScript, JET [30] and Xpand [142], are described in more detail in Section 2.2.1.

Code Generation

In the automated software development field, code generation is paramount, and is a core concept in MDE [157]. Software quality and programmers' productivity can improve significantly through correct code and proper comments. Automatic code generation can alleviate problems, which are typically costly, error-prone, and time-consuming [157].

Herrington et. al. said that "Code generation is about writing programs that write programs." [62]. In software engineering projects, code generation is a valuable tool because of its impact on productivity and quality. Code generators can be passive or active [62]. The passive generation allows users to modify the code as required. "Wizards" are an example of a passive

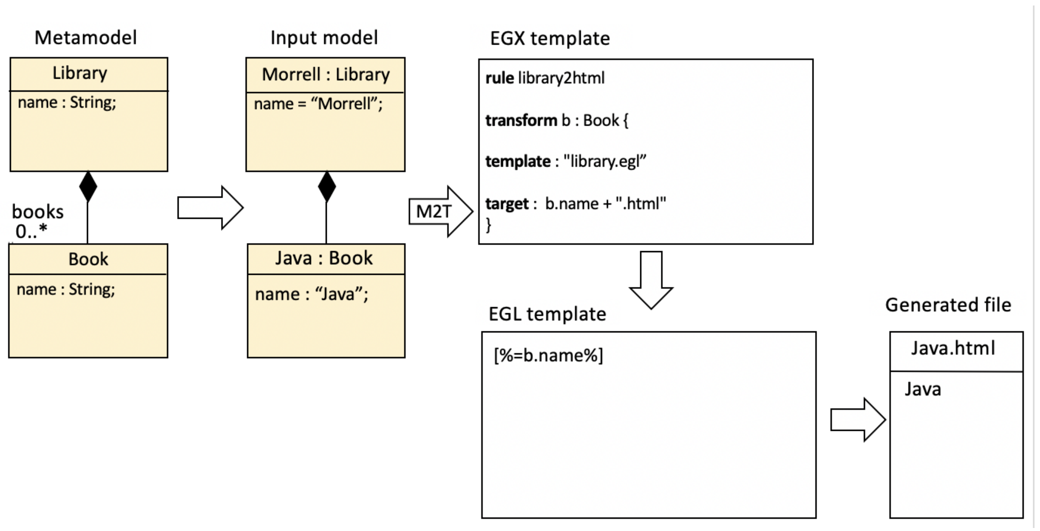


Figure 2.5: An example of M2T transformation using EGL language.

generator. On the contrary, an active generator is responsible for the code whether on the short or long term, through the ability of the generator to run the code many times on the same code output. When the code requires changes, users are able to do this and then run the code again.

According to Balzer et. al. [13] there are many advantages of code generation. It aims to mitigate the need for tedious work, and to minimise maintenance costs by raising the quality of code and helping limit programming errors [25]. Users can save time as they need to write fewer lines of code. Specifications are short compared to the program that is implementing them. Also, because specifications are generated the probability of errors is lower than when directly writing the code. However, aside from the advantages, code generation does still have some limitations [32]. For instance, there are many problems when adding hand-written text to the generated file, which requires high specifications when regenerating. One approach to code generation specification is through template languages [154].

M2T Transformation Categories

M2T transformations are divided into three categories: visitor-based, template-based, and hybrid approaches [33].

Visitor-based. This method involves providing a mechanism that enables visitors to navigate a model's internal representation and write text to a stream [72]. Jamda [33] is an example of a visitor-based programming language that expresses UML models with a collection of object classes. It can access and change models using specific APIs such as the Java metadata interface [117], and it generates text using a visitor mechanism. Other examples of tools using a visitor-based approach are Melange, Kermeta2, ATOMPM, and ATOM3 [76]. Jamda does not support the MOF standard for defining new meta-models; however, additional model element types may be added by subclassing the established Java classes.

Template-based. Templates are text files containing placeholders that resemble the final result of an M2T transformation [107]. The placeholders are variables that will be populated with data from the source model. Static sections have verbatim text written as a transformation result, but dynamic sections comprise sections with place holders. Special command tags (e.g. [% %], <% %>) are also used to surround dynamic sections. Unlike the visitor-based method, the template-based structure closely resembles the target language's syntax.

Typically, MDA tools that are currently available enable the generation of template-based model-to-code, for example FUUT-je, b+m Generator Framework, Codagen Architect, JET, AndroMDA, OptimalJ; and XDE and ArcStyler, which also provide M2M transformations [33]. AndroMDA is built upon existing opensource technology for template-based generation: Velocity [1] and XDoclet [2].

Hybrid. A visitor-based approach will be simpler than a template-based approach in some cases. However it is ineffective when the majority of generated code is static text [76]. As a result, M2T tools may be designed and implemented using template-based languages, which also employ the visitor pattern. Hybrid based applications include Actifsource[1], Modelio[2], MetaEdit+[3], Xtend[4], and GrGen.NET[5].

[1]`https://www.actifsource.com/`

[2]`https://www.modelio.org/`

[3]`https://www.metacase.com/`

[4]`https://www.eclipse.org/xtend/`

[5]`http://www.info.uni-karlsruhe.de/software/grgen/`

M2T Transformation Languages

Many M2T transformation languages have been proposed over the last few years. This section presents the most important M2M transformation languages.

Epsilon Generation Language (EGL). Epsilon's M2T transformation language is the Epsilon Generation Language (EGL) [123]. It is, in principle, similar to server-side scripting languages like PHP languages - in fact, EGL can be used for this same purpose, as illustrated in [3]. However, it inherits some of its logic and concepts from EOL [88].

A parser is at its heart, generating an abstract syntax tree (AST) for a given template that includes both static and dynamic output nodes [123]. The EGL transformation engine, unlike Acceleo, has its own specialised execution coordination mechanism (EGX) for transformation rules. EGX may generate numerous files from a single template and coordinate rule execution. It borrows EOL's imperative features, offers data types that are comparable to Java's, and allows for meta-model types to have user-defined methods.

As EGL uses templates, like other M2T transformation languages, it makes use of both dynamic and static regions [113]. These regions have different purposes, with static regions used for including verbatim text in the output; and dynamic regions used for generating the text proactively, during transformation, perhaps including data only available at this time. EOL is used to express the behaviour of dynamic regions. An EGL template can be seen either as a regular plain-text file, which includes embedded EOL code, or inversely as an EOL program that includes a capability to generate verbatim text. Its various features can be used or substituted as needed, for example with some programs using only dynamic regions, writing output text by using the output buffer variable [98]. The output variable available in EGL is named "out", and dynamic regions are marked by the use of "[%" and "%]" surrounding them. Other markers are provided for convenience, such as the starting marker "[%=" which takes an expression and outputs its text representation.

EGL itself has numerous advanced offerings and features, including:

- postprocess formatting (to enable the final output to have consistent styling)
- recording traceability information

- the ability to mark regions of the text as preserved from being overwritten by template invocations, such as for changes, termed protected regions
- the ability to use any output stream to write its text
- support for merging multiple texts together.

Acceleo. Acceleo is an M2T transformation tool used to create code generators [39], and comprises three parts: a compiler, a generation engine, and some tools. It aims to provide a pragmatic approach to OMG's M2T transformation standard for models based on EMF [100]. The language is able to query models with complete OCL support as well as robust tool support, demonstrated to be valuable in industry [20].

Acceleo generates structured files from EMF models. Modules, target files, and source model(s) all make up an Acceleo transformation. The output of transformations is text in various programming languages or any other textual formalism [4]. A module can comprise numerous templates that describe the parameters required to create text from models. The existence of an @main annotation at the beginning of the template's body indicates that it is the main template, which acts as the transformation execution entry point [107].

MOFScript. MOFScript was created by Sintef and backed by the EU Modelware project as an initial proposal to the model-to-text RFP of OMG [107]. MOFScript was created in response to the requirement of standardisation for M2T transformation languages. It is highly influenced by QVT and works with any MOF-based model, such as BPMN, WSDL, UML, and others. Transformations in MOFScript consist of groups of transformation rules organised into one or more modules. QVT-Merge operation mappings have a specialised analogue in MOFScript rules, whereas specialisations of QVT-Merge operational constructions are provided through MOFScript constructions. Transformations are able to reuse and import other transformations [122].

XPand. XPand is a template-based M2T language that is part of the open-ArchitectureWare (oAW) platform [59]. It has a small vocabulary [81], which puts limitations on the operation types it can perform. In addition to its own capabilities, it is able to access the functions that are implemented in the Xtend programming language, which is also part of the openArchitectureWare (oAW) framework. Xpand provides many features such as polymorphic dispatch, aspect-oriented programming, and static type safety. It can also, between code generations, log link information between target and source elements. In M2T transformations, Xpand is able to achieve some important requirements such as supporting active code generation through the PROTECTED constructs and the provision of beautifiers to improve the readability whether for generated code or templates. Xpend is supported by an Eclipse-based editor that provides error highlighting, syntax colouring, refactoring and code completion [106].

2.2.2 Text-to-Model transformation (T2M)

Text-to-model (T2M) transformations are able to take given texts and reconstruct models from them. Typically, in order to construct models appropriately, T2M transformations require sophisticated reverse-engineering technologies [21]. However, little attention toward these technologies has been paid by the research community until now [24]. Text-to-model tools require a parser, a grammar engine, and a target meta-model. EMFtext [61], and Xtext [46], are examples of T2M tools.

2.2.3 Key Technologies of MDE

This section discusses the modelling tools identified in this thesis, including EMF[6], which implements the MOF architecture, and the Epsilon suite[7], which in turn is used for model management.

EMF - Eclipse Modelling Framework. The most popular meta modelling architecture is almost certainly the Eclipse Modelling Framework (EMF) [140], implemented by the Eclipse Foundation, which exploits facilities available in Eclipse. EMF conforms to the four-layer MOF architecture with modellers provided with Ecore, a meta-modelling language that has well-maintained and stable tool support. This includes a graphical editor, which

[6]`http://www.eclipse.org/modeling/emf/`

[7]`http://www.eclipse.org/epsilon/`

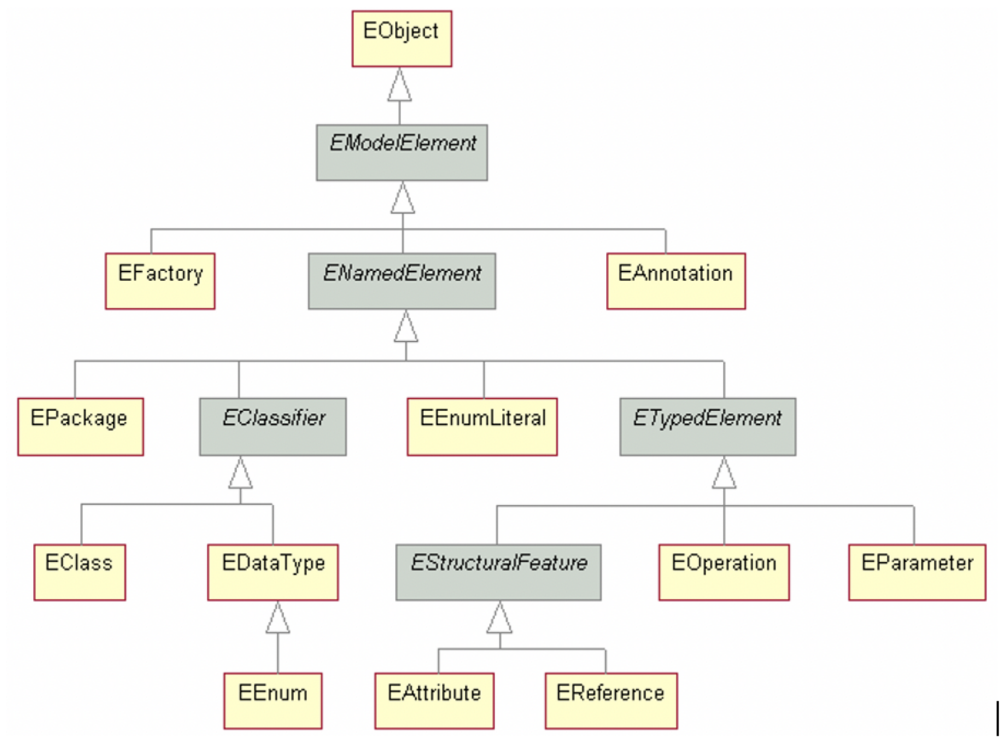


Figure 2.6: Simplified Ecore's components diagram [140].

enables meta-models definition, and tools for automatic meta-model to model editor generation.

Currently, EMF is understood to be the de-facto modelling framework, with a strongly active development community that extends the core framework, providing interoperable tools as a suite to help in model development [158]. To provide some examples, both Graphiti[8] and the Graphical Modelling Framework (GMF)[9] provide support through tooling for graphical editor customisation and generation for models; EMFText and Xtext provide users with the ability for textual modelling language definition; and Emfatic (created using Xtext) provides a textual language helping Ecore meta-model. Listing 2.1, is an example of the use of the Emfatic language to create the Ecore meta-model, depicted in Figure 2.2.

Ecore is the primary component of EMF, a meta-modelling language in the object-oriented paradigm. Definitions are found within itself, which has

[8]`http://www.eclipse.org/graphiti/`

[9]`http://www.eclipse.org/modeling/emft/emfatic/`

it described as a meta-meta-model, providing a layer of abstraction mapping to structures internal to Ecore, representing models as their equivalents in Java [161]. A diagram of Ecore high-level hierarchy—its concepts in the main level—is provided in Figure 2.6. The concepts and type system are heavily inspired by Java, even in third-party implementations such as for .NET [66], as the official implementation is in Java. The root element can be seen, EObject, with numerous type definitions available under EClassifiers, for example EEnum and EClass. EInt, EString, etc, constitute built-in types mapped verbatim to data types in Java. Similar to classes in Java, operations can be found on types along with attributes. Because of this approach, EMF is easy to understand and easily accessible to users with experience in object-oriented design [98]. However, the abstraction level is raised, which enables domain modelling to be the core focus rather than the details of implementation.

Epsilon - Extensible Platform of Integrated Languages for Model Management. Epsilon is an open-source, mature framework that includes a family of inter-operable languages used to manage diverse models based upon different meta-models and technologies [91]. The EOL [88] is a Epsilon's core imperative OCL-based expression language that supports model modification, flow control (branches, loops etc.), multiple model access, profiling, transactions, and user interaction. Although EOL is capable of general-purpose model management, its primary aim is being embedded in hybrid task-specific languages as an expression language [91]. Figure 2.7 shows the Epsilon's architecture and, an overview of the suite's available languages.

Moreover, Epsilon provides benefit in that it is a technology-agnostic suite for model management [160]. Any format can be used to express and manipulate meta-models and models by virtue of the EMC intermediate layer as shown in Figure 2.7). A novel file type or structure can be utilised as input through the implementation of a parser (called a "driver") which translates these for use by the EMC façade.

XMI - XML Metadata Interchange. There are numerous potential technology spaces, such as MDE, which uses MOF for its meta-meta-model; and XML, which uses XML Schema for its meta-metamodel [107]. It is therefore critical that a standardised method for modelling tools and frameworks exists to bridge technologies in data exchange, and to provide interopera-

```
@namespace(uri="compsMMExample", prefix="compsMMExample")
package comps;

class Model extends Component {
    val Component[*] components;
    val Connector[*] connectors;
}

class Component {
    attr String name;
    val Port[*] inPorts;
    val Port outPort;
}

class Port {
    attr String name;
    attr String type;
    ref Connector[1]#source outgoing;
    ref Connector[1]#target incoming;
}

class Connector {
    ref Port[1]#outgoing source;
    ref Port[1]#incoming target;
}
```

Listing 2.1: Emfatic code for defining the meta-model presented in Figure 2.2. The ref keyword is used to represent a standard reference, whereas the var keyword is used to represent a composition reference.

tion [35]. An XML format model developed using a modelling tool for UML, for example, can be imported into another modelling tool by first being converted to an XMI document.

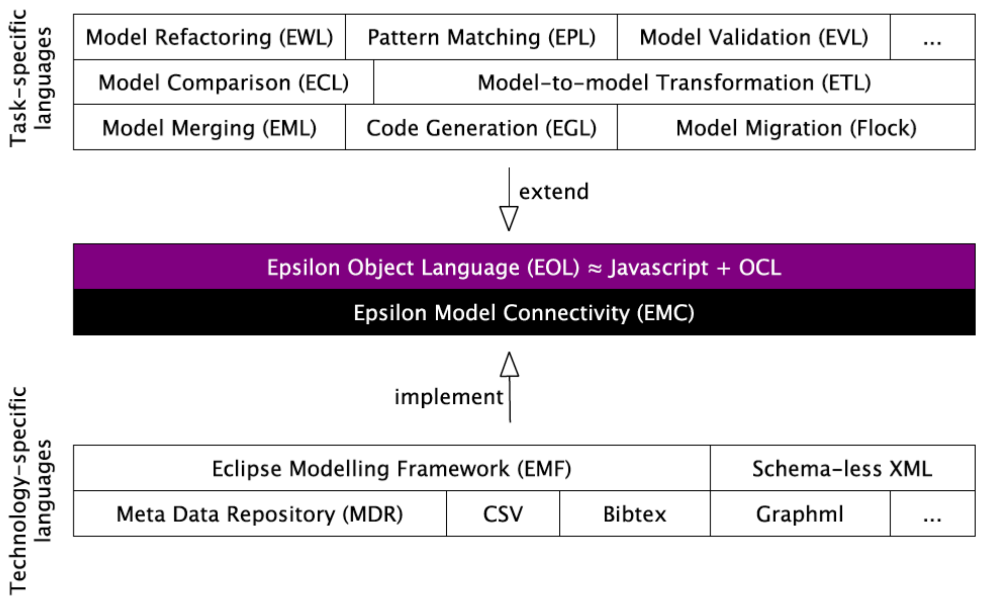


Figure 2.7: The environment of the Epsilon suite [85].

2.3 Integrating Hand-written Text in M2T Transformations

This section presents an overview of the importance of integrating hand-written text in M2T transformations. It also presents the existing mechanisms that are used to integrate hand-written text and discusses their advantages and disadvantages.

2.3.1 Introduction

As discussed in Section 2.2.1, M2T transformation is routinely used in model-based software engineering processes to generate implementation-level artefacts such as executable code, documentation and configuration scripts from abstract (typically domain-specific) models in an automated and repeatable manner.

A common way to implement M2T transformations is by using dedicated template-based languages such as Acceleo [44], Java Emitter Templates [43], Xpand [81], Velocity [52] and StringTemplate [114]. Similar to server-side scripting languages, such as PHP and ASP.NET, M2T languages provide first-class support for combining static content with text computed from the elements of one or more input models and can offer improved readability compared to imperative M2T transformation programs implemented using string concatenation [122].

Often, modelling languages do not provide sufficient expressive power to capture all the information required to achieve full code generation [22, 64, 53, 54]. In such cases, developers are called upon to complement the generated code with code that adds the missing information. There are different mechanisms to be used to integrate hand-written text as follows:

2.3.2 Existing Mechanisms for Integrating Hand-written Text

Greifenberg et al. [53] discussed eight mechanisms to integrate generated and hand-written text for OOP languages: generation gap, extended generation gap, delegation, include mechanisms, partial classes, AOP, PartMerger and protected regions. In the following, each mechanism is described in more detail.

In order to describe each mechanism, the following scenario is assumed: There is a small M2T transformation, where the input model is a UML class diagram (CD) that contains a Library class. Given that the CDs cannot specify the class behaviour, Library methods cannot be implemented at the model level. Thus, if the method for book method in Library class is to be implemented, it will need to be implemented manually using Java code at the code level. This is an example of the need to extend the generated text with hand-written text.

Generation gap is a pattern that keeps the hand-written and generated code separate by putting them in different classes linked by inheritance [48, 105]. It assumes that a default implementation and an interface are generated for each class in the source model. In this mechanism, the hand-written text is defined in a separate class (see Figure 2.8). Note, that here and in the following mechanism, (**M**) refers to the source model, (**gc**) refers to generated code and (**hc**) refers to hand-written text. The lack of ability to extend the generated interface, in addition to the necessity to create an implementation class, is considered the main disadvantage of this mechanism.

Alexandrescu [149] presented a technique called Generic Pattern Implementation (GPI), based on C++ templates and inheritance. This mechanism

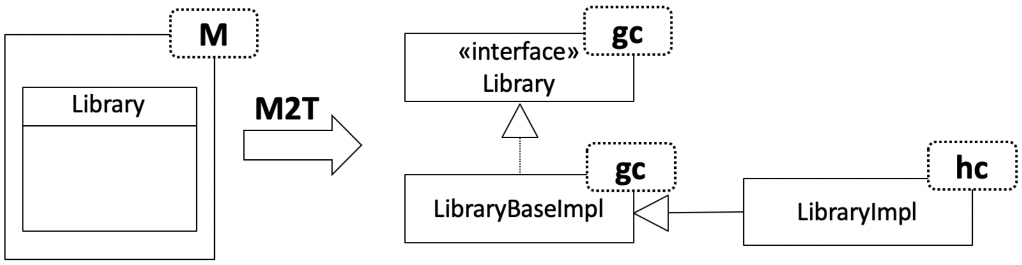


Figure 2.8: Generation gap pattern for the "Library" example.

enables a developer to tag a location in the code where a particular pattern is to be used. Then the code that is used to configure the concrete instance of that pattern can be generated automatically. This approach can help a developer use and recognise patterns, but it cannot be used if the language does not support the concept of inheritance (e.g., C, FORTRAN).

Karol et al. [38] proposed an approach for code reuse in an M2T transformation. Also, they demonstrated some advantages and also disadvantages of using the generation gap mechanism. For example, in their suggested mechanism, the automatically generated code is explicitly separated besides the ease of reusing the manually executed code fragments. However, the number of classes is high, and the generated code is also less understandable. In [67], the authors presented Prototizer, which is a tool that enables a boosted agile software development approach. During their experiments, they used the generation gap mechanism for integrating hand-written text into the generated code.

The main advantage of Prototizer is that it realises the MDE objective of centralising models, generating source code instantly, and improving prototyping abilities. This saves time and lets developers focus on core problems. The disadvantage of Prototizer is that any code generating technique typically introduces extra constraints for developers. For example, Protected regions, where developers are used to integrating hand-written text, can cause some issues [67]. Such integration be difficult to fit in them. Developers should duplicate code to accomplish the desired results.

Extended Generation Gap Pattern (EGP) is the mechanism that addresses the two defects of the basic generation gap mechanism [54]. As the generated interface "Library" extends the hand-written interface "LibraryBase", all methods that are added to "LibraryBase" are also available when accessing Library. Nevertheless, developers shouldn't have to add this hand-

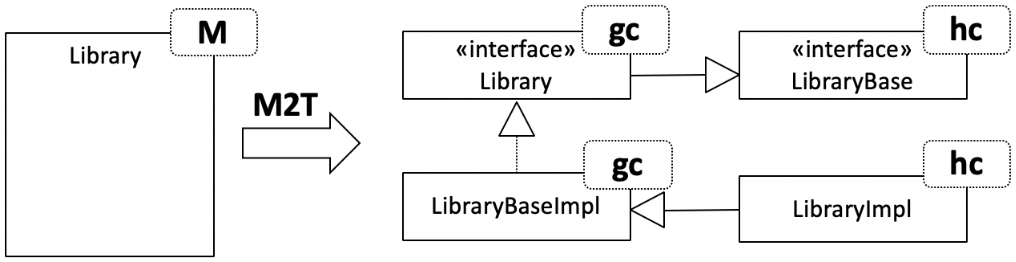


Figure 2.9: An example of extended generation gap pattern.

written interface. The reason for this is that the generator will check, at generation-time, if it exists or not. In the case that it does exist, the generated interface will extend the hand-written interface (see Figure 2.9). Thus, the generator needs to execute again after the hand-written interface is added to reflect this change in the generated code.

Delegation is an object composition pattern in object-oriented programming (OOP). It consists of two objects: the delegator and the delegate [49, 139]. Essentially, the "delegator" works by delegating parts of its functionality to the "delegate", by invoking the delegate's methods. To achieve this, the "delegate" provides an interface that declares the method signatures that can be invoked. "Library" is the "delegator" and "LibraryDelegateImpl" is the delegate implementing the methods that are defined in the "Library-Delegate" interface (see Figure 2.10).

Thomas et al. [153] described how different design patterns (e.g., delegation) can be adapted to integrate hand-written and generated code. They also stressed that the generated code should not be modified. Similarly, Bettin et al. [150] recommended avoiding integrating hand-written text to generated files, because consistency problems can occur (e.g., when the source model is changed in ways that make the hand-written text incompatible).

Include mechanisms rely on dedicated language constructs that allow specifying a certain file that should be included in another file at a specified point [65]. This mechanism can be used to easily merge the generated files with hand-written files, by including hand-written files in generated files at a specific place or vice versa (see Figure 2.11). Völter et al. [151] describe how an include mechanism can be used in C/C ++ to integrate hand-written text into generated code, along with an example.

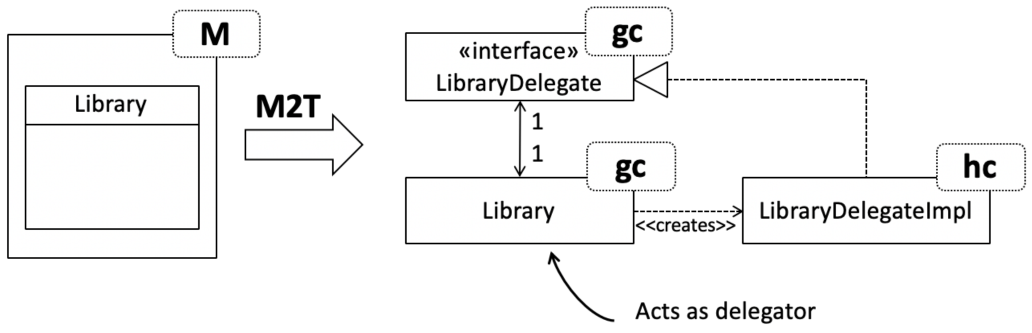


Figure 2.10: An example of delegation pattern.

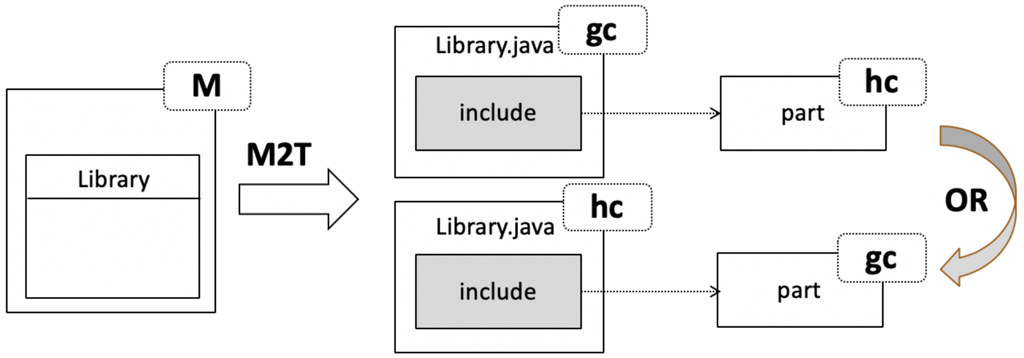


Figure 2.11: An example of how include mechanism works.

Partial class is a mechanism that aims to separate class implementations into multiple source code files. Then, all of these files are merged into a single class file during the compilation process. The result contains a grouping of all fields, methods, and supertypes of all its partial definitions [15]. Figure 2.12 illustrates the partial class mechanism.

Warmer et al. [155] developed a new application called SMART-Microsoft Software Factory. This is a complete model-driven factory that makes extensive use of the Microsoft DSL Tools. In their work they assumed that part of the system will be manually written in the code. Thus, they made extensive use of C# partial classes, to enable the developer to add hand-written text in separate files. The approach presented in [127] aimed to make the control flow in an explicit application in UML 2 activity diagrams able to preserve the possibilities of regular hand-written text implementing activity actions. Also, in the presented M2T transformation, they used partial classes to add the hand-written text into separate files.

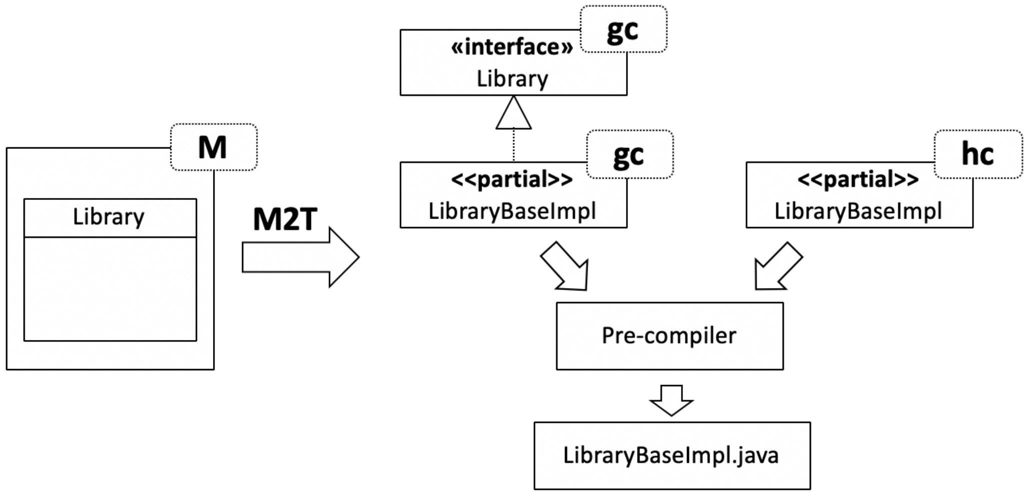


Figure 2.12: An example of how partial classes mechanism merges the hand-written and generated code to one single artifact.

Aspect-Oriented Programming (AOP) is a paradigm of programming that tries to improve modularity by the separation of cross-cutting concerns [80, 6]. AOP can be used to integrate hand-written and generated code, despite the fact that cross-cutting issues are not necessarily involved [129]. In this context, one advantage is that the generated code does not need to provide a specific architecture in order to be expanded by hand-written text [53]. The hand-written text is added through so called aspects (as shown in Figure 2.13) [53]. In addition to the added burden of integrating aspects into the source code, a key disadvantage of AOP is that it is difficult to comprehend the program flow when it is affected by aspects. In addition, refactoring in the source code may result in incorrect aspects, a phenomenon known as the fragile point-cut issue [77].

A PartMerger is a component that can combine many files of the same format, such as Java files, into a single file [53]. This idea is well suited for merging hand-written and generated parts, as these two parts can be in distinct files and PartMerger merges them into one file (see Figure 2.14). PartMerger mechanism is flexible, as it does not impose any restrictions on how to combine files. During the process of merging generated and hand-written text, PartMerger can give hand-written extensions more priority when two files are merged. In addition, different strategies can be used to invoke PartMerger and to define the files to be combined.

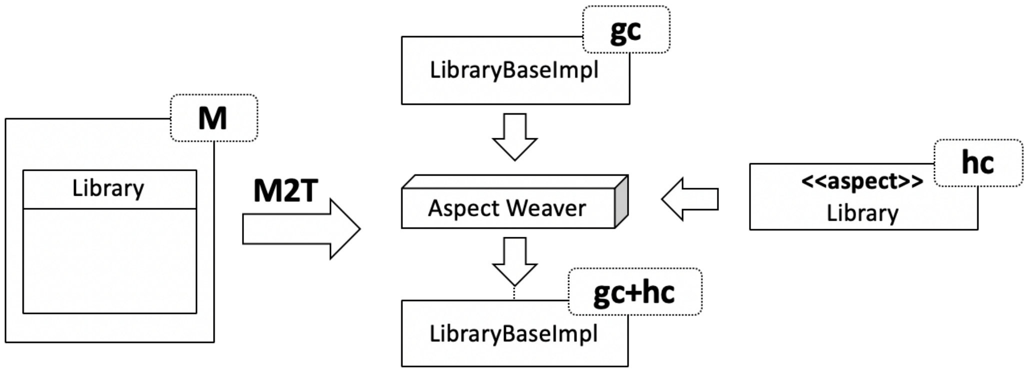


Figure 2.13: Overview of an aspect-oriented integration mechanism for a part of a generated software system.

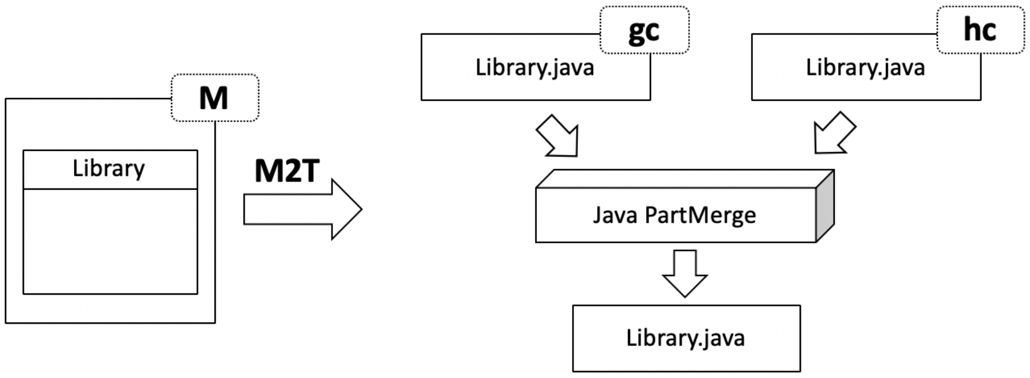


Figure 2.14: An example of how PartMerger mechanism merges source code artifacts such as Java code to one artifact.

One straightforward strategy is to automatically invoking the PartMerger for files that adhere to a certain naming convention at the artifact level, such as files with identical file names in specified folders. Another strategy is allowing developers to configure which files should be merged [53]. A disadvantage of this mechanism is the lack of tool support when editing hand-written text. Due to the strong separation between the generated and hand-written text files, typical capabilities such as code completion cannot be used to directly access portions of the generated code. Thus, if developers wish to take advantage of such tools, they should independently implement them.

Protected regions are regions that are declared in the M2T transformation templates by the developers for the purpose of adding hand-written text

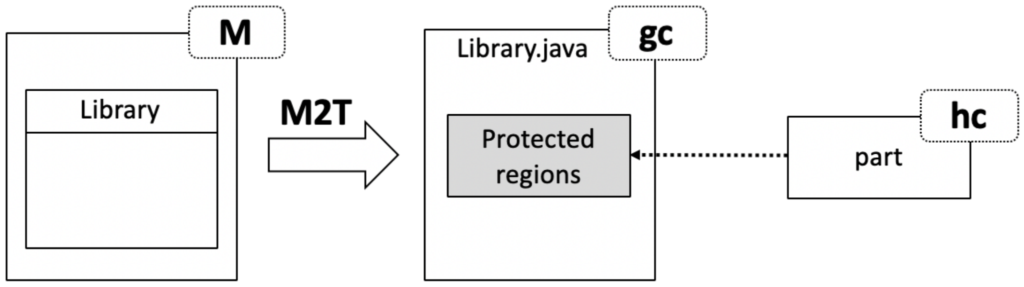


Figure 2.15: An example of protected regions mechanism in generated file.

within generated files (see Figure 2.15) [125]. Each protected region has its own comments that surround it, which contain a unique identification string to distinguish between different protected regions. Each generated file can contain one or more protected regions as necessary [51]. The contents of protected regions are preserved by the generator upon regeneration [23, 87]. However, if hand-written text is added outside the protected regions it will be lost during regeneration [95], which is one of the disadvantages of this approach. Many M2T transformation languages support the declaration of protected regions such as Epsilon Generation Language (EGL) [85], XPand [142] and Acceleo [44].

Stefan et al. [156] proposed a visual development environment for Jade – a popular framework for implementing a multiagent system (MAS) in Java. It focuses on how models and code can be synchronised. Their approach is based on adding protected regions into the source code where the user can insert the hand-written text. Thus, the user can modify the generated code without worrying about code regeneration. In contrast, in [152], the authors criticise the idea of using protected regions, as the user may lose the hand-written text upon regeneration. They suggested an alternative to it, which is separating hand-written text from the generated code using the composition features provided by the target language. However, there are many transformation languages that ensure that protected regions do not lose their content during regeneration (e.g., EGL and Acceleo).

JMerge is an open-source EMF utility [73] that enables code generators to integrate generated code with user-modified code, using XML-described rules. Java components decorated with the @generated Javadoc tag are subject to these rules. User alterations will be overwritten if they are made within code decorated by the @generated tag upon regeneration. However, user alterations will not be overwritten if the @generated tag is removed.

In the above, all the existing mechanisms for integrating hand-written text in M2T transformations and their limitations were discussed. However, using these mechanisms is not desirable because they require deciding in advance where hand-written text can be contributed (as demonstrated in the RC-1). A mechanism that overcomes the limitations of existing mechanisms, enabling developers the important freedom to include their hand-written text anywhere in the same file and without any protected regions (which must be declared in advance), would be valuable.

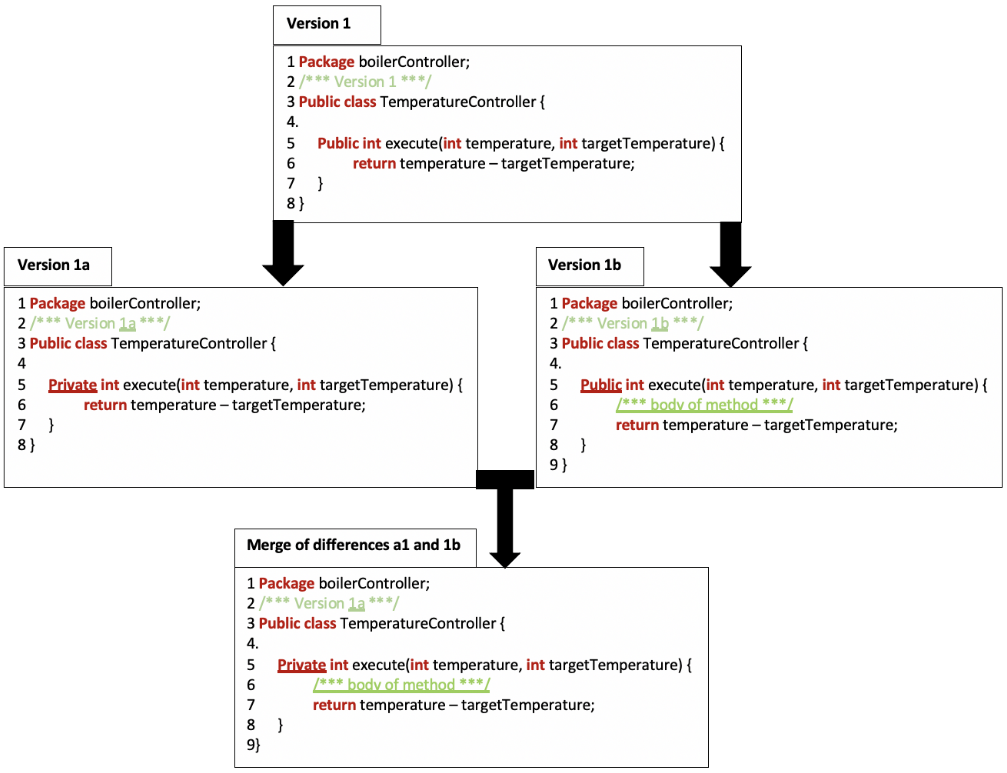
2.3.3 Two-Way Merging VS Three-Way Merging

Since this research focuses on the process of integrating handwritten text and generated text, this section investigates the areas of two-way merging and three-way merging.

Two-way merging is based on combining two versions of a software artefact without using the version that both versions came from. In contrast, in three-way merging, the original information that both versions came from is also used [97]. This makes three-way merging better than its two-way counterpart because it can find more conflicts. Because of this, most of the merge tools that are currently available are using three-way merging [101]. To illustrate the distinction between both two-way and three-way merging, consider the example of Figure 2.16, which depicts version 1 of Java class and two of its evolving variants (1a and 1b). The modifications that led to the evolution of each of these variants are given in underlined italics.

In two-way merging, just the differences between variants 1a and 2b are compared. For instance, line 3 in version 1b contains ***public int*** *execute(****int*** *temperature,* ***int*** *targetTemperature) {*, but line 3 in 1a contains ***Private int*** *execute(****int*** *temperature,* ***int*** *targetTemperature) {*. Likewise, version 1b contains */*** body of method ***/* (line 6), that version 1a does not contain. This information is inadequate to determine whether the discrepancies are the result of a line deletion, addition, or modification in just one of the evolved versions, or a simultaneous modification in both versions.

This shortcoming is not present in three-way merging [101]. For instance, the line ***public int*** *execute(****int*** *temperature,* ***int*** *targetTemperature) {* of version 1b is also present in original version (version 1), indicating that only version 1a modified this line. Likewise, since the line */*** body of method*



```
Package boilerController;
/*** Version 1 ***/
Public class TemperatureController {

    Public int execute(int temperature, int targetTemperature) {
        return temperature – targetTemperature;
    }
}
```

```
Package boilerController;
/*** Version 1a ***/
Public class TemperatureController {

    Private int execute(int temperature, int targetTemperature) {
        return temperature – targetTemperature;
    }
}
```

```
Package boilerController;
/*** Version 1b ***/
Public class TemperatureController {

    Public int execute(int temperature, int targetTemperature) {
        /*** body of method ***/
        return temperature – targetTemperature;
    }
}
```

```
Package boilerController;
/*** Version 1a ***/
Public class TemperatureController {

    Private int execute(int temperature, int targetTemperature) {
        /*** body of method ***/
        return temperature – targetTemperature;
    }
}
```

Figure 2.16: An example of a three-way merging.

***/, of version 1b; did not exist in original version (version 1), it must have been established in version 1a. The merge algorithm uses this additional information to determine which lines from versions (1a and 1b) should be included in the final merged version. On the below of Figure 2.16 is a possible result of the three-way merging. The merge integrates all modifications, additions, and deletions from both. versions (1a and 1b), and when a merge conflict occurs, the changes of version 1a take priority.

2.3.4 Discussion

In Section 2.3.2, an overview of the existing mechanisms for integrating hand-written text in M2T transformations was presented. Some of these mechanisms require a clear separation between hand-written and generated code: generation gap, extended generation gap, delegation, include mechanisms, partial classes, AOP, and PartMerger mechanisms. For example, the generation gap mechanism prescribes that a default implementation and an interface are generated for each class in the source model. Thus, the hand-written text is defined in a separate class (see Figure 2.8). Some benefits arise in separating hand-written and generated code, such as the mitigation of generated code becoming polluted, and enabling the editing of just a separate file for hand-written changes. However, a drawback is that system information may be spread between two or more places, leading to developer confusion from the extra files.

Other mechanism proposals involve merging hand-written text merged into generated code, such as using JMerge and/or protected regions. For example, the protected regions mechanism works by inserting hand-written text inside tagged blocks within generated files, eliminating the need for additional files. However, using such a mechanism, the generated code will be polluted with hand-written text and the comment/tags for protected regions, as each protected region will have two extra lines: the beginning and the end comments of the regions. Another major limitation of using a protected region mechanism is that if the developer wishes to change any part of the class, such as adding a new variable or a new field, they may be unable unless an appropriately positioned protected region is present.

To summarise, using these existing mechanisms does provide benefit to developers who separate hand-written text from generated code, and JMerge and protected region mechanisms provide benefit to developers who wish to merge the hand-written text within generated code, albeit with some restrictions. A mechanism that overcomes the limitations of existing mechanisms, enabling developers the important freedom to include their hand-written text anywhere in the same file and without any protected regions (which must be declared in advance), would be valuable.

2.4 Round-Trip Engineering (RTE)

This section presents an overview of round-trip engineering (RTE) in the context of MDE. It also presents existing mechanisms for supporting RTE and examines their limitations.

2.4.1 Round-Trip Engineering

Round-trip engineering (RTE) is the ability to automatically preserve the consistency of various changing software artifacts in software development environments/tools [63]. RTE is one aspect of MDE, because the target generated code and the source model are interrelated; altering the generated code will affect the source model and vice versa. It consists mostly of forward and reverse engineering.

Forward engineering is the process of converting conceptual models into source code, whilst reverse engineering is the process of converting source code into conceptual models [104]. Forward and reverse engineering optimisation results in gradual transformation. Not all artifacts are converted, but just the altered modules. The combination of the two approaches results in round-trip engineering (RTE), which keeps the two perspectives consistent [19]. In software development, generating code from source models and then executing round trip engineering are critical processes. Throughout the development process, it is vital that software-related artifacts such as models and their source code remain in sync [104].

Ducasse et al. [37] describe RTE as the integration of design diagrams and source code, as well as modelling and implementation. The purpose of RTE is therefore to ensure a seamless interaction between the design and execution stages. Similarly, Lenk et al. [96] agree that RTE is a software development method that includes automated forward (model-to-code) and reverse (code-to-model) transformations. Model round-trip engineering involves synchronising models and their generated artefacts by maintaining consistency, allowing the software developer the flexibility to switch between these different representations [134].

Benefits of Round-Trip Engineering

The primary benefit of RTE is that both design and implementation artefacts are constantly and automatically synchronized [78]. RTE encourages design driven development and enhances design traceability by enabling both the automated generation of source code from conceptual models and the automatic generation of conceptual models from source code. RTE enhances both the software development process and its automation. RTE techniques possess certain properties, including the capacity to handle trace information and to assist in the discovery of conflicts among RTE tasks.

According to Akoka et. al. [5], RTE can be viewed as a process that enables improved software engineering processes. The ability to have both forward and reverse engineering transforms in the software design process enables benefits such as transforming the source code into its conceptual models. In [5], the authors presented an approach that supports RTE in M2T, where better traceability of code being changed in software arises as one of the benefits of RTE in software engineering. A programmer can perform changes to the code in the software, and quickly trace the code changes that other developers may apply with the use of RTE. Reverse engineering also allows for easier removal of code that may contain errors. However, the application of RTE in the software development process still leaves the software engineer with considerable work to do, for example, keeping track of the changes being made to the generated artefacts.

The application of RTE in software engineering also reduces complexity. In a study by Ciccozzi et al. [31], RTE can be applied to reduce the complexities usually found in embedded system design and development, which also supports the ability to employ RTE in M2T. Larger software development organisations typically have shorter development cycles and higher expectations. As a result, they are constantly searching for methods to reduce the complexity of software development [60]. The application of model-driven round-trip engineering will help software engineers with the deployment of component-based systems in the telecommunication sector [31]. The ability to reduce the complexity of a system also reduces the time taken by software engineers to develop embedded systems in the industry.

The effort of keeping different artifacts consistent in the design and development of code is a time-consuming activity that is even longer when undertaken manually [135]. Rocha et al. [121] agree that evaluating and ensuring consistency in modelling artifacts is a time-consuming issue in software design. The probability of software developers making errors is also higher, and it takes a longer time to complete a software system under development. In [134], the authors identify that RTE in M2T is an essential mechanism that ensures consistency and synchronisation in different M2T transformations. RTE can be used to automate the process of ensuring consistency in the multiple models and artifacts being employed in software. Changes created in one model can also be propagated and reflected in another model quickly and efficiently [134]. RTE's structured approach to automation ensures that there is less effort spent in monitoring and maintaining consistency between artifacts. Ciccozzi et al. [31] agree that RTE results in reduced functions, which in turn lead to lower costs and more efficiency in software development.

2.4.2 Classification of Round-Trip Engineering

Round-trip engineering can be divided into partial and full round-trip engineering [116] as described below:

2.4.2.1 Partial Round-Trip Engineering

In order to prevent code generators from overwriting hand-written text, partial round-trip engineering is used. Various partial round-trip engineering approaches have been suggested to segregate the code generated by the model from the hand-written text, which may be manually updated. The following describes some of existing solutions proposed for partial round-trip engineering.

Protected Regions. Protected regions are an alterable portion of the generated code. This solution was described in detail along with examples in previous section 2.3.2.

Embedded Snippets Code Directly within Model. Many UML tools, for example Enterprise Architect [136], IBM Rhapsody [69], Papyrus-RT [118], enable fine-grained code for behaviour to be embedded into the model. This code contains all the necessary information within a single source file, which overcomes the hindrance of protected regions. This technique is consistent with certain approaches of information confinement. However, modifications to the fine-grained code are made at model level. Thus, during code change it prevents programmers from using their preferred programming editors and IDEs (and the benefits the latter comes with, like code completion, syntax highlighting, etc.), which may lead to their efficiency and productivity suffering.

2.4.2.2 Full Round-Trip Engineering

This section refers to methods that allow for modifications in both the code and the model. Typically, this form of round-trip engineering requires the use of a mechanism of synchronisation that can handle both model to code and code to model change propagation. In reality, this synchronisation is difficult to achieve since it requires at the very least bidirectional model-to-code mapping.

Fujaba. Fujaba [82] supports story-diagrams, a high-level visual programming language that combines both collaboration and activity diagrams. The specifications and control structures of complicated objects that are application specific are expressed using story-diagrams. Story-diagrams are used to visualise a system's dynamic characteristics, such as a control flow, or to adapt UML classes, collaboration diagrams, and activity, and by raising the degree of abstraction be translated into executable Java code. Furthermore, as a round-trip engineering environment, Fujaba extracts abstract syntax graphs of source code in Java to generate story- and class-diagrams. Editing the resulting code is permitted, enabling round-trip engineering providing the developers adhere to Fujaba's naming standards and implementation approaches.

Annotations are inserted step by step in the back end to recreate Fujaba story-diagrams. Multiple annotation engines look for patterns expressed in the syntax tree (such as the names of certain methods) and, as a result, add annotation elements into the tree. Following this, Fujaba attempts to ascertain graph rewriting rules that are similar to the annotations observed. The result of this re-engineering process is new graph rewriting rules, which correlate to the story-diagrams.

However, Fujaba has a number of limitations. Firstly, the back end of the parser must be hand-written. As a result of this, the process of design recognition (such as the story-diagrams) is prone to errors, so there is no assurance that the same code will result from re-engineering a diagram when it is created again. Secondly, the method relies heavily on the Fujaba code generation algorithms, and principally of Java itself. The back end of the parser requires a thorough understanding of how the code was created in the first instance. Because the back end of the parser, and the parser itself, are Java-specific, they cannot be used with any alternative programming language, with the support for RTE being hard-coded. Finally, the RTE approach is unable to handle code sections not depicted in diagrams. By using its story-diagrams to store method bodies, Fujaba attempts to circumvent this difficulty. However, after regenerating the code, fragments that cannot be recognised by the reverse engineering process are absent.

SelfSync. In [147], the authors suggest SelfSync as a round-trip engineering environment. Its goal is to synchronise data modelling perspectives with its corresponding Self object-oriented language implementation objects [146].

It is presumed that the information in the implementation and data modelling perspective is the same. There are potentially four possibilities for round-trip engineering. The first is where view entities are changed, as is the corresponding implementation; second is when view entities' relationships are changed, as is the implementation; the third possibility is when implementation objects are changed, as is the data modelling view; and lastly when implementation object relationships are changed, as is the data modelling view. Furthermore, updating is a real-time process to the implementation from the modelling view, meaning that changes in the modelling view are immediately reflected in the implementation.

Framework-Specific Modeling Language (FSML). The synchronisation of Eclipse plug-in source code, and domain-specific models, was studied by Antkiewicz and Czarnecki [9]. These models conform to an FSML that enables the interaction between Eclipse workbench components to be described. Framework extensions are properly completed by the framework due to restrictions on the FSML models. Agile RTE, the method described in [9], combines partial (human) reverse engineering and forward engineering, which is automatic. In the former, from the code, the recently extracted model is compared to the most recent model used in the process of forward engineering. Consistent modifications are disseminated, to the FSML model or code, as a result of the comparison. Conflicts are indicated by inconsistent modifications, for example through developer modification of the model and source code, which consequently require manual correction.

From the source code, FSML models are derived through reverse engineering in an approach described in [9], converting a scenario of heterogeneous synchronisation to be homogeneous instead. The actual synchronisation is then carried out via a three-way merging of the asserted, the freshly extracted, and the prior FSML model. The difference between consistent and inconsistent modifications is established by examining the code modifications, and those modifications that affected the asserted model. While the latter requires human intervention to return to a consistent condition, the former can be done automatically.

Model-and-Code Consistency Checking (MCCC). During the development and co-evolution of the source code and the design model, model-code consistency checking can assist in uncovering discrepancies. The authors in [120], suggest that in MBSE, model and code are updated often and simultaneously. As a result, it is important to check the consistency between the maturing artefacts. They present a technique of consistency checking that is incremental and discovers model and code conflicts in real-time. Their technology combines Java code with UML models into a single in-memory representation. This is deployed by consistency testing, based on rules supplied in constraint languages by developers, for example in the object constraint language (OCL). During development, testing identifies and reports the state of project consistency to the developers. If discrepancies are found, they are not resolved using a synchronisation method.

Syntactic Model-Code Round-Trip Engineering. Angyal et. al., in [7], suggest an RTE technique for synchronising source models and generated code. The purpose is to synchronise a DSML-compliant model with code to allow for iterative development. The models and code can be changed at the same time in the latter, and the suggested RTE synchronises the changes throughout development. They suggest employing a three-way strategy for platform-specific model and code synchronisation that is extremely close to the code's Abstract Syntax Tree (AST). After that, the platform-specific model is kept in synchronisation with the DSML-compliant model.

2.4.3 Discussion

In the above section, an overview of existing mechanisms for supporting synchronisation/RTE in M2T transformations has been presented. All of the mechanisms reviewed for supporting RTE have some limitations. For example, Fujaba can be used when the source model is captured in UML. However, the RTE process cannot be automated as the parser back-end needs to be written manually. it is also very dependent on its own code generation methods and on Java, and it is unable to deal with features in code which are not reflected in diagrams.

In SelfSync, a source model and its associated implementation object are identical. Both have the same structure and behaviour, specifically the prototype's structure and characteristics. Although the synchronisation process in SelfSync can be undertaken automatically, SelfSync can only be used with an Extended Entity-Relationship diagram. FSMLs will typically reflect the concerns of a small area of the framework, with multiples being used within a single framework. However, one limitation of this method is that the source model must be generated from the source code each time, mitigated only by objectionable use of code annotations and similar techniques.

The MCCC framework, although it can robustly detect inconsistencies between the design model and source code, it requires human intervention to write complex consistency rules such as in OCL. The syntactic model-code RTE method enables both hand-written code and generated code parts within the generated artifacts, rather than them being separate. However, it is an expensive method, using a three-way merging strategy that is more complex than simple template-based approaches. It also only works with specific types of models. And finally, it is complicated to use, as AST and DSML models involved have a large abstraction gap.

To summarise, using these existing mechanisms does give developers the benefit of providing support in the RTE process, but some of them are very complex and others can only be used in specific cases.

2.5 Summary

The first part of this chapter presented fundamental Model-Driven Engineering (MDE) concepts including the definitions of meta-models, models, modelling languages and their essential components such as abstract syntax, concrete syntax, and semantics. Several model management activities, including model querying, model comparisons, model validations, and model transformations (such as model-to-model, model-to-text, and text-to-model) were also discussed. along with a summary of contemporary MDE challenges, including merging and synchronisation.

The second part of this chapter presented an overview of integrating hand-written text in M2T transformations. It also summarised existing approaches that are used to integrating hand-written text in M2T transformations. This part concluded with a discussion on the limitations of these existing approaches. Finally, this chapter presented an overview of round-trip engineering (RTE); more specifically, RTE in M2T transformations. A summary of the existing methods that are used to support round-trip engineering in M2T transformations was given along with a discussion on the limitations of these existing methods.

In the next chapter, we will analyse the research problems identified in integrating hand-written text in M2T transformation and illustrate those through an example M2T transformation.

Chapter 3

Analysis and Hypothesis

In Chapter 2 methods for integration of hand-written text in M2T transformations were reviewed. Different techniques were discussed, and further research opportunities were described. There are evident research challenges relating to the existing mechanisms for integrating hand-written text in M2T transformations. However there are also additional challenges in the inability of M2T languages to support round-trip synchronized engineering. This chapter addresses both issues together with the hypothesis and anticipated outcomes of the thesis.

Chapter Structure: An overview of the target research area is discussed in Section 3.1. Section 3.2 offers an example to demonstrate the challenges targeted by the research. Section 3.3 contends that there are drawbacks and limitations in using existing integration approaches. And finally, Section 3.4 contains the hypothesis and objectives of this research.

3.1 Analysis

Model-based software engineering processes habitually use model-to-text (M2T) transformation to create implementation-level artefacts such as executable code, documentation and configuration scripts from abstract, typically domain-specific, models. This is done in an automated and repeatable manner. M2T transformations may be routinely implemented using dedicated *template-based* languages such as EGL [124], Acceleo [44], Java Emitter Templates [43], Xpand [81], Velocity [52] and StringTemplate [114].

They combine the static content with text computed content from the elements of one or more input models and can offer improved readability compared to imperative M2T transformation programs implemented using string concatenation [122]. It is often the case that modelling languages cannot provide sufficient expressive power to capture all the information required to achieve full code generation. In cases such as this, one or more of the following options may be chosen by developers:

- To augment the generated code with code that is to add the missing information, using *protected regions* or inheritance/delegation;
- Extend the abstract and concrete syntax of the modelling language with concepts required to capture the missing information within the model, ideally at an implementation-agnostic level of abstraction;
- Minimally extend the modelling language to allow modellers to embed code fragments written in the target implementation language within their models (e.g. embed Java code within UML models).

Existing integration mechanisms were discussed in Chapter 2. These mechanisms can depend on separating the hand-written text from the generated code, while in protected regions and JMerge, the generated code is combined with the hand-written text in the same file. However, it is often the case that additional issues are created when generated and hand-written text are combined together. The next section highlights these issues and offers possible solutions.

3.2 Motivating Example

In order to clearly demonstrate the problems that are being targeted in this research and to illustrate the proposed approach, an example was developed using a minimal model-to-text transformation. This example was chosen because it involves integrating handwritten and generated text. At the same time round-trip synchronisation between the source model and the generated source code was a hoped-for outcome.

For this example, we use a minimal component-connector domain-specific language (DSL), the abstract syntax of which is illustrated in Figure 3.1. In the DSL a system consisted of components and connectors. Each component has many input ports (*inPorts*) and one output port (*outPort*). Each port has a name and a type, and ports can communicate through connectors. Each connector has exactly one source port and one target port.

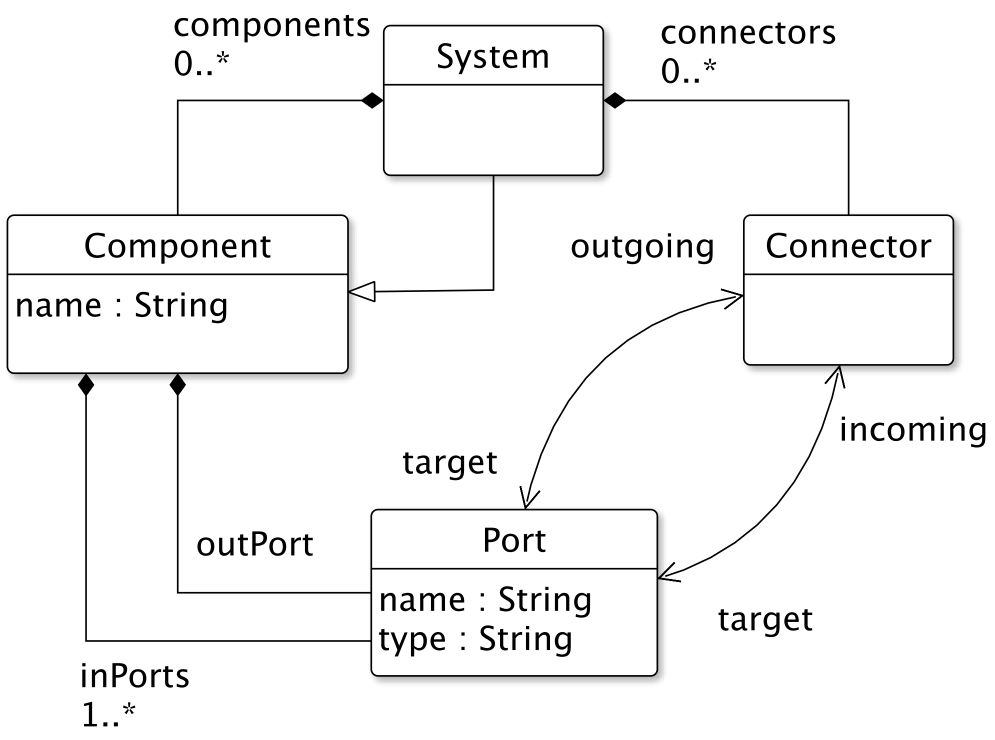


Figure 3.1: Meta-model of the Component-Connector DSL.

Figure 3.2 shows a model that conforms to the DSL and which captures a small part of the operation of a water heating boiler. The system (model) has two components: *TemperatureController* and *BoilerActuator*. It also has three input ports (namely, *temperature*, *targetTemperature*, and *boilerStatus*). The *TemperatureController* component receives input from two ports (*temperature* and *targetTemperature*). It computes the difference between the two and the result is propagated to the *BoilerActuator* component along with the then current status of the boiler. The *BoilerActuator* component decides whether to turn the boiler on or off.

From models like the one shown in Figure 3.2, we wish to generate executable Java code. This is achieved through a template-based M2T transform-

```
1  rule Model2Class
2      transform m : Model {
3      template : "../common/model2class.egl"
4      target : "src-gen-sync-regions/syncregions/" + m.name + ".java"
5  }
6
7  rule Component2Class
8      transform c : Component {
9      template : "sync-regions-component2class.egl"
10     target : "src-gen-sync-regions/syncregions/" + c.name + ".java"
11 }
```

Listing 3.1: EGL rules for generating Java code from component-connector models

ation, implemented using Epsilon Generation Language (EGL) [1], as shown in Listings 3.1-3.3.

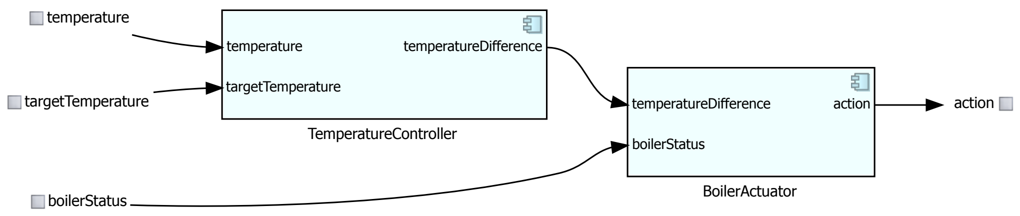


Figure 3.2: BoilerController model that conforms to the meta-model in Figure 3.1.

The program in Listing 3.1 consists of two rules. The first, in lines 1-5 is used to generate a Java class for every model element of type *System*. Line 1 gives the rule a name; line 2 contains the name of the type, instances of which the rule should transform. Line 3 declares the template that will be used for the transformation, and line 4 specifies where the generated file will be stored. The second, in lines 7-11 is used to generate one Java class for each component in the system.

The template invoked by the *System2Class* rule is shown in Listing 3.2. Line 1 prints the class name. Lines 2-8 generate an *execute()* method that has one parameter for each input port of the system and returns a value, the

[1]Although we use EGL in this example, the transformation could be implemented using any other template-based M2T language

```
1  public class [%=m.name%] {
2      public [%=m.outPort.type%] execute([%=m.inPorts.collect(p|p.type +
           " " + p.name).concat(", ")%]) {
3      [%for (child in m.components){%]
4      [%=child.name%] [%=child.name.ftlc()%] = new [%=child.name%]();
5      [%=child.outPort.type%] [%=child.name.ftlc()%]Result = [%=child.
           name.ftlc()%].execute([%=child.getInputParameters().concat(", ")
           %]);
6      [%}%]
7
8      return [%=m.outPort.incoming.source.eContainer().name.ftlc() + "
           Result"%];
9      }
10 }
11 [%
12 operation Component getInputParameters(){ {
13     var parameters : Sequence;
14     for (p in self.inPorts) {
15       if (p.incoming.source.eContainer().isTypeOf(Model)) {
16          parameters.add(p.incoming.source.name);
17       }
18       else {
19          parameters.add(p.incoming.source.eContainer().name.ftlc() + "
                Result");
20       }
21     }
22     return parameters;
23 }
24 %]
```

Listing 3.2: EGL template that generates a Java class realising the communication between components of the system

type of which is the same as the type of the output port of the system. The list of the input parameters for each component is calculated using a utility operation *getInputParameters()* defined in lines 12-24.

The second template is for the *Component2Class* rule and is shown in Listing 3.3. Line 1 prints the class name and lines 2-4 generate an *execute()* method for the component with appropriate input parameters and return type, and an empty body. When the transformation on the model of Fig-

```
public class [%=c.name%] {
    public [%=c.outPort.type%] execute([%=c.inPorts.collect(p|p.type + "
        " + p.name).concat(", ")%]) {

    }
}
```

Listing 3.3: EGL template for generating Java class for each individual component

```
public class BoilerController {
    public int execute(int temperature, int targetTemperature, boolean
        boilerStatus) {
    TemperatureController temperatureController = new
        TemperatureController();
    int temperatureControllerResult = temperatureController.execute(
        temperature, targetTemperature);
    BoilerActuator boilerActuator = new BoilerActuator();
    int boilerActuatorResult = boilerActuator.execute(
        temperatureControllerResult, boilerStatus);

    return boilerActuatorResult;
    }
}
```

Listing 3.4: Generated class for BoilerController component

ure 3.2 is executed, it produces the files shown in Listing 3.4 and 3.5 for the system, and the *BoilerActuator* component, respectively[2].

While the model contains sufficient information[3] to generate the content of the *execute()* method of the *BoilerController* as per Listing 3.4, it has no means of expressing the behaviour of each individual component. Hence, the generated *execute()* methods of the *BoilerActuator* and *TemperatureController* classes in Listings 3.5-3.6 respectively is empty.

[2]A very similar class is generated for the *TemperatureController* component, which we omit to reduce unnecessary repetition.

[3]With many assumptions e.g. regarding ordering and freedom from cycles which are necessary to keep this example minimal.

```
public class BoilerActuator {
    public int execute(int temperatureDifference, boolean boilerStatus) {

    }
}
```

Listing 3.5: Generated class for BoilerActuator component

```
public class TemperatureController {
    public int execute(int temperature, int targetTemperature) {

    }
}
```

Listing 3.6: Generated class for TemperatureController component

3.2.1 Problems

To add the missing behaviour for *execute()* methods in Listings 3.5 3.6, one could extend the generated code with hand-written text using inheritance or delegation techniques, if the developer prefers adding them in separate file. They can even be directly added within the generated files using *protected* regions and JMerge techniques, if the developer prefers adding them in the same file. There are two limitations that apply to this example as described below:

- One limitation is that, when a developer integrates hand-written text within auto-generated text using existing approaches (e.g. protected regions), this can lead to some issues. In section 3.3, an example is shown of how protected regions can be used to integrate hand-written text using EGL language to demonstrate these issues.

- Another limitation is that when a developer integrates hand-written text into target generated artefacts, whether by separating them into different files (using a technique such as inheritance) or merging them within auto-generated files (using a different technique such as protected regions), the integration can lead to a lack of consistency between the source models and their generated artefacts in M2T transformations. This limitation is discussed in more detail in Section 3.3.3.

```
public class [%=c.name%] {
    public [%=c.outPort.type%] execute([%=c.inPorts.collect(p|p.type + "
        " + p.name).concat(", ")%]) {
        [%=out.startPreserve("//", "", "execute", true)%]

        [%=out.stopPreserve()%]
    }
}
```

Listing 3.7: Declaring protected regions using EGL language

3.3 Limitations of Integrating Techniques in M2T Transformations

Although there are different techniques that can be used to integrate hand-written text, the focus of this research is solely on the techniques that allow the developer to integrate hand-written text within auto-generated artefacts (e.g. protected regions). Thus, this section describes how the developer can integrate hand-written text using protected regions technique accompanied by an example. Also, it discusses the issues of using protected regions.

3.3.1 An example of Using Protected Regions in EGL Language

In Listing 3.5, if developers want to add the body of the*execute()* method and preserve it during the re-generation then using a protected region is a viable option. Protected regions can be declared using EGL as shown in Listing 3.7 and the output of executing it against the *BoilerActuator* and *Temperature-Controller* components are shown in Listings 3.8 3.9 respectively.

Then, the developer can specify the behaviour of the *BoilerActuator* and *TemperatureController* components within the produced protected region for the generated *BoilerActuator* Java class, as shown in lines 4-9 of Listing 3.10 and the generated *TemperatureController* Java class, as shown in line 4 of Listing 3.11 respectively.

3.3.2 Issues of Using Protected Regions

Using the protected regions technique can lead to the following issues:

```java
public class BoilerActuator {
    public int execute(int temperatureDifference, boolean boilerStatus) {
        // protected region execute on begin

        // protected region execute end
    }
}
```

Listing 3.8: The result of executing the template of Listing 3.7 against the *BoilerActuator* component

```java
public class TemperatureController {
    public int execute(int temperature, int targetTemperature) {
        // protected region execute on begin

        // protected region execute end
    }
}
```

Listing 3.9: The result of executing the template of Listing 3.7 against the *BoilerActuator* component

```java
public class BoilerActuator {
    public int execute(int temperatureDifference, boolean boilerStatus) {
        // protected region execute on begin
        if (temperatureDifference > 0 && boilerStatus == true) {
            return 1;
        } else if (temperatureDifference < 0 && boilerStatus == false) {
            return 2;
        } else
            return 0;
        // protected region execute end
    }
}
```

Listing 3.10: Extended *BoilerActuator* class with behaviour

```
1 public class TemperatureController {
2     public int execute(int temperature, int targetTemperature) {
3         // protected region execute on begin
4         return temperature - targetTemperature;
5         // protected region execute end
6     }
7 }
```

Listing 3.11: Extended *TemperatureController* class with behaviour

Anticipate Location in Advance. Developers have to anticipate in advance which part of the code they want to extend. For example, in Listings 3.8 3.9, there are protected regions inside the body of the methods; any changes that the developers want to make can only be undertaken inside the method body. However, since there is only one protected region, they are unable to change any part of the class, including adding a new field.

Code Pollution. The code will be polluted with protected region comments as shown in Listing 3.10 (lines 3 and 10) and Listing 3.11 (lines 3 and 5), especially when a large number of such regions are added. In addition, if the developers mistakenly add the code outside the protected region's tags, important system information will be lost during re-generation.

3.3.3 Limitations of Consistency between Source Models and Their Generated Files

While integrating hand-written text is a common task in M2T transformations, it can lead to inconsistency between the source models and generated artefacts. Also, there is no single source of truth, since the system information is spread in two or more places.

No Single Source of Truth. Protected regions are detrimental in terms of model analysability and portability. However they remain a very popular approach among practitioners because the implementation cost is low, the target implementation language is familiar, and there is usually an aversion to complicating the syntax of the modelling language.

Despite this popularity, there are those who wish to use the target language as the single source of truth for their development. Writing the code within the modelling environment deprives them of essential features such as code completion and error reporting. Alternatively, writing the code within an IDE incurs the additional overhead of having to copy and paste it back to the modelling tool, and also includes a risk that this step is missed and that consequently code fragments are accidentally overwritten next time the M2T transformation is executed.

3.4 Research Hypothesis and Objectives

This section presents the research hypothesis in Section 3.4.1 and the objectives of the research in Section 3.4.2. Section 3.4.3 clarifies the scope of the research.

3.4.1 Research Hypothesis

There are two different research hypotheses of this thesis as follows:

> *The first hypothesis of this thesis is that it is possible to integrate and preserve **hand-written text** in generated files without needing to use **protected regions** or similar constructs in Model-to-text transformations (M2Ts). The second hypothesis is that where embedding code fragments in models is necessary to achieve full code generation, the content of these fragments can be **automatically synchronised** between the model and the generated code in Model-to-text transformations (M2Ts).*

The highlighted terms are the characteristics that were derived from the research hypothesis and used in the construction of the context for this research project. They are defined as follows:

1. **Hand-written text:** The process of modifying generated artefacts manually by users.
2. **Protected regions:** Protected regions are regions that are declared in the M2T transformation templates by the developers for the purpose of adding hand-written text within generated files.

3. **Automatically synchronised:** Any modification that occurs in generated target artefacts is reflected back to the source models. Thus, ensuring both the source models and the generated target artefact are consistent.

3.4.2 Research Objectives

The research objectives of this thesis can be summarised as follows:

1. **First Research Objective (RO-1):** Enable language-agnostic preservation of text in arbitrary locations of generated files without the need for protected regions.

2. **Second Research Objective (RO-2):** Enable automated round-trip synchronisation of code fragments embedded in generated files with the source models of the transformation.

3. **Third Research Objective (RO-3):** Assess the performance of the proposed mechanisms.

3.4.3 Scope

The scope of this research is limited to deterministic template-based model-to-text transformations that consume a single input model.

Given the importance of integrating hand-written text in M2T transformations, many mechanisms have been developed to facilitate the integrating process in M2T transformations. The shortcomings of these mechanisms are discussed in Section 2.3.

Also, despite the potential benefits of processes maintaining consistency in M2T transformations (for example, keeping source models and generated files synchronized), most M2T transformation languages still do not support automated consistency management as discussed in Section 2.4.

In light of the above, this research is limited to providing techniques to facilitate the process of merging and maintaining synchronization that can be applied to M2T languages and does not consider a general solution that can be applied to all M2T languages.

Chapter 4

Automated Line Based Merging

4.1 Introduction

This chapter describes the first contribution of this thesis, an approach for adding hand-written lines of code anywhere in generated files, and for preserving them upon re-generation. In the previous chapter, some of the problems related to the use of the protected regions technique to integrating hand-written text were described. One of these problems was that developers have to anticipate in advance which part of each generated file will be extended with hand-written text. For example, in Listing 3.8, there is a protected region inside the body of the method; if the developers want to make changes, they can only do so inside the method body. However, they will not be able to change other parts of the class, such as adding a new field, because there is only one region; they need to add additional regions each time they want to integrate hand-written text elsewhere.

This chapter presents a technique developed in the context of this project that enables developers to integrate hand-written text directly within the generated code without having to use predefined protected regions. The proposed approach is agnostic to both the modelling language and the target implementation language, and a proof-of-concept prototype has been developed on top of an existing template-based M2T language (the Epsilon Generation Language [124])[1]. EGL has been selected because it is a powerful and mature model-to-text transformation language.

[1]All source code for merging approach are available at `https://github.com/soha500/MergingApproach`

Chapter Structure: This chapter is organised as follows. Section 4.2 describes how M2T transformation languages can be extended with a merging approach. It also describes in detail how EGL language can be extended to support the merging approach. Section 4.3 outlines the evaluation process and results. Section 4.4 discusses the practicality and limitations of the proposed approach. Section 4.5 discusses alternative possible solutions and Section 4.6 concludes by summarizing this chapter.

4.2 Extending EGL with Merging Approach

To avoid the problems of using the protected regions' mechanism, the EGL language has been extended with additional features. These features aim to achieve the following:

- Enabling developers to add lines of hand-written text/code without the need to declare protected regions or separate them into other files.
- Detect conflicts between generated files, templates, and models.
- Warn the developers if they have modified or deleted any line that has been auto-generated.

The following section describes how the algorithm works.

4.2.1 Assumptions

This section provides a list of assumptions for using the Automated Line Based Merging Technique. Thus the user of this technique or other researchers looking to expand this work may benefit from having these clear assumptions.

Target Language Support Comment. One assumption, in order for this approach to be applicable, is that the target language needs to support character comments. For example, if the M2T is generating a JSON file, this approach would not be applicable because JSON does not support comments. Presently, Java-style comments (and any other language that uses a similar comment structure to Java, like HTML, Python, Ruby, etc.) are supported.

```
public class TemperatureController {
  public int execute(int temperature, int targetTemperature) {
        protected region execute on begin

    // protected region execute end
  }
}
/*
XbK6q3PAD1YNASA=ZXyhAZQ=fQ==
*/
```

Listing 4.1: An example of corrupted protected region.

Source Model Modification. Another assumption in using this approach is that after the first generation, the source model is not edited in any way. More details on this are provided in the limitations section (Section 4.4.1) of this Chapter.

Corrupted Protected Regions. Another important assumption is that the start or end line of the protected regions should not change as shown in Listing 4.1. The reason for this is that the proposed algorithm is based on counting the start line of protected regions to prevent the content of regions from being hashed (e.g. hold the content until all lines are hashed then return them back). Thus, if the start line has been changed the region content will be hashed and the algorithm will not work as expected.

Hash-Line Corruption. The hash line at the bottom of the generated files should not be changed or deleted at any time. This is also true in the case when the structure of preserving or generating the hashes in the algorithm is changed, for example from using four characters per line to five. This will make the previous hashes invalid.

Implementation of Merging Approach

An overview of the approach is given in Figure 4.1. Firstly, it is expected that developers create models (step ①) and templates (step ②) and then subsequently run the EGL transformation (step ③). This transformation will produce a set of files in the target language. A problem arises when there

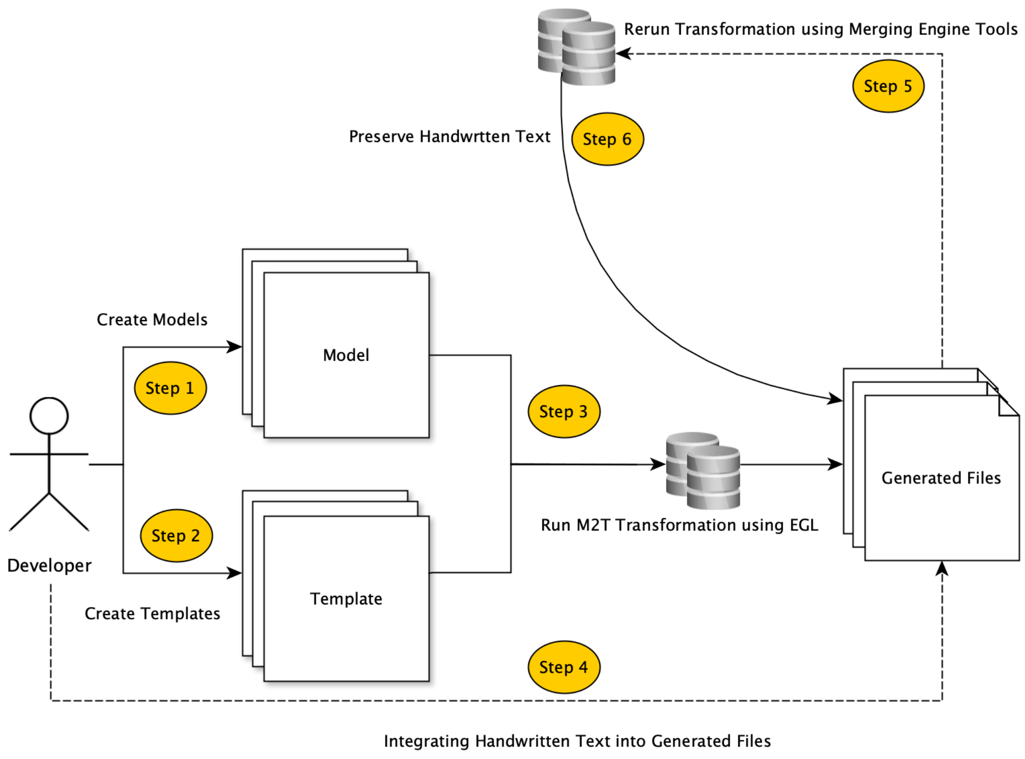

Rerun Transformation using Merging Engine Tools
Step 5
Preserve Handwrtten Text
Step 6
Create Models
Model
Step 1
Step 3
Run M2T Transformation using EGL
Generated Files
Step 2
Developer
Create Templates
Template
Step 4
Integrating Handwritten Text into Generated Files

Figure 4.1: An overview of the proposed approach for merging hand-written text into generated files using EGL.

are existing generated files that the user may have made modifications to (step ④). In its normal operation mode EGL will overwrite these potential modifications. The implementation of a merging approach enables EGL to detect these previously generated files (present on disk), and to use line-based merging tools to preserve any non-destructive changes. In these cases, the merging engine is invoked (step ⑤) with both the user-modified file and the newly generated model file. This process preserves hand-written text (step ⑥) as the final output of the transformation process.

4.2.2 Extending EGL with Merging Approach

The merging algorithm, which consists of 4 steps, is described below:

```
1  public class TemperatureController {
2
3      public int execute(int temperature, int targetTemperature) {
4
5
6      }
7  }
8  /*
9  XbK6AA==q3PACQ==ASA=AZQ=fQ==
10 */
```

Listing 4.2: The result of generating *TemperatureController* class from the template 4.3

Step A: Generate the new content of the target file using standard EGL, and generate the hash code

The standard EGL transformation produces components as files in the root directory. Just before a generated file is written, an extension developed in this project intercepts it and appends a comment at the end of the file, containing all the hashes of each line of the file. There are four-character base-64 hash codes of each line, concatenated, as shown on line 9 of Listing 4.2. Before being hashed protected regions must be entirely extracted so they are not hashed, then inserted back into the file after hashing (which will be the same lines as when removed). This is because protected regions are not produced by model transformation and can only contain hand-written text. Each generated line, including white space, is hashed and each hash is truncated into four characters. The user is cautioned against editing the hash line at the bottom of the file to preserve the algorithm functionality. It is possible to change the length of the recorded hash; for example instead of four it could be eight or sixteen characters. However, the probability of clashing (i.e. a line changing while its hash remains the same) with four characters is acceptably low (as described in Section 4.4.2).

Step B: Detecting Changes to Original Content and Extracting the Original Lines

If the developers change or delete any line that is auto-generated, then the merging algorithm will stop and produce an exception. When the original document was created, the hash code was appended to the end of the file. When a new transformation is performed each existing file, which the user may have modified, is checked for integrity. The file would be considered corrupt if the user has deleted or modified any of the original auto-generated lines. To detect this each line is hashed again and checked against the hash recorded at the bottom of the existing file. Each line hash must be present and in the same order.

Figure 4.2 illustrates the hashes of each line of an auto-generated file. The hashes would be appended to the bottom of the file normally as shown in Listing 4.2, but here they are shown on the right-hand margin.

```
1 public class TemperatureController {                                   XbK6
2                                                                        AA==
3   public int execute(int temperature, int targetTemperature) {         q3PA
4                                                                        CQ==
5                                                                        ASA=
6   }                                                                    AZQ=
7 }                                                                      fQ==
```

Figure 4.2: An example of hashes for each generated line in an original file.

Figure 4.3 illustrates a user-modified file after an integrity check, where each line is re-hashed and compared in-turn to the next expected hash. In this case, line 5 is a new line so it doesn't match hash "ASA=", but the next line does. By the end of the process all hashes are accounted for, and it is assured that all the original lines are still present (in the same order).

```
1 public class TemperatureController {                                   XbK6
2                                                                        AA==
3   public int execute(int temperature, int targetTemperature) {         q3PA
4                                                                        CQ==
5        return temperature - targetTemperature;                         Q/oD
6                                                                        ASA=
7   }                                                                    AZQ=
8 }                                                                      fQ==
```

Figure 4.3: An example of the algorithm checking all hashes are present when one line has been added.

Figure 4.4 illustrates a file where one of the original lines (line 5 in Figure 4.2) with hash "ASA=" has been deleted. The next hash it is trying to find beyond that line is "ASA=", which is not present anywhere in the document. The algorithm does not find all the hashes, and therefore reports a corrupt document. In a well-formed document, each line that can be matched with a hash (original lines) is concatenated together to reproduce the original auto generated file.

```
public class TemperatureController {                           XbK6
                                                               AA==
  public int execute(int temperature, int targetTemperature) { q3PA
                                                               CQ==
  }                                                            AZQ=
}                                                              fQ==
```

Figure 4.4: An example of a corrupt file and how the algorithm determines there are missing lines.

Figure 4.5 illustrates a file where one of the original lines (line 3 in Figure 4.2) with hash "q3PA" has been modified (e.g., the modifier for *execute()* changed from *public* to *private*). The next hash it is trying to find beyond that line is "q3PA", which is not present anywhere in the document. The algorithm does not find all the hashes, and therefore reports a corrupt document.

```
public class TemperatureController {                           XbK6
                                                               AA==
  private int execute(int temperature, int targetTemperature) {BekZ
                                                               CQ==
                                                               ASA=
  }                                                            AZQ=
}                                                              fQ==
```

Figure 4.5: An example of a corrupt file and how the algorithm determines there are modifying lines.

Step C: Merging the New Auto-generated Lines and the User Modifications

At this stage of the algorithm, it is assured that the original lines are all present. This can then be used as the history of the document to resolve conflicts. The merging strategy used is a three-way merge, similar to the Source Control Management Software Git. Three-way merging was found to be more suitable than just a two-way merge of the new auto-generated file and the user modifications. Git requires a three-way merge strategy to resolve conflicts, as when tasked with merging changes between two branches it needs to account for the file's history. The reason it is more suitable is that comparing two files is not enough to detect the changes that were made on different branches. For example, let's assume that two people made changes to the same file; one added a new line at the beginning of the file and the other one removed the last line of the file. Then both committed their changes to be merged. If they were doing a two-way merge (e.g. diff), the tool could compare the two files, and see that the first and last lines of the file are different. However, it is not possible to know how to resolve the differences, i.e. whether the merged version includes the first line and whether it includes the last line.

Alternatively, with a three-way merge, two files are compared, but against the original copy, before either of them were changed. So it is evident that first line was added, and last line was removed. Therefore, the information can be used to produce a correct and fully merged version that respects the two different changes that were made by the different developers. The differences between two-way and three-way were discussed previously in Section 2.3.3 along with an example. The algorithm that was implemented used a component of an existing library to accomplish this three-way merging, called JGit. It is invoked with the new auto-generated file, the existing file with user modifications, and the original file as extracted from the existing file.

Step D: Resolving Conflicts

When a generated file is modified, the user is permitted to create new lines of code, but is forbidden from modifying or deleting any original lines. However, it cannot entirely be avoided that the model is updated and the EGL process produces a file that causes conflicts with user-modified code. Three-way merging can avoid this problem in the case of a modified template file, where its original transformation lines can be safely identified and replaced. However, if the auto-generated lines come into conflict with user-modified code then the process must communicate to the user that these conflicts must be resolved before the process is resumed. This implementation works by creating a sibling file with a .conflict extension containing the conflicts, and printing a warning to the user if any of these files have been generated. Since the automated process could not resolve the conflicts, the user is then expected to evaluate them and resolve them by hand.

```
public class [%=c.name%] {

    public [%=c.outPort.type%] execute([%=c.inPorts.collect(p|p.type + "
        " + p.name).concat(", ")%]) {

    }
}
```

Listing 4.3: An example of EGL template for generating *TemperatureController* class

```
public class TemperatureController {

    public int execute(int temperature, int targetTemperature) {

    }
}
/*
XbK6AA==q3PACQ==AZQ=fQ==
*/
```

Listing 4.4: The result of generating *TemperatureController* class from the template 4.3

To illustrate this step, all possible scenarios for adding one or more new lines to the template, the generated files, or to both, are presented. Three of these scenarios demonstrate automatic resolution, with one requiring manual resolution. An example of an original template is presented in Listing 4.3 and the result of executing the transformation is shown in Listing 4.4. These are the first versions of the files before the modifications in later listings.

Scenario 1: Adding a new line to the template. An example of the original template is presented in Listing 4.3 and the result of running the transformation is shown in Listing 4.4. Now, if user adds a new line to the original template (as shown in Listing 4.5 (line 5)) and reruns the transformation, the addition can be merged normally as an automatic resolution, as shown in Listing 4.6 (line 5).

```
public class [%=c.name%] {

    public [%=c.outPort.type%] execute([%=c.inPorts.collect(p|p.type + " " + p.name).concat(", ")%]) {

        System.out.println("Running [%=c.name%].execute()");
    }
}
```

Listing 4.5: An example of adding a new line into the template 4.3 for *TemperatureController* class

```
public class TemperatureController {

   public int execute(int temperature, int targetTemperature) {

       System.out.println("Running TemperatureController.execute()");
    }
}
/*
XbK6AA==q3PACQ==/AP7AZQ=fQ==
*/
```

Listing 4.6: The result of adding and merging one line into the *TemperatureController* class

Scenario 2: adding a new line to the generated file. An example of the original template is presented in Listing 4.3 and the result of running the transformation is shown in Listing 4.4. Now, if user adds a new line to the generated file (as shown in Listing 4.7 (line 5)) and reruns the transformation, the addition can be merged normally as an automatic resolution, as shown in Listing 4.8 (line 5).

Scenario 3: adding a new line to both template/model and the generated file but in clearly distinctive places. An example of the original template is presented in Listing 4.3 and the result of running the transformation is shown in Listing 4.4. Now, if the user adds a new line to the template (as shown in Listing 4.9 (line 5)) and also adds a new line to generated file (as shown in Listing 4.10 (line 4)) and then reruns the trans-

```java
public class TemperatureController {

    public int execute(int temperature, int targetTemperature) {

      return temperature - targetTemperature;
    }
}
/*
XbK6CQ==q3PAAA==MPU=fQ==
*/
```

Listing 4.7: An example of adding a new line into the generated file 4.4 for *TemperatureController* class

```java
public class TemperatureController {

    public int execute(int temperature, int targetTemperature) {

        return temperature - targetTemperature;
    }
}
/*
XbK6CQ==q3PAAA==MPU=fQ==
*/
```

Listing 4.8: The result of adding a new line into *TemperatureController* class

formation, the addition can be merged normally as an automatic resolution, as shown in Listing 4.11 (lines 4 and 6).

Scenario 4: adding a new line to both template/model and the generated file in the same place. An example of the original template is presented in Listing 4.3 and the result of running the transformation is shown in Listing 4.4. Listing 4.12 gives an example of a new line being added to a template file (line 5), which is auto-generated. Listing 4.13, as part of the same example, shows a new line that has been added by the user, problematically in the exact same place (line 5). In this case the algorithm is unable to automatically resolve these files and generates a conflict file as shown in Listing 4.14. The next time the algorithm is executed it detects

```
public class [%=c.name%] {

    public [%=c.outPort.type%] execute([%=c.inPorts.collect(p|p.type + "
         " + p.name).concat(", ")%]) {

            System.out.println("Running [%=c.name%].execute()");
        }
}
```

Listing 4.9: An example of adding a new line into the template 4.3 for *TemperatureController* class

```
public class TemperatureController {

    public int execute(int temperature, int targetTemperature) {
        return temperature − targetTemperature;

    }
}
/*
XbK6AA==q3PACQ==AA==AZQ=fQ==
*/
```

Listing 4.10: An example of adding a new line into the generated file 4.4 for *TemperatureController* class, but in different place from the one that added to the template

these .conflict files and expects that the user has made the changes they intended to make. It replaces the hashes at the bottom of the file with the newly merged line hashes so that the next time the algorithm is executed it accepts the previous changes.

4.2.3 Algorithms

This section presents how the proposed approach works, how hashes are generated in the EGL, how original lines are extracted from new lines, and how addition lines can be merged.

```
public class TemperatureController {

   public int execute(int temperature, int targetTemperature) {
       return temperature - targetTemperature;

       System.out.println("Running TemperatureController.execute()");
   }
}
/*
XbK6AA==q3PACQ==AA==AZQ=fQ==
*/
```

Listing 4.11: The result of merging both lines in *TemperatureController* class after the transformation rerun

```
public class [%=c.name%] {

    public [%=c.outPort.type%] execute([%=c.inPorts.collect(p|p.type + "
        " + p.name).concat(", ")%]) {

        System.out.println("Running TemperatureController.execute()");
    }
}
```

Listing 4.12: An example of adding a new line into the template 4.3 for *TemperatureController* class

Algorithm for All Merging Steps

Algorithm 1 describes how the proposed merging approach works. The algorithm iterates through all the generated files in the transformation's output root folder (lines 3-28). Line 1 creates a list of all the files in the working directory and line 2 creates an empty list for messages. For each file, the algorithm first checks for any deletions or modifications of the original lines of code (lines 4-7) and if any original lines are missing it records an error and skips the file. If all the original lines are present, it proceeds to extract protected regions from the file content, leaving only the region start, which can be later merged without conflicts, collecting the region bodies into a queue (lines 8-14). After extraction of the regions, a copy of the file's original lines is created, where this and the new content produced from the transformation

```
public class TemperatureController {

   public int execute(int temperature, int targetTemperature) {

         return temperature - targetTemperature;

   }
}
/*
XbK6AA==q3PACQ==CQ==AZQ=fQ==
*/
 conflicted
```

Listing 4.13: An example of adding a new line into the generated file 4.4 for *TemperatureController* class, but with different value from the one that added to the corresponding line in the template.

```
public class TemperatureController {

   public int execute(int temperature, int targetTemperature) {

<<<<<<< O
        System.out.println("Running TemperatureController.execute()");
=======
       return temperature - targetTemperature;

>>>>>>> T
   }
}
```

Listing 4.14: The result of detecting conflicts between the template 4.12 and the generated file 4.13

and the file as a whole, are merged together by a three-way merge algorithm (lines 15-17).

Then the algorithm checks for any conflicts between these three bodies of text (lines 18-21). This is done by checking if any lines contain textual artefacts inserted by the merging algorithm to signify to developers, IDEs, and VCSs, that there are conflicts that require manual intervention to resolve. It creates a message with the name of the file that contains the conflicts, and adds a *conflicted* message at the end of the filename prompting the developer to fix all conflicts in order to continue. The algorithm then iterates through the resulting merged lines, and at each instance of a region start it reinserts the region bodies directly underneath from the *ListOfAllRegions* queue, on a first-in-first-out basis (lines 22-26). At the end of processing each file, the merged lines are joined into the document and written to disk, replacing the potentially user-modified file (line 27). Finally, the algorithm prints any messages it accumulated for all files to the console (lines 29-31).

Algorithm for Generating Hashes.

Algorithm 2 is used to generate hashes for the content of each line, invoked for each file in the working directory. It iterates through all lines in the file and hashes them using Java's native object hashes, encoded as base-64 (lines 4-7). Then it concatenates all the hashes together (line 8). This forms a line of hashes that contain comments using a detected comment style (lines 9-10), detected with Algorithm 3. Finally it appends this comment at the end of the file, with a buffer of two new lines between it and the file content (line 11-12).

Algorithm for Detecting Comment Style

Algorithm 3 illustrates how the merging algorithm detects the comment style for target languages. This function first splits the file by newlines (line 2), then extracts the first line (line 3). This first line is compared (lines 6, 10, 14, 18) to several potential language tags, which are used to simply inform the merging algorithm which language is in use. If no match is found then default C-style (also used in Java) comments are used (lines 4-5). This function itself does not exist in the implementation of the merging algorithm but does illustrate the logic around matching language tags. This function,

Algorithm 1 How merging engine works

```
 1: $ListOfFiles \leftarrow$ all the files in the folder
 2: $ListOfMessages \leftarrow \emptyset$
 3: for all $f \in ListOfFiles$ do
        Step 1: Check if original lines were deleted/modified
 4:     if not AllOriginalLinesPresent($f$) then
 5:         Append to $ListOfMessages$ "at least one of original lines was deleted or modified in $f$"
 6:         continue
 7:     end if
        Step 2: Extract any protected regions, leaving a one-line region start in their places
 8:     $ListOfAllRegions \leftarrow \emptyset$
 9:     for all $line \in f$ do
10:         if IsRegionStart($line$) then
11:             Add region starting on $line$ to $ListOfAllRegions$
12:             Remove these lines from $f$, with just $line$ in its place
13:         end if
14:     end for
        Step 3: Merge original lines with developer modifications and new transformation
15:     $OriginalLines \leftarrow$ ExtractOriginalLines($f$)
16:     $NewLines \leftarrow$ GetNewModelTransformation($f$)
17:     $MergedLines \leftarrow$ ThreeWayMerge($OrginalLines$, $NewLines$, $f$)
        Step 4: Check if there were any conflicts
18:     if $f$ contains conflict tag then
19:         Append to $ListOfMessages$ "There were conflicts in merging the contents for $f$"
20:         Append to $f$ "conflicted"
21:     end if
        Step 5: Return any protected regions back into their places
22:     for all $line \in MergedLines$ do
23:         if IsRegionStart($line$) then
24:             Return the next region from $ListOfAllRegions$ in place of $line$
25:         end if
26:     end for
        Step 6: Write the new document as the result of the transformation and merge back to the file
27:     write $f$ JoinByLines($MergedLines$)
28: end for
    Step 7: print any messages to the console
29: for all $message \in ListOfMessages$ do
30:     print($message$)
31: end for
```

Algorithm 2 How EGL generates hashes

```
1:  function HashFile($f$)
2:      $ListOfLines \leftarrow$ SplitByLines($f$)
3:      $ListOfHashes \leftarrow \emptyset$
4:      for all $line \in ListOfLines$ do
5:          $HashedLine \leftarrow$ Base64Hash($line$)
6:          Append to $ListOfHashes$ Truncate($HashedLine$, 4 characters)
7:      end for
8:      $HashLine \leftarrow$ JoinByLines($ListOfHashes$)
9:      $CommentStyle \leftarrow$ DetectCommentStyle($f$)
10:     $CommentedHashLine \leftarrow$ Decorate($HashLine$, $CommentStyle$)
11:     Append two newlines to $f$
12:     Append $CommentedHashLine$ to $f$
13: end function
```

for illustrative purposes, returns a tuple of the comment start and end (line 22), which are local variables in the implementation.

Algorithm for Checking All Original Lines are Present

Algorithm 4 illustrates how the merging algorithm checks that all original lines from a previous transformation are still present in a potentially user-modified file. It checks whether all the hashes for the original lines exist sequentially. Regions being included in the input file would not necessarily be a problem for this function, but to avoid potential problems in the context of merging they are removed for the purpose of this check (line 2). To parse the file for its original hashes it is split by newlines (line 3); the last four lines are removed (line 4); and the 3rd of these lines, the line with the hashes concatenated together, is extracted (line 5) and subsequently split by groups of four characters (line 6). It iterates through all lines in the files, checking that each hash is present one after the other (Lines 8-17). It does this by hashing each line of the file and comparing it to the next expected hash in the sequence (lines 9-11). It can determine if any original lines are missing by looking at the difference between the number of hashes found and the number of original hashes (line 18). If all the hashes have been found early in the iteration of all lines it stops the iteration early (line 14). For example,

Algorithm 3 How merging algorithm detects comment style for generated files

```
function DetectCommentStyle($f$)
    $ListOfLines \leftarrow$ SplitByLines($f$)
    $FirstLine \leftarrow$ First($ListOfLines$)
    $CommentStart \leftarrow$ "/*"
    $CommentEnd \leftarrow$ "*/"
    if $FirstLine$ equals "<!–HTML–>" then
        $CommentStart \leftarrow$ "<!–"
        $CommentEnd \leftarrow$ "–>"
    end if
    if $FirstLine$ equals "#Python" then
        $CommentStart \leftarrow$ "”’"
        $CommentEnd \leftarrow$ "”’"
    end if
    if $FirstLine$ equals "#Ruby" then
        $CommentStart \leftarrow$ "=begin"
        $CommentEnd \leftarrow$ "=end"
    end if
    if $FirstLine$ equals "–Haskell" then
        $CommentStart \leftarrow$ "{-"
        $CommentEnd \leftarrow$ "-}"
    end if
    return ($CommentStart$, $CommentEnd$)
end function
```

if the file is 100 lines long, and it finds the first ten hashes in the first twenty lines, then it stops looking after twenty lines.

Algorithm for extracting original lines

Algorithm 5 illustrates how the merging algorithm extracts the original lines (the ones that were hashed after a previous transformation) from a file. Similar to Algorithm 4, this function iterates through all lines (without regions) in the file, although it also collects original lines found (line 13). If there are too few original lines present, though in the case of the merging algorithm, this is never the case as a corrupt document would terminate the algorithm

Algorithm 4 How merging algorithm checks all original lines are present

```
1: function AllOriginalLinesPresent($f$)
2:     $WithoutRegions \leftarrow$ RemoveRegions($f$)
3:     $ListOfLines \leftarrow$ SplitByLines($WithoutRegions$)
4:     $EndOfFileLines \leftarrow$ RemoveLast($ListOfLines$, 4)
5:     $HashLine \leftarrow$ Nth($EndOfFileLines$, 3)
6:     $ListOfHashes \leftarrow$ Split($HashLine$, 4 characters)
7:     $h \leftarrow 1$
8:     for all $line \in ListOfLines$ do
9:         $HashedLine \leftarrow$ HashLine($line$)
10:        $OriginalHash \leftarrow$ Nth($ListOfHashes$, $h$)
11:        if $HashedLine$ equals $OriginalHash$ then
12:            $h \leftarrow h + 1$
13:            if $h >$ Length($ListOfHashes$) then
14:                break
15:            end if
16:        end if
17:    end for
18:    return $h$ equals Length($ListOfHashes$)
19: end function
```

before merging, it would return as many original lines as it could sequentially find. In the end, it returns the list of original lines it extracted (line 20).

4.3 Evaluation

This section outlines the results of the evaluation of the correctness and scalability of the proposed approach[2]. It also discusses threats to the validity of the results and the algorithm itself.

4.3.1 Correctness

Several unit tests have been developed to build confidence on the correctness of the prototype implementation of the approach. This has been done using

[2]All unit tests, input models, generated files, raw and analysed results for all the experiments presented in this section are available at `https://github.com/soha500/MergingApproach`

Algorithm 5 How merging algorithm extra original lines

```
function ExtractOriginalLines($f$)
    $WithoutRegions \leftarrow$ RemoveRegions($f$)
    $ListOfLines \leftarrow$ SplitByLines($WithoutRegions$)
    $EndOfFileLines \leftarrow$ RemoveLast($ListOfLines$, 4)
    $HashLine \leftarrow$ Nth($EndOfFileLines$, 3)
    $ListOfHashes \leftarrow$ Split($HashLine$, 4 characters)
    $ListOfOriginalLines \leftarrow \emptyset$
    $h \leftarrow 1$
    for all $line \in ListOfLines$ do
        $HashedLine \leftarrow$ HashLine($line$)
        $OriginalHash \leftarrow$ Nth($ListOfHashes$, $h$)
        if $HashedLine$ equals $OriginalHash$ then
            Append $Line$ to $ListOfOriginalLines$
            $h \leftarrow h + 1$
            if $h >$ Length($ListOfHashes$) then
                break
            end if
        end if
    end for
    return $ListOfOriginalLines$
end function
```

the JUnit library to ensure that the presented algorithm behaves as expected in the following scenarios:

Scenario 1: Preserve Integrated Handwritten Text. In this scenario, several tests were performed to check whether the proposed technique is able to preserve integrated handwritten text within the auto-generated text. These tests were as described in the following:

- **Test 1: Additional Line.** When one line has been added into the auto-generated lines and preserved without using protected region markers as shown in Table 4.1. This test would fail if the algorithm found that the original had been modified, that there was a conflict, or it did not manage to preserve the additional line. The expected outputs are status codes from the algorithm that must be given in these scenarios.

Test I: Adding New Lines to Original Lines	

Original Content of Generated File	public class TemperatureController {
	    public int execute(int temperature, int targetTemperature) {
	
	    }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Adding New Lines to Generated Lines	public class TemperatureController {
	    public int execute(int temperature, int targetTemperature) {
	        return temperature - targetTemperature;
	    }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Expected Output	MergedSuccessfully
Result of The Test	Pass

Table 4.1: Inputs and expected outputs of adding a new line to the original lines.

- **Test 2: Additional Lines.** When multiple lines have been added into the auto-generated lines and preserved without using protected region markers as shown in Table 4.2. This test would fail if the algorithm found that the original had been modified, that there was a conflict, or it did not manage to preserve the additional line. The expected outputs are status codes from the algorithm that must be given in these scenarios.

- **Test 3: Additional Protected Region Lines.** When the size of the protected regions can be extended without error as shown in Table 4.3. The algorithm should not perform any merging or report any errors. The file should remain exactly the same outside of the protected/*sync* regions.

Scenario 2: Detect any Modification or Deletion to Auto-generated Text. In this scenario, several tests were performed to check whether the proposed technique is able to detect if any of the auto-generated text has been modified or deleted. Possible corner cases in this scenario were covered as described in the following:

Test I: Adding New Lines to Original Lines	

Original Content of Generated File	public class TemperatureController {
	    public int execute(int temperature, int targetTemperature) {
	
	    }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Adding New Lines to Generated Lines	public class TemperatureController {
	    public int execute(int temperature, int targetTemperature) {
	        //Add body of the method here
	        return temperature - targetTemperature;
	    }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Expected Output	MergedSuccessfully
Result of The Test	Pass

Table 4.2: Inputs and expected outputs of adding new lines to the original lines.

- **Test 1: Modification Lines.** One test where the algorithm detects that original lines have been lost, when at least one of the auto-generated lines has been modified as shown in Table 4.4. The algorithm should not attempt to do any merging nor report any conflicts.

- **Test 2: Deletion Lines.** One test where the algorithm detects that original lines have been lost, when at least one of the auto-generated lines has been deleted as shown in Table 4.5. The algorithm should not attempt to do any merging nor report any conflicts.

- **Test 3: Hash Line Modified.** When the hash line was modified as shown in Table 4.6, the original file is reported as corrupt. This is because the algorithm will be expecting hashes from the hash line that do not exist in the file. The algorithm should not attempt any merging, nor report any conflicts.

Scenario 3: Detect Conflicts between Auto-generated and Hand-written Text. In this scenario, several tests were performed to check whether the proposed technique is able to detect any conflicts between the

Test IV: Adding New Lines Inside Protected Regions	

Original Content of Generated File	public class TemperatureController {
	    public int execute(int temperature, int targetTemperature) {
	    // protected region execute on begin
	
	    // protected region execute end
	    }
	}
	/*
	XbK6q3PAD1YNASA=ZXyhAZQ=fQ==
	*/

Adding New Lines Inside Protected Regions	public class TemperatureController {
	    public int execute(int temperature, int targetTemperature) {
	    // protected region execute on begin
	        return temperature - targetTemperature;
	    // protected region execute end
	    }
	}
	/*
	XbK6q3PAD1YNASA=ZXyhAZQ=fQ==
	*/

Expected Output	MergedSuccessfully
Result of The Test	Pass

Table 4.3: Inputs and expected outputs of adding new lines inside protected regions.

Test II: Modifying Original Lines	

Original Content of Generated File	public class TemperatureController {
	    public int execute(int temperature, int targetTemperature) {
	
	    }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Modifying Generated Lines	public class Boiler {
	    public int execute(int temperature, int targetTemperature) {
	
	    }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Expected Output	OriginalWasModified
Result of The Test	Pass

Table 4.4: Inputs and expected outputs of modifying original lines.

Test III: Deleting Original Lines	

Original Content of Generated File	public class TemperatureController {
	        public int execute(int temperature, int targetTemperature) {
	
	        }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Deleting Generated Lines	
	        public int execute(int temperature, int targetTemperature) {
	
	        }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Expected Output	OriginalWasModified
Result of The Test	Pass

Table 4.5: Inputs and expected outputs of deleting original lines.

Test VII: Modifying Hash Line	

Original Content of Generated File	public class TemperatureController {
	        public int execute(int temperature, int targetTemperature) {
	
	        }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Modifying Hash Line	
	        public int execute(int temperature, int targetTemperature) {
	
	        }
	}
	/*
	NK==XbK6q3PACQ==AZQ=fQ==
	*/

Expected Output	OriginalWasModified
Result of The Test	Pass

Table 4.6: Inputs and expected outputs of modifying hash line.

Test V: Adding Corresponding Lines	

Original Content of Generated File	public class TemperatureController {
	        public int execute(int temperature, int targetTemperature) {
	
	        }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Adding New Line to Generated Lines	public class TemperatureController {
	        public int execute(int temperature, int targetTemperature) {
	                return temperature - targetTemperature;
	        }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Adding New Line to EGL Template	public class [%=c.name%] {
	        public [%=c.outPort.type%] execute([%=c.inPorts.collect(p|p.type + " " +
	                p.name).concat(", ")%]) {
	                return temperature - targetTemperature;
	        }
	}

Expected Output	MergedSuccessfully
Result of The Test	Pass

Table 4.7: Inputs and expected outputs of adding new similar lines in the same position of the template and its generated file.

files on disk and the files that are about to be generated. These tests were as described in the following:

- **Test 1: Corresponding Lines.** When in the same position of the template and its generated file two exactly equal values have been added after the first transformation and there are no conflicts in this case as shown in Table 4.7. The algorithm should not report conflicts, nor lose the added line, and keep only one copy of the line.

- **Test 2: Conflicting Lines.** When in the same position of the template and its generated file two different values have been added manually after the first transformation and there is a conflict as shown in Table 4.8. The algorithm should not keep both lines and should not report a conflict. The algorithm should be able to provide a merged text with conflicts labelled for a potential user to correct.

The algorithm can act with all the above scenarios and continues if only new lines are added. However, with all other scenarios the developer must

Test VI: Adding Conflicting Lines	

Original Content of Generated File	public class TemperatureController {
	        public int execute(int temperature, int targetTemperature) {
	
	        }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Adding New Line to Generated File	public class TemperatureController {
	        public int execute(int temperature, int targetTemperature) {
	                return temperature - targetTemperature;
	        }
	}
	/*
	XbK6q3PACQ==AZQ=fQ==
	*/

Original Content of EGL Template	public class [%=c.name%] {
	        public [%=c.outPort.type%] execute([%=c.inPorts.collect(p|p.type + " " +
	                p.name).concat(", ")%]) {
	                return temperature + targetTemperature;
	        }
	}

Expected Output	ConflictsFound
Result of The Test	Pass

Table 4.8: Inputs and expected outputs of adding new different lines in the same position of the template and its generated file.

be involved to fix the issue. None of the tests are for anything outside of the algorithm, such as reading and writing files or appending hash lines.

4.3.2 Performance

This algorithm is meant to be applied to potentially hundreds of files, each with potentially hundreds of lines of code, which means that performance of the algorithm is an important factor. To measure its performance tests have been conducted which stress different features:

Additional Lines Test. Adding a number of random strings in random lines in the generated file. This will stress that part of the algorithm that does merging, but should not cause any merging conflicts nor line modifications.

Cloned Lines Test. Copying multiple existing lines of the generated file and duplicating them onto random lines. This will stress that part of the algorithm that is able to distinguish between the generated and manually added lines, even if they are of the same value.

```
long start = System.currentTimeMillis();

MergingAndConflicts results = MergingAndConflicts.DoMergingAndConflicts
      (modifiedContents, transformationContents);
writeResultsToCsvFile(System.currentTimeMillis() − start, ...) ;
```

Listing 4.15: An example of calculating the time taking

Deletion Line Test. Deleting one line in the file and detecting this as part of the integrity check.

Conflicting Line Test. Adding one random string to a random position in the generated file, while at the same time adding a random string in the template in the same position, but with a different value. This will stress that part of the algorithm that attempts, but fails to, resolve conflicts.

Conflicting Lines Test. Adding two random strings to two random positions in the generated file, as well as adding two random strings in the template in the same places but with different values. This will further stress the same part of the algorithm as the test above.

Each test is executed 100 times in order to allow the Java Virtual Machine (JVM) that the software runs on to "warm up", and to ensure a stable average between tests. The JVM may perform operations periodically, which impedes the performance of the tests, such as Garbage Collection (GC), so this is important.

The EGL generation, and file reading and writing, are not considered in the performance tests; the tests are focused entirely on the merging and conflicts algorithm. This includes the file integrity check, extracting original lines for three-way merging, the merging itself, reporting any unresolved conflicts, and any automatic conflict resolution. See Listing 4.15 as an example of only the algorithm being tested.

To assess the performance of the proposed algorithm (based on the size of the model), the M2T transformation was executed (as discussed in Section 4.2) on files of various sizes: containing 500 lines, then incrementing by 500 lines to 10,000 lines. To measure performance, the algorithm was timed in completing its task, repeated one hundred times for each increment of the number of lines in a file.

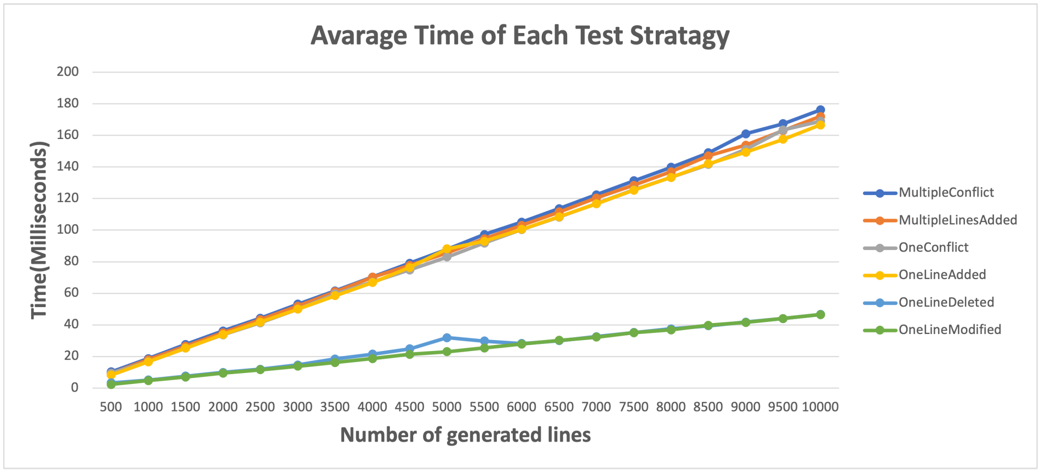


Figure 4.6: Result of measuring the average time when M2T transformation run 100 times.

	MS at 10K Lines	Comparison
MultipleConflict	176.19	3.7
MultipleLinesAdded	172.03	3.7
OneConflict	168.99	3.6
OneLineAdded	166.7	3.5
OneLineDeleted	46.58	1.0
OneLineModified	46.46	1.0

Table 4.9: Result of the comparison.

Figure 4.6 illustrates various performance tests of the algorithm. Each data-point is an average to the nearest 500 lines for comparison (as the tests generate a variable number of lines each time), and plotted against one another. Each test strategy used the same source text but modified it in different ways. The algorithm took a similar time when one line was deleted or modified. However, in case of any additions or conflicts the time increased 350% to 370% compared to other tests.

The results of performance testing are encouraging, as the execution time of the proposed algorithm increases linearly as the size of the generated files (lines) increases, as shown in Figure 4.6.

When multiple lines were added, or were in conflict, the algorithm took almost quadruple the time compared to when one line was added or modified. The test used a maximum of ten thousand lines in order to demonstrate wide correlations between stresses on the algorithm. In reality actual projects would have far fewer lines per file than this. Therefore, for actual projects, the algorithm would perform sufficiently well enough to run over hundreds of lines within a few seconds, not considering file read/write time.

The actual tests performed embody the different ways to stress the algorithm which were described earlier:

- *OneLineAdded*. When one string is added in random line in the generated file.
- *MultipleLinesAdded*. When 100 random strings are added to random positions in the generated lines.
- *OneLineDeleted*. When one auto-generated lines is deleted in the generated files.
- *OneLineModified*. When one auto-generated lines is modified in the generated files.
- *OneConflict*. When one random string is added to a random position in the generated lines, while at the same time adding a random string in the template in the same position but with a different value.
- *MultipleConflict*. When 100 random strings are added to random positions in the generated lines, as well as adding some random strings in the template in the same places but with a different value.

4.3.3 Threats to Validity

The correctness and performance tests used in the evaluation may be deficient in some ways to real-world uses of the proposed solution. The arbitrary decisions on the content, structure, and alterations in the templates and generated files may not accurately replicate organic circumstances the algorithm may be used in. The templates and model used could rather be of real-world projects, instead of an almost entirely minimal viable template and model. The number of files, and number of lines per file, and number and shape of alterations to each file, might not be representative of real-world projects either, both in excess and insufficiency. The simulated edits to the templates and generated files may not be representative of the edits typically made by developers, such as being too randomised compared to operationally useful modifications. The complexity of the model would quite possibly always fall short of real-world models, as the components and connections between components were the principal variable.

```
public class TemperatureController {
   public int execute(int temperature, int targetTemperature) {
            protected region execute on begin

      // protected region execute end
   }
}
/*
XbK6q3PAD1YNASA=ZXyhAZQ=fQ==
*/
```

Listing 4.16: An example of corrupted protected region

Threats to Operation and Suitability

In this section, various threats to the operation of the algorithm by developers are described, together with their impression of the suitability of algorithm for their workflow.

Operation: Supported Comment Formats. The proposed approach is not applicable if the target language does not support comments (e.g. JSON).

Operation: Corrupted Protected Regions. Another threat is when the developer manually, and mistakenly, changes the start or end line of the protected regions as shown in Listing 4.16. The reason for this is that the proposed algorithm is based on counting the start line of protected regions to prevent the content of regions from being hashed (e.g. hold the content until all lines are hashed then return them back). Thus, if the start line has been changed the region content will be hashed and the algorithm will not work as expected.

Operation: Hash-Line Corruption. Any modification to the hash line at the bottom of the generated files raises an exception in the algorithm. This is also true in the case when the structure of preserving or generating the hashes in the algorithm is changed, for example from using four characters per line to five. This will make the previous hashes invalid. Another uncontrollable issue is any unforeseen errors or undefined behaviour within

EGL, which the algorithm has not been designed to handle, requiring the developers of EGL to fix or define the new behaviour.

The next two threats refer to the implementation rather than to the results.

Suitability: EGL Lock-in. The way the algorithm is implemented is directly within EGL's source code, rather than a separate component. This was done to give intimate access to the behaviour and features of EGL transformations e.g. intercepting the file-writing mechanism to add a hash-line. However, this means that any updates that the EGL authors publish would be very time-consuming, and the work of re-integrating the algorithm would be prone to error. For example, whole files such as `OutputBuffer.java`.java could be completely rewritten (refactored) or deleted, which is where very important code for the algorithm resides. A similar problem is that the algorithm can only be used for EGL transformation within Eclipse, and is unable to support any other Integrated Development Environment (IDE) such as Visual Studio, model transformation software, nor any other workflow.

Suitability: Algorithm Distribution. As mentioned above, the algorithm is integrated directly into the source code of Epsilon. This means in order to redistribute the software, a user would need to install this particular, "patched", version of Epsilon. This would require some effort for anyone wishing to use the algorithm: to create an installer that can make the necessary changes to a standard Epsilon installation (of the correct version); or the user to make changes manually to Epsilon's source code.

4.4 Discussion

In the previous sections, it was explained how M2T languages can be extended with the merge technique (Automated Line Based Merging) to simplify the process of integrating hand-written text within generated artefacts. This section elaborates on the differences between the automated line-based merging technique and the existing merging approaches of Protected regions and JMerge. Integrating hand-written text using Protected regions and JMerge require the developers to anticipate in advance which part of the code they want to extend.

Whereas, using an automated line-based merging technique does not require that procedure and the developers have the freedom to integrate their hand-written text in any part of the generated artefacts. Furthermore, in the automated line-based merging technique, conflicts between source models and the generated artefacts are detected by the M2T transformation engine, while in the protected regions approach, changes are only preserved so they are not overwritten during the re-run transformations.

4.4.1 Limitations

This section presents the limitations of the proposed approach.

Change Permanence. The proposed approach does not provide any tools or options to undo any operations it performs. Unlike Git, which provides a history, a user can navigate to find the past code they want, this algorithm makes changes directly to the files and deletes older versions. The reason for this is that the hashes are changeable each time the transformation is run; the original hashes are not preserved. A solution to this would be to keep copies or differentials of every version the algorithm has generated and somehow allow the user to navigate this.

Original Line Preservation. Another limitation is that the proposed approach does not allow the user/developer to change or delete generated lines. This is problematic as it becomes impossible to replace or override existing code, only to add new methods and fields, or new lines within the existing code base.

Inability to detect where lines were deleted or modified. In the case where the developer has removed or modified original lines, the algorithm is unable to determine which lines underwent these changes, since the user is also able to add new lines. This is because the algorithm is not intelligent enough to search forward through the hashes to check whether there are later matches. A more sophisticated version of the algorithm is left as future work.

4.4.2 Probability of Hash Collisions

An overview of the probabilities of hash collisions is described in detail in Table 4.10. With four characters of base-64 hash there are 64^4 unique hashes, which is a 1 in 16 777 216 probability of error; eight characters would be 64^8 which is 2.814e+14 unique hashes, which is excessive. The impact of the probability of error is reduced further by line hash collisions only occurring between sections of hand-written text and their next original line - any two dissimilar (therefore problematic) hand-written or original lines in a file could have the same hashes, but they would need to be uninterrupted by other original lines to cause the proposed solution problems. That is, only contiguous sections of hand-written text pose a threat, rather than the entire file. Any two dissimilar lines in the file causing hash collisions is at a probability of c for N number of lines, for H possible hashes. P is a function to calculate non-repeating permutations for r digits of base x [68].

$$P(x, r) = \frac{x!}{(x-r)!}$$

$$c = 1 - \frac{P(H, N)}{H^N}$$

The proposed solution has a lower probability of collision than might be the case for entire files, because c would typically be lower for n number of lines in a *section* as $N \geq n$. A table of the results of this equation can be seen in 4.10. In summary, having two-character hashes can lead to collisions (≈1% probability) even when the number of generated lines is small (ten lines), whereas with five-character hashes the collision possibility is negligible (one million times smaller chance for ten lines). Four-character hashes become unsuitable for uninterrupted regions of one thousand lines (≈3%), but five-character hashes are still quite feasible in this case (≈0.05% probability).

4.5 Alternatives

There are other alternative ways to record hashes as described below:

Hash Every Line. The hash could be preserved at the end of every generated line as shown in Listing 4.17. However, if this approach is used, the generated files will be excessively polluted. Also, it would be easy for a user to accidentally corrupt the hashes at the end of each line while editing.

Number of lines	Hash length	Chance
10	2 characters	1.0935%
10	3 characters	0.0172%
10	4 characters	0.000268%
10	5 characters	0.00000419%
100	2 characters	70.430%
100	3 characters	1.871%
100	4 characters	0.0295%
100	5 characters	0.000461%
1000	2 characters	100%
1000	3 characters	85.160%
1000	4 character	2.93%
1000	5 characters	0.0465%
1000	6 characters	0.000727%
1000	7 characters	0.000011%

Table 4.10: Probability of hash collisions.

```
public class BoilerActuator { /*rxeb*/
   public execute(int temperatureDifference, boolean boilerStatus) {/*
       WcQc*/
    /*Ghes*/
   } /*AHed*/
} /*Jwre*/
```

Listing 4.17: An example of generating hashes at the end of every generated line.

Hash at Beginning of File or Lines. Another alternative is to put the hashes at the beginning of the file as shown in Listing 4.18. However, this would get in the way of the user and increase the likelihood of corruption. Another option is putting each line's hash at the beginning of the line in a comment as shown in Listing 4.19, but this causes a similar pollution as mentioned previously.

Master Copies Directory. Another alternative is to maintain a master copy of all the generated files, such as in a sub-directory of the working directory. This is the approach that Git has with its .git sub-directory as shown in Figure 4.7. This would avoid potential corruption to any hashes placed directly in the files, as it would be completely separate from the source

```java
/*rxebWcQcAHedJwre*/
public class BoilerActuator {
    public execute(int temperatureDifference, boolean boilerStatus) {

    }
}
```

Listing 4.18: An example of generating hashes at the begining of generated file.

```java
/*rxeb*/ public class BoilerActuator {
/*WcQc*/    public execute(int temperatureDifference, boolean boilerStatus
    ) {
/*Ghes*/
/*AHed*/    }
/*Jwre*/ }
```

Listing 4.19: An example of generating hashes at the begining of every generated line.

code. An added benefit of this approach is that it's the only one that enables intelligent identification of modifications and deletions to the original lines of code; thereby informing the user on how to correct these corruptions. However, saving a master copy of every file would double the size of each project. Also these folders would need to be kept in sync when files are renamed, deleted etc.

```
\generated-files
  \file1.java
  \file2.java
  \file3.java
\master
  \file1.java
  \file2.java
  \file3.java
```

Figure 4.7: An example of preserving the copy of the generated files in a "master" directory.

Separate hash files. Incorporating a separate hash into different files could be another solution, as shown in Figure 4.8. However, this may cause confusion as in the case when a source file is deleted and the hash file remains, or the hash files are accidentally deleted aside from the source code. Also, this approach has an higher memory consumption than other methods, as typically small files (which these would be) are saved as larger files than necessary on the disk, and this creates an inefficiency for the engine to read / write hashes into many other files.

```
\generated-files
  \file1.java
  \file2.java
  \file3.java
\hashes
  \file1.java.hashes
  \file2.java.hashes
  \file3.java.hashes
```

Figure 4.8: An example of preserving the hashes of the generated files in a "hashes" directory.

Separate hashes file. If all the hashes are preserved into one separate file, this may be considered another alternative approach, shown in Figure 4.9. However, the shortcomings of this approach are an increased time for the engine to read / write hashes into another file, and the increased memory usage of the algorithm to keep all the hashes in memory for all files at once, or to read the file multiple times over.

```
\generated-files
    \file1.java
    \file2.java
    \file3.java
\hashes.txt
```

Figure 4.9: An example of preserving all the hashes in a separate file.

4.6 Summary

This chapter presents an approach that facilitates the process of embedding and preserving hand-written text into the generated files of M2T transformations. This approach has been implemented on top of the existing M2T language Epsilon Model Generation Language (EGL). Moreover, this approach has been evaluated for its correctness and performance.

Chapter 5

Synchronised Regions

Chapter 2 presented a detailed review of round-trip engineering in model-to-text transformations (M2T). The review showed that most of the previous research focused on supporting round-trip engineering in M2M rather than round-trip engineering in M2T transformations.

Chapter 3 motivates the need of integrating hand-written text in M2T transformations to achieve full code generation. However, it was also emphasised that using existing approaches, such as protected regions, can lead to complex problems (as described in Section 3.3.2). Chapter 4 presents a novel approach (Automated Line Based Merging) for integrating hand-written text in M2T languages. However, an example also highlighted the shortcomings of Automated Line Based Merging Approach, in which hand-written text is integrated into generated artefacts without propagating changes back to the source model. This results, in case like the one in the example, in the violation of the sing source of truth principle, as the information of the system is in two or more different places.

This chapter presents another novel approach, that of *sync* regions, that solve this shortcoming. *Sync* regions are declared in templates and are intended to enable developers to add hand-written text within them. More importantly, *sync* regions propagate this addition back to the source model to maintain consistency between the models and their generated files. The exact scope of this approach is discussed in Section 5.4.

Chapter Structure: Section 5.1 introduces the reader to the concept of *sync* regions. Section 5.2 describes how M2T transformation languages can be extended with *sync* regions. It also presents the implementation of the *sync* regions technique in an existing M2T language (EGL). Section 5.3 outlines the results of the evaluation that was undertaken. Section 5.4 discusses the practicability and the limitations of the *sync* regions technique. Lastly, Section 5.5 concludes by summarizing the contents of this chapter.

5.1 Introduction

A *sync* region is a region in a generated file that is appropriately ring-fenced using identifiable start/end comments. It encloses content that needs to be kept in sync with a specific slot (pair of model elements and attributes) in the model. The synchronised regions technique consists of two steps to support synchronization in M2T transformations. Declaring *sync* regions in the template allows developers to integrate the hand-written text as a first step. As a second step, developers re-run the transformation to propagate any changes that occurred inside these regions back into the models. Propagating changes back to the source models depends on the name of the attribute and the model element, information which is included in the start comment of every *sync* region. In the next section, *sync* regions and the aforementioned steps are explained in detail.

5.2 Extending EGL with *Sync* Regions

This section discusses how EGL can be extended with support for *sync* regions; noting, however, that the same principles can be used to extend any other template-based M2T language in a similar manner. The reason for choosing EGL was that in addition to the local expertise, it is a powerful and mature model-to-text transformation language.

5.2.1 Assumptions

This section provides a list of assumptions for using the Sync Regions Technique. Thus the user of this technique or other researchers looking to expand this work may benefit from having these clear assumptions.

The prototypical extension works with EMF-based models [141] persisted in the XMI format, where each element has a unique persistent ID, and is limited to M2T transformations that consume a single model as an input.

Target Language Support Comment. One assumption, in order for this approach to be applicable, is that the target language needs to have some character comments. For example, if the M2T is generating a JSON file, this approach would not be applicable because JSON does not support comments. Presently, Java-style comments (and any other language that uses a similar comment structure to Java, like HTML, Python, Ruby, etc.) are supported.

Source Model Modification. Another assumption in using this approach is that between generation and synchronisation, the source model is not edited in any way. More details on this are provided in the limitations section (Section 5.4) of this Chapter.

Single Input Model. Currently, the proposed approach expects two inputs: one model and one folder, which contains all the generated artefacts. This assumption is discussed in detail in (Section 5.4) of this Chapter.

Embedded Code Fragments using *Sync* Regions. The proposed approach expects the handwritten text to be integrated inside the body of *sync* regions. If the handwritten text was integrated outside the body of *sync* regions, then this additional text will not be preserved during the regeneration.

Direct Changes to the Embedded Code Fragments. It is important to highlight that in the proposed approach it was assumed that the code fragments should not be modified within the modelling framework. Changes are expected to be made only within the IDE (i.e. the generated files). This assumption is discussed in detail in the Limitations section (Section 5.4) of the Chapter.

Deleted *Sync* Regions. While the synchronisation algorithm can cope with inconsistently updated and malformed *sync* region markers, it cannot cope with *sync* regions being deleted altogether from generated files. Ideally, such missing *sync* regions should be reported to the user, but this cannot be achieved without keeping additional metadata outside the generated files, which is undesirable.

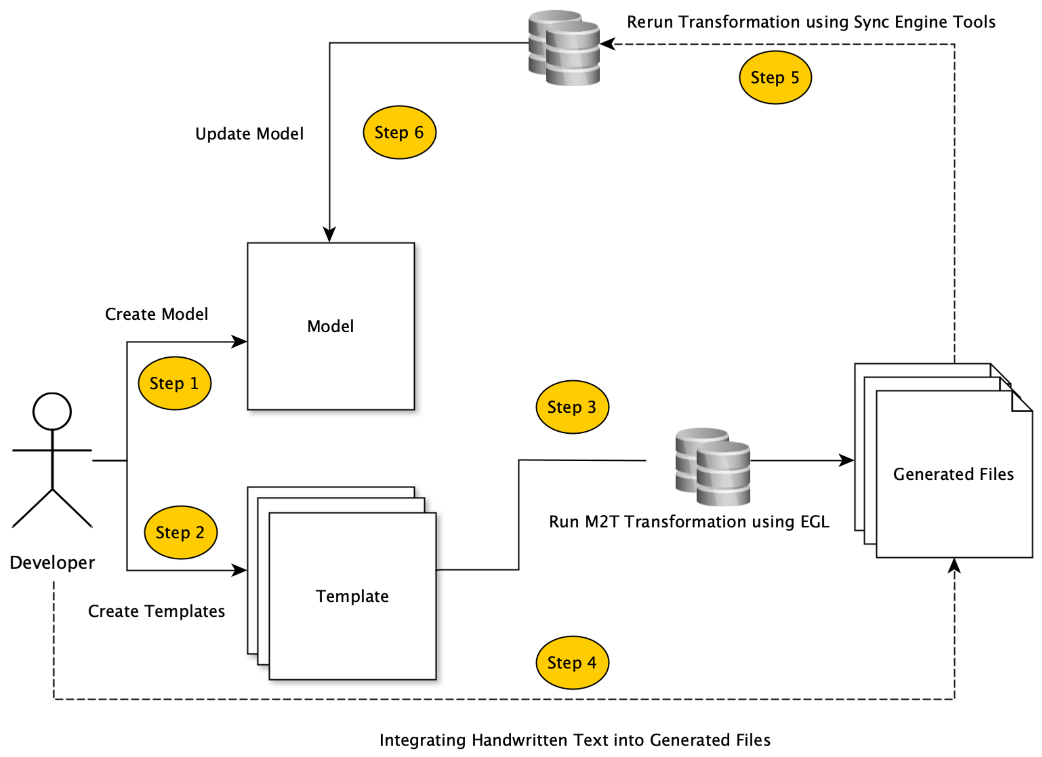

Rerun Transformation using Sync Engine Tools
Step 5
Update Model
Step 6
Create Model
Model
Step 1
Step 3
Generated Files
Developer
Step 2
Run M2T Transformation using EGL
Create Templates
Template
Step 4
Integrating Handwritten Text into Generated Files

Figure 5.1: An overview of the proposed approach for synchronizing source model with target generated files using EGL.

Implementation of *Sync* Regions in EGL

This section presents the steps for developing the synchronisation technique in detail. An overview of the approach is given in Figure 5.1. Firstly, it is expected that developers/users have created model (step $\textcircled{1}$) and text generation templates they would like to run against this model (step $\textcircled{2}$). The M2T transformation is then executed (step $\textcircled{3}$). This transformation will result in files in the target languages that have been specified by the developer. A potential problem arises when there are existing generated files that the developers may have made modifications to (step $\textcircled{4}$), as the M2T will overwrite these modifications. Another problem is that EGL offers no consistency in adding such modifications to the generated files without updating the source model, which can be provided by the following step.

The implementation of a *sync* regions approach enables users to integrate the hand-written text inside *sync* regions (which EGL will detect), and to instruct the *sync* engine tools to preserve them. In these cases, the sync engine is invoked (step ⑤) considering both the potentially user-modified (hand-written text) files and the newly generated model files. This process preserves hand-written text and updates the source model (step ⑥) as the final output of the transformation process, which provides a more convenient developer workflow. All the steps of the proposed approach are explained in more detail in the following sections.

5.2.2 Extending EGL with *Sync* Regions

EGL has been extended with two additional methods to specify *sync* regions[1]:

- *startSync(String startComment, String id, String attribute)*: This emits a single line comment in the target file, starting with the *startComment* character sequence (e.g. // for Java), which denotes the start of a *sync* region, and contains the *id* of the model element and the name of its attribute that the content of the *sync* region needs to be kept in sync with. A variant of the method with an extra *endComment* parameter is also available to accommodate languages that require both a prefix and a suffix for their comments (e.g., HTML).
- *endSync()*: Emits a single-line comment that marks the end of the active *sync* region.

The use of the methods above is demonstrated in Listing 5.1, which is an extended version of the original Listing 3.3 template that generates Java classes from individual components. In the extended version of the template, three new lines have been added (lines 3-5) and the output of executing it against the *TemperatureController* component is shown in Listing 5.2. Line 3 of the template, produces the comment in line 3 of the generated file, which denotes the start of a *sync* region. The generated comment starts with the // character sequence as instructed by the first argument of the *startSync* method. It continues with the *sync* token that allows the synchronisation engine described later on to distinguish *sync* region comments from general

[1]All source code for *sync* regions approach are available at `https://github.com/soha500/EglSyncNew/`

```
1  public class [%=c.name%] {
2      public [%=c.outPort.type%] execute([%=c.inPorts.collect(p|p.type + "
           " + p.name).concat(", ")%]) {
3        [%=out.startSync("//", c.id, "behaviour")%]
4
5        [%=out.endSync()%]
6      }
7  }
```

Listing 5.1: Extended version of the template of Listing 3.3 with a *sync* region

```
1  public class TemperatureController {
2      public int execute(int temperature, int targetTemperature) {
3        //sync _bfpnFUbFEeqXnfGWlV2_8A, behaviour
4
5        //endSync
6      }
7  }
```

Listing 5.2: The result of executing the template of Listing 5.1 against the *TemperatureController* component

comments in the generated file, and then it contains the ID of the component (*_bfpnFUbFEeqXnfGWlV2_8A*[2]) and the name of the attribute against which the content of the *sync* region should be synchronised (e.g., *behaviour*). Line 4 prints the content of the behaviour attribute of the component (empty in the initial version of the model), and Line 5 produces the *//endSync* comment in the generated file, that denotes the end of the *sync* region.

The algorithm identifies the correct attribute to store the hand-written text by concatenating the model element's ID and the attribute name. Thus, as each model element has a unique id, it is acceptable to repeat the same attribute name *for different model elements* in more than one region. However, repeating the same attribute name for the *same model element* leads to inconsistencies that need to be solved - an approach for dealing with such inconsistencies is described in Section 5.2.3.

[2]This is an auto-generated XMI ID produced by the Eclipse Modelling Framework, that was used to implement the component-connector DSL and the sample instance model.

```
1 public class TemperatureController {
2     public int execute(int temperature, int targetTemperature) {
3         //sync _bfpnFUbFEeqXnfGWlV2_8A, behaviour
4             return temperature − targetTemperature;
5         //endSync
6     }
7 }
```

Listing 5.3: Extended *TemperatureController* class with hand-written behaviour

5.2.3 Synchronising *Sync* Regions with Model Elements

A developer can now specify the behaviour of the *TemperatureController* component within the produced *sync* region of the generated *TemperatureController* Java class, as shown in line 4 of Listing 5.3, benefiting from modern IDE features such as code completion and syntax highlighting. Once they have made the desirable changes to the behaviour of generated components, the next step is to trigger a synchronisation mechanism (the second part of the proposed approach), which identifies and copies the hand-written behaviour into the *behaviour* attributes of the respective components in the source model.

A requirement of the proposed approach is that, between generation and synchronisation, only insertions of new code is permitted, with original source lines not edited in any way. This limitation is explained in detail in Section 5.4 of this Chapter.

The synchronisation algorithm, which consists of 3 steps, is described below:

Step A: *Sync* Region Identification

The synchronisation algorithm receives two inputs: the root directory, under which files, already generated by the M2T exist, and the EMF model to be synchronised. The algorithm recursively scans all files under the root directory and identifies *sync* regions that start and end with appropriate comments (Algorithm 6, line 2). For each *sync* region, the algorithm checks (see Algorithm 6, lines 4-12) how well formed it is. This means that the element ID and attribute of the region are correctly specified, that they correspond to valid elements and attributes in the model, and that the text within the *sync* region can be converted to a value compatible with the type of the respective attribute. A complete list of errors is provided in Section 5.3.1 where correctness is evaluated. This check is employed on line 5. If any of the regions are found to be malformed, the algorithm exits with an appropriate error message (lines 7-8).

For well-formed regions of the file (line 10), the algorithm iterates over each region's ID, attribute, and value (line 11). The ID and attribute are concatenated with a period in between (line 12) to form a key, and this key along with the region value is inserted into a dictionary of region keys to values (line 13). Once all files in the folder have been processed, the dictionary is returned (line 16).

Algorithm 6 How synchronise algorithm identifies *sync* regions

```
1: function SyncRegionsOfFolder($path$)
2:     $ListOfFiles \leftarrow$ all files recursively found in $path$
3:     $RegionKeysToValues \leftarrow \emptyset$
4:     for all $f \in ListOfFiles$ do
5:         $BadRegions \leftarrow$ all badly-formed or incomplete sync regions in $f$
6:         if any $BadRegions$ then
7:             print "Badly-formed or incomplete sync region in " $f$
8:             return $null$
9:         end if
10:        $Regions \leftarrow$ all well-formed sync regions in $f$
11:        for all $(ID, Attribute, Value) \in Regions$ do
12:            $Key \leftarrow$ concatenate(ID, ".", attribute)
13:            insert $(Key, Value)$ into $RegionKeysToValues$
14:        end for
15:    end for
16:    return $RegionKeysToValues$
17: end function
```

Step B: *Sync* Region Consistency Checking

Since, in principle, the same attribute of the same model element can appear in multiple *sync* regions across the generated code-base, before the model is updated, it needs to ensure the consistency of *sync* regions that refer to the same element and attribute.

Lines 2-3 of Algorithm 7 prepares a dictionary of region keys to values, and a dictionary of model attributes to values. For all model attributes and their values (line 4) the values for the region key are extracted from the keys to values dictionary (line 5). Lines 6-9 exit the algorithm with an error if no values were found for this key. Line 10 prepares a set of unique region values (which are strings) for the following cases:

- If the regions have the same value, they are marked as consistent: Lines 1-6 of Listing 5.4 show that for the same model element and the same attribute, the values are the same and in this case different from the value stored in the model. The model is updated with the new, consistent value. This is illustrated in Algorithm 7 lines 11-16.
- If the regions have two unique values and one of them is the same as the value of the attribute in the source model, then the other (different) value is marked as the "new" value for the attribute: Lines 8-13 of Listing 5.4 show that for the same attribute values only one of them is different from the value stored in the model. The model is then updated with the "new" value. This is illustrated in Algorithm 7 lines 17-20.
- If the regions have two or more unique values, none of which correspond to the value of the attribute in the model, they are marked as inconsistent; Lines 15-20 of Listing 5.4 show that for the same model element the values differ and are both different from the value stored in the model. Consequently, the model is not updated and the inconsistency is reported to the user. This is illustrated by Algorithm 7 line 21, which is reached when the previous two cases did not apply. It is expected that the user will remedy these inconsistencies manually for the synchronisation algorithm to run to completion again.

Step C: Model Updating

At this point, *sync* regions have been verified to be well-formed (step A) and free of conflicts (step B). As such, the algorithm can proceed with updating the attributes of the model elements to which *sync* regions refer. For each

```
//sync _bfpnFUbFEeqXnfGWlV2_8A, behaviour
  return temperature - targetTemperature;
//endSync
//sync _bfpnFUbFEeqXnfGWlV2_8A, behaviour
  return temperature - targetTemperature;
//endSync

//sync _bfpnFUbFEeqXnfGWlV2_8B, behaviour
  return temperature - targetTemperature;
//endSync
//sync _bfpnFUbFEeqXnfGWlV2_8B, behaviour
  return temperature + targetTemperature;
//endSync

//sync _bfpnFUbFEeqXnfGWlV2_8B, behaviour
  return temperature > targetTemperature;
//endSync
//sync _bfpnFUbFEeqXnfGWlV2_8B, behaviour
  return temperature < targetTemperature;
//endSync
```

Listing 5.4: Sync regions checking examples.

element/attribute involved, the content of the respective *sync* region is parsed to the type of the attribute and the parsed value is assigned to the attribute. Once all elements/attributes have been updated, the model is saved to disk.

5.3 Evaluation

This section outlines the results of the evaluation of the correctness, scalability, and generalisability of the *sync* region implementation approach and reflects on its applicability and known limitations[3].

[3]All unit tests, input models, generated files, raw and analysed results for all the experiments presented in this section are available at `https://github.com/soha500/EglSyncNew/`

Algorithm 7 How the synchronisation algorithm checks for and responds to value consistencies

```
function SynchroniseFolder($path$, $model$)
    $RegionKeysToValues \leftarrow$ SyncRegionsOfFolder($path$)
    $AttributesToValues \leftarrow$ element attribute values of $model$
    for all $(Key, ModelValue) \in AttributesToValues$ do
        $Values \leftarrow$ values of $Key$ in $RegionKeysToValues$
        if $Values$ is empty then
            print "Key not found in model"
            return
        end if
        $UniqueValues \leftarrow$ unique values of $Values$
        if count of $UniqueValues = 1$ then
            if $ModelValue$ differs from 1st of $UniqueValues$ then
                update $model$ attribute $Key$ with 1st of $UniqueValues$
            end if
            continue
        end if
        if count of $UniqueValues = 2$ and $UniqueValues$ contains $ModelValue$ then
            update $model$ attribute $Key$ with value of $UniqueValues$ differing from $ModelValue$
            continue
        end if
        print "There are two or more different values from the one in the model"
    end for
end function
```

5.3.1 Correctness

To build confidence on the correctness of the developed approach, several unit tests have been developed using the JUnit library to ensure that the synchronisation algorithm behaves as expected under normal circumstances (well-formed and conflict-free *sync* regions) and gracefully fails when models or generated files are modified manually in inconsistent ways.

***Sync* Regions Content.** The first part of the correctness tests was to focus on the content of the *sync* regions in the generated files when compared to the respective one in the model. A summary of the possible cases that may happen when developers are using *sync* regions' approaches can be listed as follows:

(i) There is one *sync* region that contains the same value in the model.

(ii) There is one *sync* region with a different value from the one in the model.

(iii) There are two *sync* regions with the same value as the one in the model.

(iv) There are two *sync* regions with one different value to the value in the model.

(v) There are two *sync* regions but one has a different value to the one in the model.

(vi) There are two *sync* regions and they have two different to from the value in the model.

(vii) There are three *sync* regions, which contain the same value as in the model.

(viii) There are three *sync* regions but with one with a different value to the one in the model.

(ix) There are three *sync* regions but with two different values, but one of them having same value as in the model.

(x) There are three or more *sync* regions but with two or more different values to the value in the model.

Table 5.1 demonstrates the results of the tests that were undertaken. The results show that the proposed approach is working as expected when any of the possible cases occurred, and by warning the developers where the error is by displaying a clear message to the console. To avoid duplication only the result of one test for one type (String) is shown. However, tests were written for all the types (e.g. int, double, float, and boolean) and the results were similar to the "String" type.

Test	Number of ***Sync*** Regions in Generated Files	Input (Content of each ***Sync*** Region)	Respective Value in The Model	Expected Output	Result of The Test
i	One *sync* region	//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerActuator** //endSync	name="BoilerActuator"	name="BoilerActuator"	Pass
ii	One *sync* region	//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController** //endSync	name="BoilerActuator"	name="BoilerController"	Pass

iii	Two *sync* regions	//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerActuator**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerActuator**
		//endSync
			name="BoilerActuator"	name="BoilerActuator"	Pass

iv	Two *sync* regions	//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController**
		//endSync
			name="BoilerActuator"	name="BoilerController"	Pass

v	Two *sync* regions	//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerActuator**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController**
		//endSync
			name="BoilerActuator"	name="BoilerController"	Pass

vi	Two *sync* regions	//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **Controller**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController**
		//endSync
			name="BoilerActuator"	**an inconsistency error**	Pass

vii	Three *sync* regions	//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerActuator**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerActuator**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerActuator**
		//endSync
			name="BoilerActuator"	name="BoilerActuator"	Pass

viii	Three *sync* regions	//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController**
		//endSync
			name="BoilerActuator"	name="BoilerController"	Pass

ix	Three *sync* regions	//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerActuator**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController**
		//endSync
			name="BoilerActuator"	name="BoilerController"	Pass

x	Three *sync* regions	//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerActuator**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **BoilerController**
		//endSync
		
		//sync _bfpnGUbFEeqXnfGWlV2$_8A,$ *name* **Controller**
		//endSync
			name="BoilerActuator"	**an inconsistency error**	Pass

Table 5.1: A result of testing the possible cases for using *sync* regions.

```java
public class TemperatureController {
    public int execute(int temperature, int targetTemperature) {
        //sync , behaviour
            return temperature − targetTemperature;
        //endSync
    }
}
```

Listing 5.5: An example of *sync* region without ID Element.

```java
public class TemperatureController {
    public int execute(int temperature, int targetTemperature) {
        //sync _bfpnFUbFEeqXnfGWlV2_8A,
            return temperature − targetTemperature;
        //endSync
    }
}
```

Listing 5.6: An example of *sync* region without attribute.

***Sync* Regions Syntax.** The second part of the correctness test focused on the syntax of the *sync* regions. The possible cases that could occur in this aspect could be listed as follows:

Test 1: *Sync* Region without ID Element. If at least one of the *sync* regions does not contain the ID element as shown in Listing 5.5.

Test 2: *Sync* Region without Attribute Name. If at least of one of the *sync* regions does not contain the attribute name as shown in Listing 5.6.

Test 3: *Sync* Region without startSync Token. If at least one of the *sync* regions does not contain the *startSync* token as shown in Listing 5.7.

Test 4: *Sync* Region without endSync Token. If at least one of the *sync* regions does not contain the end endSync token as shown in Listing 5.8.

Test 5: Respective Element does not Exist in Source Model. If at least one of the *sync* regions contains an element that no longer exists

```
public class TemperatureController {
    public int execute(int temperature, int targetTemperature) {

            return temperature - targetTemperature;
        //endSync
    }
}
```

Listing 5.7: An example of *sync* region without startSync Token

```
public class TemperatureController {
    public int execute(int temperature, int targetTemperature) {
        //sync _bfpnFUbFEeqXnfGWlV2_8A, behaviour
            return temperature - targetTemperature;
    }
}
```

Listing 5.8: An example of *sync* region without endSync Token.

in the source model. For example, if the ID element (*_bfpnFUbFEeqXnfG-WlV2_8A*) for the *sync* region in Listing 5.9 was modified or removed from the source model.

Test 6: Attribute Name does not Exist in Source Model. If at least one of the *sync* regions contains an attribute name that no longer exists in the source model. For example, if the attribute name (*behaviour*) for the *sync* region in Listing 5.10 was modified or removed from the source model.

```
public class TemperatureController {
    public int execute(int temperature, int targetTemperature) {
        //sync _bfpnFUbFEeqXnfGWlV2_8A, behaviour
            return temperature - targetTemperature;
        //endSync
    }
}
```

Listing 5.9: An example of *sync* region without respective element in source model.

```
public class TemperatureController {
    public int execute(int temperature, int targetTemperature) {
        //sync _bfpnFUbFEeqXnfGWlV2_8A, behaviour
            return temperature − targetTemperature;
        //endSync
    }
}
```

Listing 5.10: An example of *sync* region without attribute name in source model.

```
public class TemperatureController {
    public int execute(int temperature, int targetTemperature) {
        //sync _bfpnFUbFEeqXnfGWlV2_8A, behaviour
            123
        //endSync
    }
}
```

Listing 5.11: An example of *sync* region with incompatible content.

Test 7: Incompatible Content. If the type of the content in the *sync* region is not compatible with the type in the respective element in the source model. For example, if the user adds integer content for the *sync* region in Listing 5.11, but the attribute behaviour was defined as a string in the source model.

5.3.2 Performance and Scalability

To assess the performance and scalability of the implementation, and to ensure that it was free from unnecessary bottlenecks, the M2T transformation discussed in Section 5.2 was executed on models of various sizes, producing 5 sets of files ranging from 2,000 to 10,000 files (with a step of 2,000 files). Each experiment was repeated 3 times, collecting the average time and memory usage, for each of the following scenarios [4]:

- Each generated file had *one sync* region.

[4]The experiments were executed on a laptop computer with the following specifications: MacOS Mojave 106.14., Intel Core i7, 2-cores @ 3.5Ghz, 1x16 GB 2133MHz LPDDR3 RAM

# Files	# *Sync* Regions	Average (Total) Time (in s)	Average Memory Used (in MB)
2000	1	**40.83**	316.77
2000	2	88.61	58.97
2000	3	124.60	48.43
2000	4	168.39	283.53
2000	5	212.60	276.61
4000	1	167.89	97.24
4000	2	345.54	75.94
4000	3	545.45	94.69
4000	4	740.26	852.79
4000	5	950.55	546.58
6000	1	384.27	**1131.14**
6000	2	754.12	968.73
6000	3	1192.69	599.44
6000	4	1736.60	1030.33
6000	5	2224.22	955.47
8000	1	724.76	968.26
8000	2	1510.34	827.48
8000	3	2341.29	842.78
8000	4	3188.45	777.47
8000	5	4221.06	680.05
10000	1	1198.81	244.07
10000	2	2368.72	596.18
10000	3	3678.04	624.46
10000	4	5000.96	507.28
10000	5	**6366.22**	231.46

Table 5.2: Average execution time and memory consumption form the different number of files and *sync* regions

- Each generated file had *two sync* regions.
- Each generated file had *three sync* regions.
- Each generated file had *four sync* regions.
- Each generated file had *five sync* regions.

In this experiment, the values included in the generated files were *always different from those stored in models*, thus the values in every element in the source model had to be updated. The solution was evaluated with up to 10,000 files, a number that significantly exceeds the number of files produced by typical M2T transformations. The results are summarised in Table 5.2 and in Figures 5.2-5.5.

Figure 5.2 presents the results of the average total time required for the execution of the synchronisation for the 5 different sizes of generated file sets and the 5 scenarios with the different number of *sync* regions. Each line represents one file set, while the horizontal axis is the number of *sync* region(s) for each file in the set. As can be seen, the execution time increases linearly as the number of *sync* regions increases for all the different sets of files. However, it is not clear if the execution time increases linearly or has a exponential trend when the number of files increases, while keeping the number of *sync* regions the same (see Figure 5.4).

In terms of absolute values, in the scenario of having 2,000 files with 1 *sync* region in each, the proposed approach required approximately 40 seconds to complete the synchronisation (see Table 5.2 - top highlighted value). For the biggest experiment (i.e. having 10,000 files each of which had 5 *sync* regions) the average time taken for the 3 runs was approximately 1 hour and 45 minutes (6,366.22 seconds). Even in this extreme scenario the synchronisation completed successfully.

Figure 5.4 presents the same data but this time as the number of generated files is increasing for each of the scenarios for the same number of *sync* regions. Each line in Figure 5.4 represents a scenario with a fixed number of *sync* regions and the 5 data points on the line represent the 5 different sizes of file sets.

Finally, in Figures 5.3 and 5.5 the average memory consumption is shown as the number of *sync* regions increases (keeping the number of files fixed) and as the number of files increases (keeping the number of *sync* regions fixed), respectively. There is no clear correlation, which is explained by the fact that Java garbage collection is clearing up memory when needed. What is of importance is that in the worst case the prototype solution consumed about 1GB of memory (1131.14MB - see highlighted value in the memory consumption column of Table 5.2).

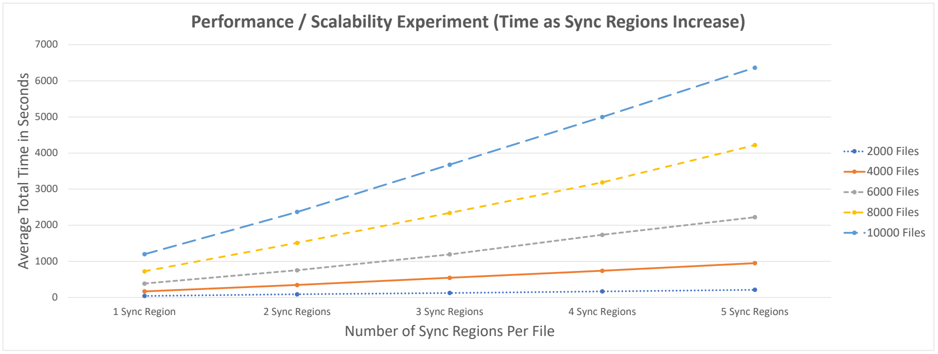


Figure 5.2: Results of measuring the average time for different size of models and number of *sync* regions as the number of *sync* regions increases.

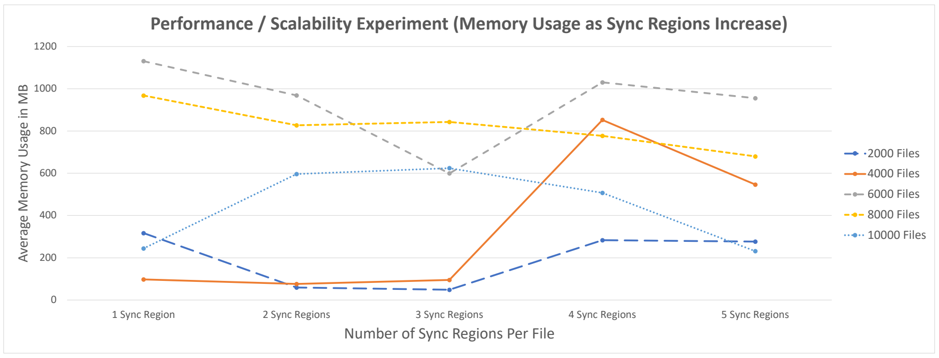


Figure 5.3: Results of measuring the average memory usage for different size of models and number of *sync* regions as the number of *sync* regions increases.

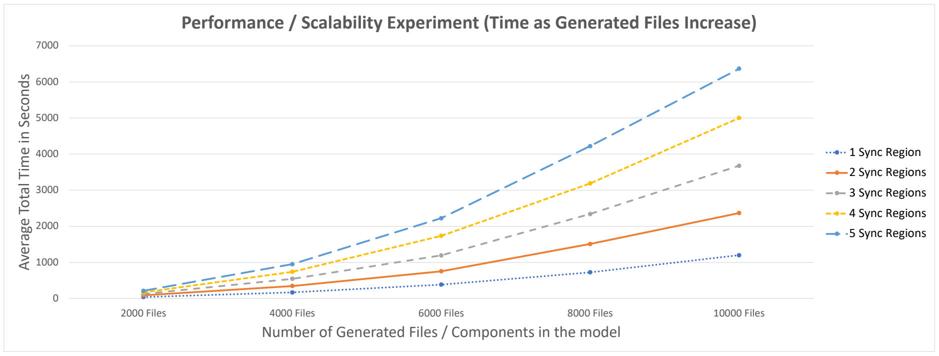


Figure 5.4: Results of measuring the average time for different size of models and number of *sync* regions as the number of files increases.

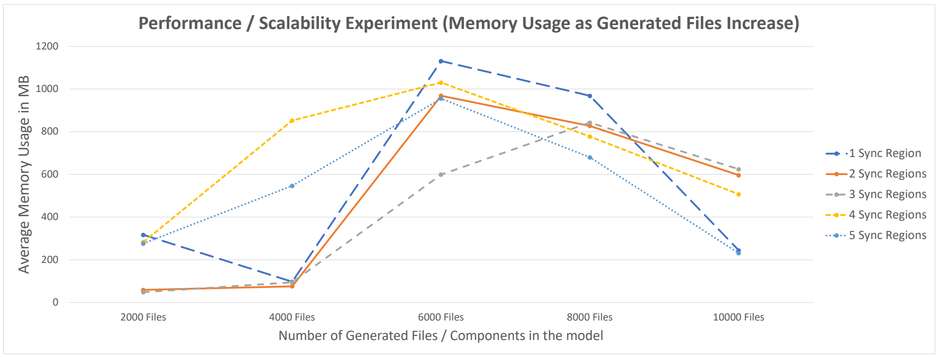


Figure 5.5: Results of measuring the average memory usage for different size of models and number of *sync* regions as the number of files increases.

5.3.3 Generalisability

Distinguish Between Different Comments. To make the EGL engine distinguish between different comments for each language, the user is required to declare at the first line of the EGL template the target programming language of the files that they wish to generate. This way the *sync* regions engine will look for the appropriate comments according to the flag at the fist line. As the default behaviour of the *sync* regions engine is to locate Java-like comments, the users do not have to declare this for Java (and any other language in which comments are identical to Java's).

To assess the generalisability of the synchronisation algorithm, the M2T transformation was adapted to generate files for different languages, such as Java, Python, HTML, and Ruby. The proposed approach was tested by repeating the synchronisation (in the form of JUnit tests) 100 times for each of the aforementioned programming languages. In this experiment, the content of *all* regions in the generated files was different to the corresponding values in the source model. The following section describes the test in more detail, together with examples:

Template for Different Programming Languages. (Step 1) Declaring a *sync* region in different programming languages as shown in Listings (5.12 5.13 5.14 and 5.15) respectively.

Generated Files for Different Programming Languages. (Step 2) The results of running the above templates are shown in Listings (5.16 5.17 5.18 and 5.19) respectively.

Adding Hand-written Code within *Sync* Regions. (Step 3) New behaviours were added for each language as shown in Listings (5.20 5.21 5.22 and 5.23) respectively.

Propagate Changes Back to The Models. (Step 4) Finally, all the models (i.e., the respective attributes) are updated with new behaviours for each language as shown in Listings (5.6 5.7 5.8 and 5.9). The synchronisation algorithm passed all the tests and updated the models as expected. Table 5.3 presents the open/close comment format used for each of the target programming languages and the results, i.e. the synchronisation algorithm passed all the tests and updated the models as expected.

Target language	Opening Comment Format	Closing Comment Format	Test Result
Java	// or /*	*/	Pass
HTML	<!—	—>	Pass
Python	#	N/A	Pass
Ruby	# or =begin	=end	Pass

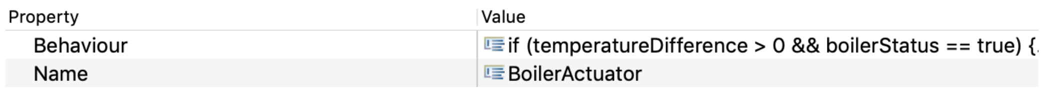
Table 5.3: Generalisability experiment results

```
package syncregions;

public class [%=c.name%] {
    public [%=c.outPort.type%] execute([%=c.inPorts.collect(p|p.type + " " + p.name).concat(", ")%]) {
      [%=out.startSync("//", c.id, "behaviour")%]

      [%=out.endSync()%]
    }
}
```

Listing 5.12: An example of EGL template to generate Java code with a *sync* region

```
<!--HTML-->
<td> [%=c.name%]
        [%=out.startSync("<!--", "-->", c.id, "htmlBehaviour")%]

        [%=out.endSync()%]
</td>
```

Listing 5.13: An example of EGL template to generate an HTML page with a *sync* region



Property	Value
Behaviour	if (temperatureDifference > 0 && boilerStatus == true) {
Name	BoilerActuator

Figure 5.6: Result of updating the *BoilerActuator* component in the model with behaviour written in Java.

```
#Python
    def execute(self, [%=c.inPorts.collect(p|p.type + " " + p.name).concat
        (", ")%]):
      [%=out.startSync("#" , c.id, "pythonBehaviour")%]

      [%=out.endSync()%]
```

Listing 5.14: An example of EGL template to generate Python code with a *sync* region

```
#Ruby
    def execute([%=c.inPorts.collect(p|p.type + " " + p.name).concat(", ")
        %])
      [%=out.startSync("#", c.id, "rubyBehaviour")%]

      [%=out.endSync()%]
    end
```

Listing 5.15: An example of EGL template to generate Ruby code with a *sync* region

```
package syncregions;

public class BoilerActuator {
    public int execute(int temperatureDifference, boolean boilerStatus) {
      //sync _bfpnGUbFEeqXnfGWlV2_8A, behaviour

      //endSync
    }
}
```

Listing 5.16: An example of generating Java code for the *BoilerActuator* component with one *sync* region

```
<!--HTML-->
<td> BoilerActuator
    <!--sync _bfpnGUbFEeqXnfGWlV2_8A, htmlBehaviour -->

    <!--endSync -->
</td>
```

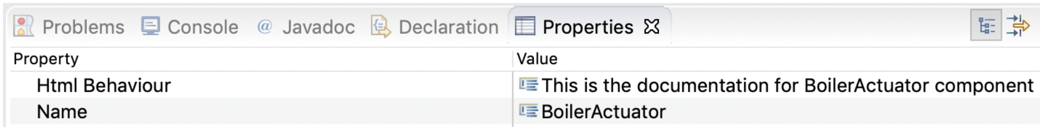
Listing 5.17: An example of generating an HTML page for the *BoilerActuator* component with one *sync* region

```
#Python
    def execute(self , int temperatureDifference, boolean boilerStatus):
        #sync _bfpnGUbFEeqXnfGWlV2_8A, pythonBehaviour

        #endSync
```

Listing 5.18: An example of generating Python code for the *BoilerActuator* component with one *sync* region

```
#Ruby
    def execute(int temperatureDifference, boolean boilerStatus)
      #sync _bfpnGUbFEeqXnfGWlV2_8A, rubyBehaviour

      #endSync
    end
```

Listing 5.19: An example of generating Ruby code for the *BoilerActuator* component with one *sync* region



Property	Value
Html Behaviour	This is the documentation for BoilerActuator component
Name	BoilerActuator

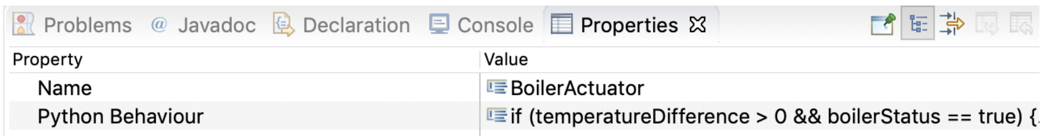
Figure 5.7: Result of updating the *BoilerActuator* component in the model with hand-written HTML markup.

```
package syncregions;

public class BoilerActuator {
    public int execute(int temperatureDifference, boolean boilerStatus) {
      //sync _bfpnGUbFEeqXnfGWlV2_8A, behaviour
        if (temperatureDifference > 0 && boilerStatus == true) {
          return 1;
        } else if (temperatureDifference < 0 && boilerStatus == false) {
          return 2;
        } else
          return 0;
      //endSync
    }
}
```

Listing 5.20: An example of adding hand-written Java code for the *BoilerActuator* component inside a *sync* region.

```
<!--HTML-->
<td> BoilerActuator
    <!--sync _bfpnGUbFEeqXnfGWlV2_8A, htmlBehaviour -->
    This is the documentation for BoilerActuator component of the system
    <!--endSync -->
</td>
```

Listing 5.21: An example of adding hand-written text in HTML code for the *BoilerActuator* component inside a *sync* region.



Property	Value
Name	BoilerActuator
Python Behaviour	if (temperatureDifference > 0 && boilerStatus == true) {

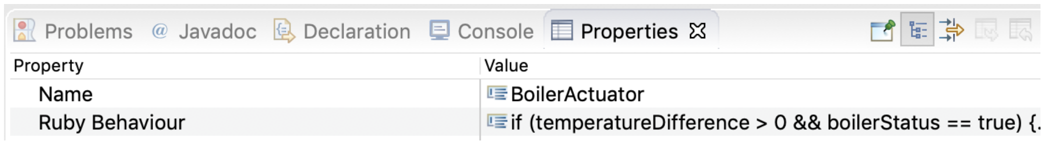
Figure 5.8: Result of updating the *BoilerActuator* component with new behaviour in python model.

```
#Python
    def execute( self ,  int  temperatureDifference, boolean boilerStatus):
        #sync _bfpnGUbFEeqXnfGWlV2_8A, pythonBehaviour
         if  (temperatureDifference > 0 && boilerStatus == true) {
            return  1;
         }  else  if  (temperatureDifference < 0 && boilerStatus == false) {
            return  2;
         }  else
            return  0;
        #endSync
```

Listing 5.22: An example of adding hand-written Python code for the *BoilerActuator* component inside a *sync* region.

```
#Ruby
    def execute(int  temperatureDifference, boolean boilerStatus)
      #sync _bfpnGUbFEeqXnfGWlV2_8A, rubyBehaviour
          if  (temperatureDifference > 0 && boilerStatus == true) {
             return  1;
          }  else  if  (temperatureDifference < 0 && boilerStatus == false) {
             return  2;
         }  else
            return  0;
      #endSync
    end
```

Listing 5.23: An example of adding hand-written Ruby code for the *BoilerActuator* component inside a *sync* region.



Property	Value
Name	BoilerActuator
Ruby Behaviour	if (temperatureDifference > 0 && boilerStatus == true) {.

Figure 5.9: Result of updating the *BoilerActuator* component with new behaviour in ruby model.

5.3.4 Threats to Validity

In this section all the threats that could make using the proposed approach invalid are discussed:

Supported Comment Formats. In order to use *sync* regions that are situated in the code, it must be possible to use comments. The comments used in the presented implementation are for marking the start and end of the regions. Not all programming languages have the same style of comments, so a sync-region solution would have to offer support for each desired language.

Corrupted *Sync* Regions. As the *sync* regions are situated directly in the code, the users are frequently editing around them and have nothing to prevent them from accidentally or wilfully modifying the regions; even potentially corrupting them. Examples of these possible modifications are as follows:

- Delete/modify by mistake the whole start line of the *sync* regions, as shown in the first *sync* region in Listing 5.24 (lines 4-6).
- Delete/modify the ID of the *sync* regions, as shown in the second *sync* region (Listing 5.24 - lines 8-10).
- Delete/modify the attribute name of the *sync* regions, as shown in the third *sync* region (Listing 5.24 - lines 12-14).
- Delete/modify the whole end line of *sync* regions as shown in the last *sync* region (Listing 5.24 - lines 16-18).

Changing Model Properties. If the model is changed in such a way that IDs or attribute names change, while already transformed files with that respective information exist, those regions will become invalid. This would be an inconvenience to the user.

Situational Differences Between how Code Works in Duplicated *Sync* regions. If the same region (and therefore the same code) is used in multiple positions within a component, the user may accidentally write code that is not correct in all places simultaneously. For example, different

```
public class TemperatureController {
    public int execute(int temperature, int targetTemperature) {


            return temperature − targetTemperature;
        //endSync

        //sync , behaviour
            return temperature − targetTemperature;
        //endSync

        //sync _bfpnFUbFEeqXnfGWlV2_8A,
            return temperature − targetTemperature;
        //endSync

        //sync _bfpnFUbFEeqXnfGWlV2_8A, behaviour
            return temperature − targetTemperature;

    }
}
```

Listing 5.24: An examples of corrupted *sync* regions for *TemperatureController* class

situations may have different lexical scopes (i.e. symbols, variables, class names, etc.) meaning code within a region might work in one position but not in another.

Machine-readable Attribute Names. A user must rely on EGL to provide the ID to each component's attribute. This means that it must be within the EGL template first, even if it is then copied many times. A user with knowledge of the component would be unable to simply write the component name and any other human-readable information (e.g. a readable ID) to identify the attribute. It may be possible that instead of the unique ID of an attribute, names separated by dots are used. For example, the "BoilerActuator.execute.behaviour". This would also allow use of *sync* regions without first declaring them in the EGL template.

5.4 Discussion

The previous sections have described how M2T languages can be extended with *sync* regions to achieve consistency between source models and generated artefacts. This section discusses the limitations of the proposed approach.

5.4.1 Applicability and Limitations

The applicability of the proposed approach and its main limitations are now discussed.

Single Input Model. As discussed in Section 5.2, the proposed approach expects two inputs: One model and one folder, which contains all the generated artefacts. To support additional input models, the format of sync regions needs to be extended to also record (an identifier of) the model from which the respective element originates.

Target Language Support Comment. In order for this approach to be applicable, the target language needs to support comments. For example, if the M2T is generating a JSON file, this approach would not be applicable because JSON does not support comments. Presently, Java-style comments (and any other language that uses a similar comment structure to Java, like HTML, Python, Ruby, etc.) are supported.

Anticipate Location in Advance. Using the *sync* regions technique requires developers to anticipate in advance which part of the code they want to extend. For example, in Listing 5.2, there is a *sync* region inside the body of the method. If the developers want to make changes, they only can do it inside the method body.

Model Element Identities. As discussed in Section 5.2, *sync* regions approach requires model elements to have unique, persistent, and immutable identities, as these are used to trace *sync* regions back to the model elements of interest. The majority of modelling tools support such identities (e.g. XMI-IDs in EMF, GUIDs in PTC Integrity Modeller); however, there are also tools, such as Matlab Simulink, where exposed model element IDs are path-based and can change when elements are moved/renamed in a model, and where, as a consequence, the proposed technique is not applicable.

Meta-model and Model Pollution. Each *sync* region requires a respective attribute in the metamodel. As such, the more *sync* regions that are introduced, the bigger the metamodel; the models that conform to it will be polluted with implementation-level information. With reference to the running example, as long as the *behaviour* of a component fits within the body of the execute() method, then extending the metamodel with a *behaviour* attribute is a reasonable compromise. However, if changes need to be made to other parts of the component class as well (e.g. new import statements, fields, utility methods), then the metamodel and the M2T transformation need to be extended with respective attributes and *sync* regions for each such part, which can feel increasingly uncomfortable. This is an inherent issue of using this approach (i.e. not limited to this particular example) and needs to be taken into consideration before its adoption. For M2T transformations that require the generated code to be augmented in several places, other integration techniques such as inheritance/delegation may be more appropriate.

Meta-model Evolution. The generated markers for *sync* regions use the name of the attribute with which the content in the region must be synced before re-generation. If the metamodel evolves and the attribute is renamed, retyped in a breaking way (e.g. from String to Integer), or disappears altogether, the reference implementation will report an error and it will be up to the developer of the M2T transformation to rectify any inconsistent *sync* regions in previously generated files.

Embedded Code Consistency. Since code fragments embedded in *sync* regions are copied verbatim between *sync* regions and the model, changes made to the model can invalidate the embedded code fragments, making them uncompilable, or worse, inadvertently changing their semantics. For example, if the *temperatureDifference* port of the *BoilerActuator* component of Figure 3.2 is renamed to e.g. *tempDiff* in the model, then when the code is re-generated, the body of the execute method will produce compilation errors, as it will still refer to the temperature difference variable by its former name.

Direct Changes to the Embedded Code Fragments. It is important to highlight that in the proposed approach it was assumed that the code fragments should not be modified within the modelling framework. Changes are expected to be undertaken only within the IDE (i.e. the generated files). These changes are picked up by the proposed approach and propagated to the model. If changes are made within the modelling tool (i.e. directly to the model), then the changed value in the model will be treated as the “current" value and the same checks as those described in Section 5.2.3 will be performed to update or flag the region as inconsistent. If the value stored in the model is different from the one included in the generated files, then it will be updated, which might be an undesirable result.

Deleted *Sync* Regions. While the synchronisation algorithm can cope with inconsistently updated and malformed *sync* region markers, it cannot cope with *sync* regions being deleted altogether from generated files. Ideally, such missing *sync* regions should be reported to the user, but this cannot be achieved without keeping additional metadata outside the generated files, which is undesirable.

5.5 Summary

This chapter presented a novel approach that facilitates the automated synchronisation between models and textual artefacts generated from them via template-based M2T transformations. This approach includes solutions to round-trip synchronisation challenges in M2T transformations. The approach is implemented on top of an existing M2T language Epsilon Model Generation Language (EGL). Finally, an evaluation of its correctness, generalisability, and scalability has been conducted.

Chapter 6

Conclusion

This thesis investigated consistency management in model-driven engineering (MDE) in the context of model-to-text transformations (M2T). Using existing M2T transformation techniques for integrating hand-written text within auto-generated artefacts is still not desirable (as described in Section 2.3). Also, the inability for maintaining consistency when auto-generated artefacts have been changed manually. This thesis has also contributed to the research hypotheses stated in Section 3.4.1.

There are two different research hypotheses of this thesis as follows:

> *The first hypothesis of this thesis is that it is possible to integrate and preserve* ***hand-written text*** *in generated files without needing to use* ***protected regions*** *or similar constructs in Model-to-text transformations (M2Ts). The second hypothesis is that where embedding code fragments in models is necessary to achieve full code generation, the content of these fragments can be* ***automatically synchronised*** *between the model and the generated code in Model-to-text transformations (M2Ts).*

The research objectives of this thesis, as stated in Section 3.4.2 are as follows:

- Enable language-agnostic preservation of text in arbitrary locations of generated files without the need for protected regions.
- Enable automated round-trip synchronisation of code fragments embedded in generated files with the source models of the transformation.

- Assess the performance of the proposed mechanisms.

The remainder of this chapter is organised as follows. Section 6.1 presents an overview of the primary discussions in the thesis. Section 6.2 presents the main contributions to the field. Section 6.3 presents suggestions for future work.

6.1 Summary

This thesis presented two novel solutions according to the research objectives described in Section 3.4.2. **Chapter 2** provided the literature that relates to the concepts behind this research project such as Model-Driven Engineering, integrating hand-written text and round-trip engineering in model-to-text transformations. It also investigated the capabilities of many state-of-the-art mechanisms that are used in all these domains. **Chapter 3** provided the findings of the literature review and outlined the study framework, including the research problems, hypothesis, objectives, and scope. **Chapter 4** presented the design and implementation of the first proposed approach "- that of automated line based merging -" that facilitates the process of integrating hand-written text into the generated files in M2T transformations. It also presented the evaluation of the proposed approach, its limitations, and the alternative solutions. **Chapter 5** presented the design and implementation of the second proposed approach "- that of synchronised regions -" that supports round-trip engineering to facilitate the automated synchronisation between models and the textual artefacts generated from them, via template based M2T transformations. It also presented the evaluation of the proposed approach and its limitations.

6.2 Thesis Contributions

This section presents the contributions of this research project.

Automated Line Based Merging

In Chapter 4, an automated line-based merging technique was proposed. It facilitates the process of embedding hand-written text into the generated files of M2T transformations. The approach includes solutions for integrating hand-written text within generated artefacts in model-to-text transformations (M2T).

Moreover, this approach had been implemented on top of the existing M2T language Epsilon Generation language (EGL). The proposed approach was evaluated for its correctness and performance.

Synchronised Regions

In Chapter 5, synchronised regions technique was proposed. This technique facilitates the automated synchronisation between models and textual artefacts generated from them, via template-based M2T transformations. This approach includes solutions to round-trip synchronisation challenges in Model-to-Text Transformations.

The approach has been implemented on top of an existing M2T language Epsilon Model Generation Language (EGL). Also, the evaluation of its correctness, generalisability, and scalability has been conducted.

6.3 Future Work

In this section, suggestions for future work on the proposed approaches are presented. There are items for future work both related to practical implementation and conceptual improvement. We use (PI) for Practical Implementation and (CI) for conceptual improvement in the discussion below.

Automated Line Based Merging Approach

More intelligent conflict resolution (CI). Currently, the automated line-based merging approach is unable to determine which lines underwent changes, since the user is also able to add new lines. This limitation leaves the user without any support; mitigated only by any backup copies they may have taken themselves prior to making their code modifications. Original lines that have been offset by deletions appear to the algorithm as new lines. This is because the algorithm is not intelligent enough to search forward through the hashes to check whether there are later matches. Searching forward through hashes is a non-trivial approach for which there are existing intelligent algorithms better suited to the task. The process would be just as complex as automated merging and conflict resolution, and would require a complete copy of the original file, not just the hashes. One approach is to try every combination of deletion and addition with respect to the expected hashes, but from a computational perspective this is expensive, even for relatively small files.

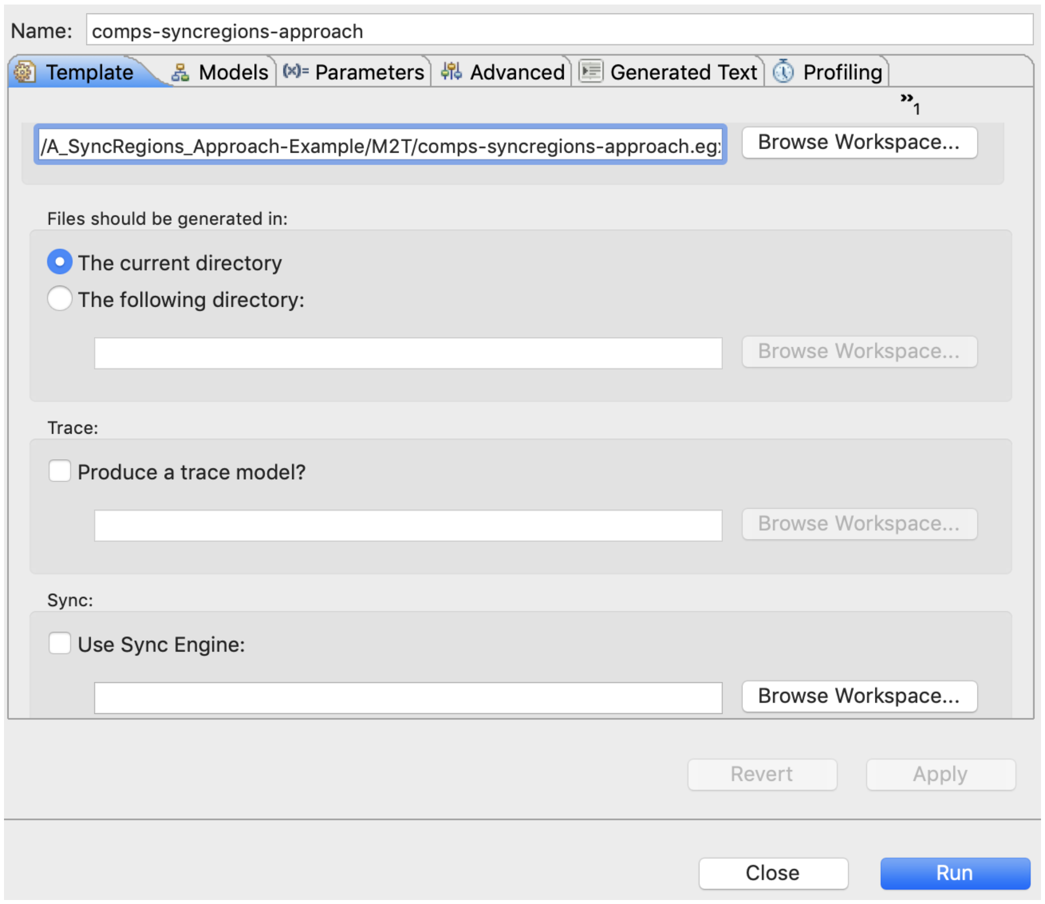
Clean the Implementation of the Merging Approach (PI). As described in Section 4.2, the merging approach has been implemented directly by modifying the EGL source code. It could instead be in its own software package which could be installed alongside existing plugins of Epsilon.

Length of Hashes (CI). As discussed in Section 4.2, the merging approach has been implemented using a hash length of four characters. Four-character hashes become unsuitable for uninterrupted regions of one thousand lines (≈3% collision probability), but five-character hashes are still quite acceptable in this case (≈0.05%). Future work could involve changing the length of hashes from four characters to five or more characters.

Intelligent history (CI). As seen in Section 4.4, the proposed approach does not have the capability to provide a history of code changes. This is not a primary concern of the proposed solution, which does not keep a full copy of previous (original) changes to start with. As future work, it would be interesting to extend traceability in EGL to record the history of transformations and allow users to have the ability to find, compare, and restore previous changes.

Synchronised Regions Approach

Evaluation against other developed prototypes (PI). The synchronised regions approach was evaluated using a minimal component-connector domain-specific language (DSL), with different aspects such as correctness, generalisability, and performance. It would be interesting to assess the applicability of the proposed approach with larger and more complex evaluation. This could use a developed prototype to re-implement existing M2T transformations that produce code that needs to be manually extended (e.g. EMF's built-in code generator that produces Java code from Ecore metamodels.


Name: comps-syncregions-approach

/A_SyncRegions_Approach-Example/M2T/comps-syncregions-approach.eg

Files should be generated in:

[✓ The current directory]

The following directory:

Trace:

Produce a trace model?

Sync:

Use Sync Engine:

Figure 6.1: An example of a sync engine.

Extend Other M2T Transformation Languages (PI). As seen in Section 2.2.1, there are multiple M2T transformation languages (e.g., Acceleo, MOFScript, XPand). It might be valuable to also extend these languages with *sync* regions.

Organizing/Simplicity of *Sync* Engine (PI). As described in Section 5.2, some parts of the implementation of the *Sync* engine have been embedded within the source code of EGL. As future work, it might be interesting for these parts to be separated in their own packages. Furthermore, the ability for the users to engage or disengage the *Sync* engine whenever they see fit, as an optional feature of transformation, would be desirable. This could be achieved by adding a new option to the EGL run configuration interface (see Figure 6.1).

Multiple Models (CI). As discussed in Section 5.4, to support additional input models, the format of sync regions needs to be extended to also record (an identifier of) the model from which the respective element originates. The user should be able to simply specify multiple models as they are currently able in EGL to do. It would be good if *Sync* engine able to support multiple models and this has been left as future work.

Model Element Identities (CI). The proposed approach requires model elements to have unique, persistent, and immutable identities, as these are used to trace *sync* regions back to the model elements of interest. There are tools, such as Matlab Simulink, where exposed model element IDs are path-based and can change when elements are moved/renamed in a model. Investigation into how to incorporate this style of identity into the proposed approach could be undertaken.

Meta-model Evolution (CI). As discussed in Section 5.2, the generated markers for *sync* regions use attribute names. The evolution of a metamodel can change the name and type of an attribute, which would prevent the proposed approach from functioning. As future work, it might be possible to detect and report such issues.

Direct Changes to the Embedded Code Fragments (CI). As discussed in Section 5.2, the proposed approach warns the user that they should not modify code fragments within the modelling framework, as it will cause inconsistencies in transformation. Future work could be undertaken to actively prevent or control these changes, such as retaining copies of models to compare changes upon transformation.

Deleted *Sync* Regions (PI). As discussed in Section 5.2, the proposed approach cannot handle regions that have been deleted from generated files. Future work could leverage access to the original templates to compare *sync* regions markers in the templates to *sync* region markers in the generated files.

Bibliography

[1] Velocity, the apache jakarta project, the apache software foundation. available : https://velocity.apache.org/engine/1.7/. [accessed 20 dec 2022].

[2] Xdoclet—attribute oriented programming. available : http://xdoclet.sourceforge.net/xdoclet/index.html. [accessed 20 dec 2022].

[3] "using egl as a server-side scripting language in tomcat," [online]. available : https://www.eclipse.org/epsilon/doc/articles/egl-server-side/. [accessed 20 dec 2022].

[4] Abbas Abdulhameed, Ahmed Hammad, Hassan Mountassir, and Bruno Tatibouet. An approach combining simulation and verification for sysml using systemc and uppaal. In *CAL 2014, 8ème conférence francophone sur les architectures logicielles*, pages 9–pages, 2014.

[5] Jacky Akoka and Isabelle Comyn-Wattiau. Roundtrip engineering of nosql databases. *Enterprise Modelling and Information Systems Architectures*, 13:281–292, 2018.

[6] Vander Alves, Pedro Matos, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. In *Transactions on aspect-oriented software development IV*, pages 117–142. Springer, 2007.

[7] László Angyal, László Lengyel, and Hassan Charaf. A synchronizing technique for syntactic model-code round-trip engineering. In *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ecbs 2008)*, pages 463–472. IEEE, 2008.

[8] Anthony Anjorin, Marius Paul Lauder, Michael Schlereth, and Andy Schürr. Support for bidirectional model-to-text transformations. *Electronic Communications of the EASST*, 36, 2011.

[9] Michał Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *International Conference on Model Driven Engineering Languages and Systems*, pages 692–706. Springer, 2006.

[10] Caroline M Ashworth. Structured systems analysis and design method (ssadm). *Information and Software Technology*, 30(3):153–163, 1988.

[11] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *IEEE software*, 20(5):36–41, 2003.

[12] D. E. Avison. Merise: A european methodology for developing information systems. *European Journal of Information Systems*, 1(3):183–191, 1991.

[13] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, (11):1257–1268, 1985.

[14] Tim Berners-Lee and Daniel Connolly. Hypertext markup language: A representation of textual information and metainformation for retrieval and interchange. *URL: http://info. cern. ch/hypertext/WWW/-MarkUp/HTML. html*, 1993.

[15] Lorenzo Bettini, Viviana Bono, and Erica Turin. I-java: an extension of java with incomplete objects and object composition. In *International Conference on Software Composition*, pages 27–44. Springer, 2009.

[16] Jean Bézivin. Model driven engineering: An emerging technical space. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 36–64. Springer, 2005.

[17] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280. IEEE, 2001.

[18] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture*, pages 33–46. Springer, 2004.

[19] Grady Booch. *The unified modeling language user guide*. Pearson Education India, 2005.

[20] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[21] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.

[22] Thomas Buchmann and Bernhard Westfechtel. Towards incremental round-trip engineering using model transformations. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, pages 130–133. IEEE, 2013.

[23] Nicolás Buezas, Esther Guerra, Juan de Lara, Javier Martín, Miguel Monforte, Fiorella Mori, Eva Ogallar, Oscar Pérez, and Jesús Sánchez Cuadrado. Umbra designer: Graphical modelling for telephony services. In *European Conference on Modelling Foundations and Applications*, pages 179–191. Springer, 2013.

[24] Loli Burgueno. Testing m2m/m2t/t2m transformations. In *SRC@ MoDELS*, pages 7–12, 2015.

[25] Loli Burgueño, Jordi Cabot, Shuai Li, and Sébastien Gérard. A generic lstm neural network architecture to infer heterogeneous model transformations. *Software and Systems Modeling*, 21(1):139–156, 2022.

[26] Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. Static fault localization in model transformations. *IEEE Transactions on Software Engineering*, 41(5):490–506, 2015.

[27] Juan Cadavid, Benoît Combemale, and Benoit Baudry. *Ten years of Meta-Object Facility: an analysis of metamodeling practices*. PhD thesis, INRIA, 2012.

[28] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[29] Joanna Chimiak_Opoka, Michael Felderer, Chris Lenz, and Christian Lange. Querying uml models using ocl and prolog: A performance study. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 81–88. IEEE, 2008.

[30] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Jtl: a bidirectional and change propagating transformation language. In *International Conference on Software Language Engineering*, pages 183–202. Springer, 2010.

[31] Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. Round-trip support for extra-functional property management in model-driven engineering of embedded systems. *Information and Software Technology*, 55(6):1085–1100, 2013.

[32] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[33] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[34] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[35] Christian Heide Damm, Klaus Marius Hansen, Michael Thomsen, and Michael Tyrsted. Tool integration: experiences and issues in using xmi and component technology. In *Proceedings 33rd International Conference on Technology of Object-Oriented Languages and Systems TOOLS 33*, pages 94–107. IEEE, 2000.

[36] Christopher John Date. *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[37] Serge Demeyer, Stéphane Ducasse, and Er Tichelaar. Why famix and not uml? uml shortcomings for coping with round-trip engineering. In *In Proceedings of« UML'99», Fort Collins*. Citeseer, 1999.

[38] Anna DEREZIŃSKA and Karol REDOSZ. Reuse of project code in model to code transformation. *Information Systems Architecture and Technology*, page 79.

[39] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Dealing with the coupled evolution of metamodels and model-to-text transformations. In *Me@ models*, pages 22–31, 2014.

[40] Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.

[41] Maged Elaasar and Lionel Briand. An overview of uml consistency management. *Carleton University, Canada, Technical Report SCE-04-18*, 2004.

[42] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. A generic approach for automatic model composition. In *International Conference on Model Driven Engineering Languages and Systems*, pages 7–15. Springer, 2007.

[43] Eclipse Foundation. Java emitter templates (jet2), Oct 2019.

[44] The Eclipse Foundation. Acceleo. [online], november 2019. available : `https://www.eclipse.org/acceleo/`. [accessed 20 dec 2022].

[45] The Eclipse Foundation. Xpand. [online], may 2016. available : `https://www.eclipse.org/modeling/m2t/?project=xpand`. [accessed 20 dec 2022].

[46] The Eclipse Foundation. Xtext. [online], march 2021. available : `https://www.eclipse.org/Xtext/`. [accessed 20 dec 2022].

[47] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[48] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[49] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[50] Lars Marius Garshol. Bnf and ebnf: What are they and how do they work. *acedida pela última vez em*, 16, 2003.

[51] Jose M Gascuena, Elena Navarro, Patricia Fernández-Sotos, Antonio Fernández-Caballero, and Juan Pavón. Idk and icaro to develop multi-agent systems in support of ambient intelligence. *Journal of Intelligent & Fuzzy Systems*, 28(1):3–15, 2015.

[52] Joseph D Gradecki and Jim Cole. *Mastering Apache Velocity*. John Wiley & Sons, 2003.

[53] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, et al. A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 74–85. IEEE, 2015.

[54] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, et al. Integration of handwritten and generated object-oriented code. In *International Conference on Model-Driven Engineering and Software Development*, pages 112–132. Springer, 2015.

[55] Timothy J Grose, Gary C Doney, and Stephen A Brodsky. *Mastering Xmi: Java Programming with Xmi, XML and UML*, volume 21. John Wiley & Sons, 2002.

[56] Object Management Group. "meta object facility (mof) core specification," online, 2014, available : `http://www.omg.org/mof/`. [accessed 20 dec 2022].

[57] Object Management Group. "object constraint language,".

[58] Object Management Group. "unified modeling language,". available : `http://www.omg.org/spec/UML/`. [accessed 20 dec 2022].

[59] Arno Haase, Markus Völter, Sven Efftinge, and Bernd Kolb. Introduction to openarchitectureware 4.1. 2. In *MDD Tool Implementers Forum*, 2007.

[60] Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM systems journal*, 45(3):451–461, 2006.

[61] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 114–129. Springer, 2009.

[62] Jack Herrington. *Code generation in action.* Manning Publications Co., 2003.

[63] Thomas Hettel, Michael Lawley, and Kerry Raymond. Model synchronisation: Definitions for round-trip engineering. In *International Conference on Theory and Practice of Model Transformations*, pages 31–45. Springer, 2008.

[64] Thomas Hettel, Michael Lawley, and Kerry Raymond. Model synchronisation: Definitions for round-trip engineering. In *International Conference on Theory and Practice of Model Transformations*, pages 31–45. Springer, 2008.

[65] Mark Hills, Paul Klint, and Jurgen J Vinju. Static, lightweight includes resolution for php. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 503–514, 2014.

[66] G. Hinkel. ".net modelling framework (nmf) repository," [online]. available : `https://github.com/NMFCode/NMF.` [accessed 20 dec 2022].

[67] Aram Hovsepyan and Dimitri Van Landuyt. Prototizer: Agile on steroids. In *FlexMDE@ MoDELS*, pages 51–60, 2015.

[68] IBM. Gigacalculator, 2022. [online]. available : `https://www.gigacalculator.com/calculators/permutation-calculator.php#howtocalculate.` [accessed 10 nov 2022].

[69] IBM. Ibmrhapsody, 2016. [online]. available : `http://www-03.ibm.com/software/products/en/ratidoor.` [accessed 20 dec 2022].

[70] IEEE. Ieee 1076-2008: Vhdl language reference manual. standard, institute of electrical and electronics engineers, 2008.

[71] Knowledge Based Systems Inc. Idef: Integrated definition methods. [online], 1980. available : `https://www.idef.com.` `[Accessed23July2021].`

[72] Sebastien Jeanmart, Yann-Gael Gueheneuc, Houari Sahraoui, and Naji Habra. Impact of the visitor pattern on program comprehension and maintenance. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 69–78. IEEE, 2009.

[73] JMerge. Jmerge. [online]. available : `http://wiki.eclipse.org/JET_FAQ_What_is_JMerge%3F.` [accessed 28 june 2022].

[74] Frédéric Jouault and Jean Bezıvin. Using atl for checking models. In *Proc. International Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia (September 2005)*. Citeseer, 2005.

[75] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. in satellite events at the models 2005 conference. *Springer*, 43:45, 2006.

[76] Nafiseh Kahani and James R Cordy. Comparison and evaluation of model transformation tools. *Queen's University, Kingston, Tech. Rep.*, 2015.

[77] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *European Conference on Object-oriented Programming*, pages 501–525. Springer, 2006.

[78] Pasi Kellokoski. Round-trip engineering. Master's thesis, 2000.

[79] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation.* John Wiley & Sons, 2008.

[80] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[81] Benjamin Klatt. Xpand: A closer look at the model2text transformation language. *Language*, 10(16):2008, 2007.

[82] Thomas Klein. *From UML to Java and back again.* Univ.-GH-Paderborn, Fachbereich Mathematik/Informatik, 2000.

[83] Shekoufeh Kolahdouz-Rahimi, Kevin Lano, Suresh Pillay, Javier Troya, and Pieter Van Gorp. Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming*, 85:5–40, 2014.

[84] Dimitrios Kolovos. *An extensible platform for specification of integrated languages for model management.* PhD thesis, University of York, 2008.

[85] Dimitrios Kolovos, Louis Rose, Richard Paige, and A Garcıa-Domınguez. The epsilon book. 178:1–10, 2010.

[86] Dimitrios S Kolovos. Establishing correspondences between models with the epsilon comparison language. In *European conference on model driven architecture-foundations and applications*, pages 146–157. Springer, 2009.

[87] Dimitrios S Kolovos, Jordi Cabot, F Bordeleau, J Bruel, and J Dingel. Towards a corpus of use-cases for model-driven engineering courses. In *EduSymp/OSS4MDE@ MoDELS*, pages 14–18, 2016.

[88] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon object language (eol). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer, 2006.

[89] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. On the evolution of ocl for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis*, pages 204–218. Springer, 2009.

[90] Dimitrios S Kolovos, Louis M Rose, Saad Bin Abid, Richard F Paige, Fiona AC Polack, and Goetz Botterweck. Taming emf and gmf using model transformation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 211–225. Springer, 2010.

[91] Dimitris S Kolovos and Richard F Paige. The epsilon pattern language. In *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*, pages 54–60. IEEE, 2017.

[92] Ivan Kurtev, Klaas Van Den Berg, and Frédéric Jouault. Evaluation of rule-based modularization in model transformation languages illustrated with atl. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1202–1209, 2006.

[93] Christian Lange, MRV Chaudron, Johan Muskens, LJ Somers, and HM Dortmans. An empirical investigation in quantifying inconsistency and incompleteness of uml designs. In *Workshop consistency problems in uml-based software development II*, pages 26–34, 2003.

[94] Juan de Lara and Hans Vangheluwe. Atom 3: A tool for multi-formalism and meta-modelling. In *International Conference on Fundamental Approaches to Software Engineering*, pages 174–188. Springer, 2002.

[95] Rafael Andrés Leaño Gutiérrez et al. Using change intentions to guide evolution and versioning in model driven software product lines. Master's thesis, Bogotá-Uniandes, 2009.

[96] Matthias Lenk, Arnd Vitzthum, and Bernhard Jung. Non-simultaneous round-trip engineering for 3d applications. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2012.

[97] Olaf Leßenich and Christian Lengauer. *Adjustable Syntactic Merge of Java Programs.* PhD thesis, MA thesis. Department of Informatics and Mathematics, University of Passau . . . , 2012.

[98] Sina Madani. *Parallel and Distributed Execution of Model Management Programs.* PhD thesis, University of York, 2020.

[99] Sina Madani, Dimitrios S Kolovos, and Richard F Paige. Parallel model validation with epsilon. In *European Conference on Modelling Foundations and Applications*, pages 115–131. Springer, 2018.

[100] Hussein M Marah, Raheleh Eslampanah, and Moharram Challenger. Dsml4tinyos: Code generation for wireless devices. In *MODELS Workshops*, pages 509–514, 2018.

[101] Tom Mens. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, 28(5):449–462, 2002.

[102] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic notes in theoretical computer science*, 152:125–142, 2006.

[103] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A Fernandez. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical software engineering*, 18(1):89–116, 2013.

[104] Leckraj Nagowah, Zarah Goolfee, and Chris Bergue. Rtet-a round trip engineering tool. In *2013 International Conference of Information and Communication Technology (ICoICT)*, pages 381–387. IEEE, 2013.

[105] Stefan Naujokat. *Heavy meta: model-driven domain-specific generation of generative domain-specific modeling tools*. PhD thesis, 2017.

[106] Babajide Ogunyomi, Louis M Rose, and Dimitrios S Kolovos. Incremental execution of model-to-text transformations using property access traces. *Software & Systems Modeling*, pages 1–17, 2018.

[107] Babajide J Ogunyomi. *Incremental Model-to-Text Transformation*. PhD thesis, University of York, 2016.

[108] Jon Oldevik, Tor Neple, and Jan Øyvind Aagedal. Model abstraction versus model to text transformation. *Computer Science at Kent*, page 188, 2004.

[109] OMG. Business process model and notation (bpmn) version 2.0. specification, object management group, year = 2012. available : `https://www.omg.org/spec/BPMN/2.0/`.

[110] OMG. Systems modeling language (sysml) v.1.6. specification, object management group, 2019. available : `https://www.omg.org/spec/SysML`.

[111] OMG. Unified modeling language (uml). specification, object management group, 2017. available : `https://www.omg.org/spec/UML`.

[112] Oracle. Java oracle. online., 2016.

[113] Richard F Paige, Dimitrios S Kolovos, Louis M Rose, Nicholas Drivalos, and Fiona AC Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 162–171. IEEE, 2009.

[114] Terence Parr. StringTemplate, 2013.

[115] Francisco Pérez Andrés, Juan de Lara, and Esther Guerra. Domain specific languages with graphical and textual views. In *International Symposium on Applications of Graph Transformations with Industrial Relevance*, pages 82–97. Springer, 2007.

[116] Van Cam Pham. *Model-Based Software Engineering: Methodologies for Model-Code Synchronization in Reactive System Development*. PhD thesis, Université Paris-Saclay (ComUE), 2018.

[117] John D Poole. Model-driven architecture: Vision, standards and emerging technologies. In *Workshop on Metamodeling and Adaptive Object Models, ECOOP*, volume 50, 2001.

[118] Ernesto Posse. Papyrusrt: modelling and code generation. In *Workshop on Open Source for Model Driven Engineering (OSS4MDE'15)*, 2015.

[119] Mark Richters and Martin Gogolla. On formalizing the uml object constraint language ocl. In *International conference on conceptual modeling*, pages 449–464. Springer, 1998.

[120] Markus Riedl-Ehrenleitner, Andreas Demuth, and Alexander Egyed. Towards model-and-code consistency checking. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 85–90. IEEE, 2014.

[121] Thiago Rocha Silva, Marco Winckler, and Hallvard Trætteberg. Ensuring the consistency between user requirements and task models: A behavior-based automated approach. *Proceedings of the ACM on Human-Computer Interaction*, 4(EICS):1–32, 2020.

[122] Louis M Rose, Nicholas Matragkas, Dimitrios S Kolovos, and Richard F Paige. A feature model for model-to-text transformation languages. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pages 57–63. IEEE Press, 2012.

[123] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. The epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 1–16. Springer, 2008.

[124] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. The epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 1–16. Springer, 2008.

[125] N Rouquette, Tracy Neilson, and George Chen. The 13th technology of deep space one. 1999.

[126] Beatriz Angelica Sanchez Pina. *Conservative and traceable executions of heterogeneous model management workflows*. PhD thesis, University of York, 2021.

[127] Stefan Sarstedt. Model-driven development with activecharts-tutorial. 2006.

[128] Guido Scherp. *A framework for model-driven scientific workflow engineering*. BoD–Books on Demand, 2013.

[129] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. PhD thesis, Dissertation, Techn. Hochsch., 2011, 2012.

[130] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[131] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[132] Bran Selic. What will it take? a view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.

[133] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003.

[134] Shane Sendall and Jochen Küster. Taming model round-trip engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, volume 1. Citeseer, 2004.

[135] Thiago Rocha Silva and Marco Winckler. A scenario-based approach for checking consistency in user interface design artifacts. In *proceedings of the XVI Brazilian symposium on human factors in computing systems*, pages 1–10, 2017.

[136] SparxSystems. Enterprise architect. [online]. available : `http://www.sparxsystems.eu/start/home/`. [accessed 20 dec 2022].

[137] J Michael Spivey. An introduction to z and formal specifications. *Software Engineering Journal*, 4(1):40–50, 1989.

[138] John Spriggs. *GSN-the goal structuring notation: A structured approach to presenting arguments.* Springer Science & Business Media, 2012.

[139] Lynn Andrea Stein. Delegation is inheritance. *ACM SIGPLAN Notices*, 22(12):138–146, 1987.

[140] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework.* Pearson Education, 2008.

[141] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009.

[142] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures*, 52:43–62, 2018.

[143] Masoumeh Taromirad. *A Modelling Approach to Multi-Domain Traceability.* PhD thesis, Enterprise Systems Research Group, Department of Computer Science . . . , 2014.

[144] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 295–304, 2007.

[145] Javier Troya, Sergio Segura, and Antonio Ruiz-Cortés. Automated inference of likely metamorphic relations for model transformations. *Journal of Systems and Software*, 136:188–208, 2018.

[146] D Ungar and RB Smith. Self: The power of simplicity. inobject-oriented programming systems, languages, and applications, 1987.

[147] Ellen Van Paesschen, Wolfgang De Meuter, and Maja D'Hondt. Selfsync: a dynamic round-trip engineering environment. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 190–191, 2005.

[148] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of dsm graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pages 233–238. IEEE, 2014.

[149] John Vlissides and Andrei Alexandrescu. To code or not to code, part i. *C++ Report*, 2000.

[150] M Völter and J Bettin. Patterns for model-driven software-development, version 1.4, may 2004, 2015.

[151] Markus Völter. A catalog of patterns for program generation. In *EuroPLoP*, pages 285–320, 2003.

[152] Markus Völter. Best practices for dsls and model-driven development. *Journal of Object Technology*, 8(6):79–102, 2009.

[153] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management.* John Wiley & Sons, 2013.

[154] Guido Wachsmuth. A formal way from text to code templates. In *International Conference on Fundamental Approaches to Software Engineering*, pages 109–123. Springer, 2009.

[155] Jos Warmer. A model driven software factory using domain specific languages. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 194–203. Springer, 2007.

[156] Stefan Warwas, Christian Hahn, and Klaus Fischer. A visual development environment for jade. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 1349–1350. Citeseer, 2009.

[157] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32, 2019.

[158] James R Williams. *A novel representation for search-based model-driven engineering.* PhD thesis, University of York, 2013.

[159] James R Williams, Dimitrios S Kolovos, Fiona AC Polack, and Richard F Paige. Requirements for a model comparison language. In *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, pages 26–29, 2011.

[160] Athanasios Zolotas. *Type inference in flexible model-driven engineering.* PhD thesis, University of York, 2016.

[161] " "EMF Ecore Javadoc. Eclipse software foundation, [online]. available : `https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html.` [accessed 20 dec 2022].