Conservative and Traceable Executions of Heterogeneous Model Management Workflows

Beatriz Angelica Sanchez Pina

Doctor of Philosophy

University of York
Computer Science

September 2021

Abstract

One challenge of developing large scale systems is knowing how artefacts are interrelated across tools and languages, especially when traceability is mandated e.g., by certifying authorities. Another challenge is the interoperability of all required tools to allow the software to be built, tested, and deployed efficiently as it evolves. Build systems have grown in popularity as they facilitate these activities. To cope with the complexities of the development process, engineers can adopt model-driven practices that allow them to raise the system abstraction level by modelling its domain, therefore, reducing the accidental complexity that comes from e.g., writing boilerplate code. However, model-driven practices come with challenges such as integrating heterogeneous model management tasks e.g., validation, and modelling technologies e.g., Simulink (a proprietary modelling environment for dynamic systems). While there are tools that support the execution of model-driven workflows, some support only specific modelling technologies, lack the generation of traceability information, or do not offer the cutting-edge features of build systems like conservative executions i.e., where only tasks affected by changes to resources are executed. In this work we propose ModelFlow, a workflow language and interpreter able to specify and execute model management workflows conservatively and produce traceability information as a side product. In addition, ModelFlow reduces the overhead of model loading and disposal operations by allowing model management tasks to share already loaded models during the workflow execution. Our evaluation shows that ModelFlow can perform conservative executions which can improve the performance times in some scenarios. ModelFlow is designed to support the execution of model management tasks targeting various modelling frameworks and can be used in conjunction with models from heterogeneous technologies. In addition to EMF models, ModelFlow can also handle Simulink models through a driver developed in the context of this thesis which was used to support one case study.

List of Contents

Abstract 3

List of Contents 6

List of Tables 7

List of Figures 9

List of Listings 12

List of Algorithms 13

Acknowledgments 15

Author Declaration 16

1 Introduction 17
- 1.1 Research results 20
- 1.2 Thesis structure 22

2 Background 23
- 2.1 Model-Driven Engineering 23
  - 2.1.1 Models and metamodels 23
  - 2.1.2 Modeling languages 25
  - 2.1.3 Model management operations 27
  - 2.1.4 Global model management 30
  - 2.1.5 Model management platforms 32
  - 2.1.6 MDE challenges 34
  - 2.1.7 Scalability 34
  - 2.1.8 Co-evolution 39
  - 2.1.9 Heterogeneity and interoperability 40
- 2.2 Traceability 41
  - 2.2.1 Defining traceability 41
  - 2.2.2 Demanding traceability 43
  - 2.2.3 The challenges of traceability 44
  - 2.2.4 Traces in MDE 45

2.2.5 Tools 45
2.3 Automation of task processes 48
2.3.1 Business processes 48
2.3.2 Model management workflows 49
2.3.3 Build systems 52
2.3.4 Continuous integration 58
2.4 Summary 59

3 Analysis and hypothesis 60
3.1 Analysis 60
3.2 Research overview 65
3.2.1 Hypothesis 65
3.2.2 Objectives 65
3.2.3 Scope 66

4 ModelFlow: A model management workflow framework 69
4.1 ModelFlow's features 69
4.1.1 Declarative workflow 69
4.1.2 Conservative executions 70
4.1.3 Automated model management 70
4.1.4 Model management traces 70
4.2 Architecture 71
4.2.1 Components 72
4.3 Language 73
4.3.1 Abstract syntax 74
4.3.2 Concrete syntax 76
4.3.3 Workflow metamodel 78
4.3.4 Semantics 80
4.4 System design 81
4.4.1 Knowing when to execute 81
4.4.2 From declarations to runnable entities 85
4.4.3 Conservative task executions 90
4.4.4 Model management traces 98
4.5 Implementation 106
4.5.1 Decisions 106
4.5.2 Plugins 107
4.6 Summary 108

5 Supporting heterogeneous models: MATLAB/Simulink 110
5.1 Background 112
5.2 Integration with Epsilon 117
5.2.1 Simulink models 118

5.2.2 Collection query optimisation 125
5.3 Evaluation 128
5.3.1 Experiment on Simulink models 128
5.3.2 Experiment on collection queries 137
5.4 Observations and lessons learned 145
5.5 Related work 147
5.6 Integration with ModelFlow 149

6 Evaluation 151
6.1 Case study: Component workflow 151
6.1.1 Background 152
6.1.2 Experimental setup 155
6.1.3 Results 157
6.1.4 Discussion 159
6.1.5 Threats to validity 160
6.2 Case study: EuGENia 160
6.2.1 Background 161
6.2.2 Approach 164
6.2.3 Setup 171
6.2.4 Correctness results 176
6.2.5 Performance results 180
6.2.6 Discussion 184
6.2.7 Threats to validity 185
6.3 Case study: Industrial workflow 185
6.3.1 Background 186
6.3.2 Approach 187
6.3.3 Results 193
6.3.4 Discussion 194
6.4 Extensibility 199
6.5 Interoperability 199
6.5.1 Eclipse 199
6.5.2 Build tools 199
6.6 Summary 201

7 Conclusion 202
7.1 Summary 202
7.2 Thesis contributions 203
7.2.1 Novel tools and techniques 203
7.2.2 Notable additional results 205
7.3 Future work 206

8 Bibliography 212

List of Tables

2.1 Classification of existing build systems. 54

5.1 Evaluated invariants 131
5.2 Number of elements per type by MATLAB model file size (MB). 132
5.3 Number of elements per type on each model. 140
5.4 Mean query execution time and percentage of time spent sending commands to MATLAB and awaiting a response. 141
5.5 Performance improvement by query and model. 144

6.1 Executed tasks per scenario 177
6.2 Execution time by execution stage for `MF` 182
6.3 Generated files through EGX. 189
6.4 Number of traces extracted by ModelFlow from the execution. . 194

List of Figures

2.1 Social network sample model 24
2.2 Modeling layers in a Model-Driven Architecture 25
2.3 Exogenous Model-to-Model transformation 28
2.4 Endogenous Model-to-Model transformation 28
2.5 Epsilon architecture 33
2.6 Model indexing framework architecture 39
2.7 Example of a Hawk model index 40
2.8 Trace triplet 42
2.9 Types of traceability 43
2.10 Capra architecture 47
2.11 ChainTracker main screen 48
2.12 MTC-Flow metamodel 51
2.13 Example Workflow+ metamodel and instance 51

4.1 ModelFlow architecture 71
4.2 ModelFlow component diagram 72
4.3 Class diagram of ModelFlow's abstract syntax 75
4.4 Workflow specification metamodel 79
4.5 ModelFlow dependency graph of Listing 4.7. 84
4.6 ModelFlow execution graph of Listing 4.7. 85
4.7 Class diagram of classes involved in the task instantiation process 87
4.8 Class diagram of classes needed to contribute plugins 88
4.9 Class diagram of the IModelResourceInstance interface 91
4.10 Execution trace metamodel 92
4.11 Class diagram of the IHasher interface 96
4.12 Sequence diagram of the process of a task's first time execution 97
4.13 Class diagram of traces returned by Epsilon languages 101
4.14 Model management trace metamodel 102
4.15 Class diagram of the model management trace builder 105

5.1 Example MATLAB/Simulink model. 112
5.2 Example of MATLAB/Stateflow model elements 113
5.3 Simulink element types in Massif's Simulink metamodel 115
5.4 Class diagram with architecture of Simulink driver 119

5.5 Model management execution process for approaches 130
5.6 Imported EMF model size vs. original MATLAB files. 132
5.7 Execution time vs. MATLAB file size per stage of validation process 133
5.8 Total execution time vs. MATLAB file size 134
5.9 Structure of generated Simulink models 138
5.10 Distribution of the query performance on the models with and without optimisations. 142
5.11 Performance of queries, with and without optimisation, against the number of elements in the models. 143

6.1 Component workflow dependency graph 152
6.2 Boiler components 153
6.3 *configuration* and *component* metamodels 153
6.4 Execution time of each scenario in milliseconds. 159
6.5 SCL editor generated with EuGENia 162
6.6 EuGENia task-resource and inter-task dependencies 163
6.7 EuGENia ModelFlow dependency graph. 172
6.8 EuGENia ModelFlow execution graph. 173
6.9 BPMN model created using the graphical editor generated with ModelFlow. 174
6.10 EuGENia model inter-dependencies. 178
6.11 EuGENia traces of ETL rule 179
6.12 EuGENia traces of EOL task 179
6.13 EuGENia traces of GmfMap2GmfGen task 180
6.14 EuGENia mean execution times per scenario and approach . . 181
6.15 Time spent in ModelFlow features per scenario 182
6.16 Core task logic execution times per scenario 183
6.17 ModelFlow dependency graph of industrial case study. 188
6.18 Step navigator view of the original EEC design tool. 195
6.19 ModleFlow run configuration 200

7.1 User Interface that supports the GMF execution process 209

List of Listings

2.1 Social network metamodel in Emfatic 23
2.2 Sample EOL program 33

4.1 Example of a task declaration. 74
4.2 Concrete syntax of a model resource declaration. 76
4.3 Model resource declaration example. 76
4.4 Concrete syntax of a task declaration. 77
4.5 Concrete syntax of <ModelCall>. 77
4.6 Task declaration example. 78
4.7 Sample workflow declaration 82
4.8 Example of task and model definition classes. 88
4.9 Example of parameter configuration in task definition. 89
4.10 Annotated input methods of an epsilon:egx task definition. . 93
4.11 EGX input and output declaration in task definition. 95
4.12 Tree metamodel 98
4.13 Tree validation in EVL 99
4.14 Graph metamodel 99
4.15 ETL transformation from Tree to Graph 99
4.16 EGL template that generates a Graphviz/Dot graph from a graph model 100
4.17 Retrieving trace from epsilon:etl task definition 103
4.18 An EOL program creating traces at runtime 105
4.19 Trace utility to capture traces through EOL programs. 105

5.1 MATLAB Simulink functions 114
5.2 MATLAB Java API 115
5.3 Collection of block names in EOL 118
5.4 MATLAB functions to collect Simulink blocks and their names. 118
5.5 Model element creation 120
5.6 Linking methods for block elements in EOL 121
5.7 Stateflow element creation in MATLAB 121
5.8 Stateflow element creation in EOL 121
5.9 Model element deletion in EOL 122
5.10 Simulink element deletion in MATLAB 122

5.11 MATLAB Simulink element getter and setters 122
5.12 Get and set Simulink element properties in EOL 122
5.13 Get and set Stateflow element properties in MATLAB and EOL 123
5.14 Retrieval of model elements in EOL 123
5.15 Retrieval of Simulink elements in MATLAB 124
5.16 Retrieval of Stateflow elements in MATLAB 124
5.17 Retrieval of Stateflow elements in EOL 124
5.18 MATLAB function structure 124
5.19 EOL method structure 124
5.20 Invocation of MATLAB functions as EOL methods 125
5.21 Sample MATLAB functions that act on Simulink elements . . . 125
5.22 EOL collection of Simulink block names 125
5.23 EOL selection of Simulink inport blocks 125
5.24 MATLAB Simulink and Stateflow collection operations 126
5.25 Simulink element selection MATLAB function 127
5.26 EOL selection of Simulink gain blocks 127
5.27 MATLAB selection of Simulink gain blocks 127
5.28 Stateflow element selection MATLAB function 127
5.29 EOL selection of Stateflow states 127
5.30 MATLAB selection of Stateflow states 127
5.31 Sample EVL script with invariant 9 from Table 5.1 129
5.32 Port dimension block property in EOL with Simulink Model Driver 130
5.33 Port dimension block property in EOL with EMF/Massif . . . 130
5.34 List of EOL queries 139
5.35 Example Simulink model declaration in ModelFlow 149
5.36 MATLAB function used to compute dependencies of a Simulink model. 150
5.37 MATLAB function used to compute the ID of Simulink and Stateflow model elements 150

6.1 Sample EVL invariants 154
6.2 Generated code of the *TemperatureComparator* component . . . 154
6.3 Gradle workflow 156
6.4 ModelFlow workflow 157
6.5 Annotated Emfatic metamodel of an SCL 161
6.6 ModelFlow EuGENia workflow parameters and variables 165
6.7 Models used in the ModelFlow EuGENia workflow 165
6.8 Ecore step tasks in the ModelFlow EuGENia workflow 166
6.9 GenModel step tasks in the ModelFlow EuGENia workflow . . 167
6.10 GMF step in ModelFlow EuGENia workflow 169
6.11 GmfGen step in ModelFlow EuGENia workflow 170

6.12 EmfCode step in ModelFlow EuGENia workflow 170
6.13 Extension fragment to polish script 173
6.14 Original EuGENia delegate execution order. 175
6.15 ModelFlow parameters for the industrial case study. 189
6.16 `EEC` and `Simulink` models in ModelFlow. 189
6.17 `generateSimulink` ETL transformation in ModelFlow. 190
6.18 Code generating multi-tasks in ModelFlow. 191
6.19 Code generating tasks from the `Simulink` model in ModelFlow. 192
6.20 HTML model group and trace extraction task in ModelFlow. . 193
6.21 Supporting model and task declaration extension 197
6.22 Dynamically model generation ModelFlow proposal with model matching in tasks 198
6.23 Nested workflow ModelFlow proposal for industrial case study . 198
6.24 Sample Maven ModelFlow plugin 200

7.1 Workflow nesting proposal 207
7.2 EOL explicit and implicit file dependencies 210

List of Algorithms

1 Execution plan construction algorithm. 86

To my dad and his never-ending admiration for technology.
To my husband for sharing this journey with me.

Acknowledgements

In the first place I would like to thank my supervisors Prof. Dimitris Kolovos and Prof. Richard Paige for their continuous support and valuable guidance along the way. They are both extraordinary supervisors, researchers, teachers, and people. I am deeply thankful for I know that thanks to them I have grown not only as a researcher and practitioner but also as a person.

Thank you to both Dr. Javier Camara and Prof. Tim Kelly for their support and guidance during internal milestones. I am also grateful to my examiners Prof. Juergen Dingel and Dr. Siyuan Ji for taking the time to review this work. Similarly, I would like to thank Dr. Konstantinos Barmpis, Dr. Simos Gerasimou, Dr. Horacio Hoyos, Dr. Alfonso de la Vega, Dr. Sina Madani, Qurat ul ain Ali and Sorour Jahanbin for taking the time participate in my Mock Viva and provide feedback on the thesis.

Thank you to academic and industrial colleagues for discussions and feedback. In particular, thanks to Dr. Simos Gerasimou, Dr. Konstantinos Barmpis, Dr. Thanos Zolotas, Justin Cooper, Caroline Brown, Jason Hampson, Dr. Mole Li, Dr. Mike Bennett, Philip Elliott, Dr. Steve Law, Dr. Stuart Hutchesson and Dr. Alan Grigg.

A big thank you to my husband, Dr. Horacio Hoyos, for fruitful discussions, advice and feedback on my work but also for his support and care.

Finally, thank you to my family and friends for their support and encouragement.

Author Declaration

This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as references. Except where stated, this thesis is a presentation of original work by the author. Parts of this thesis have been previously published by the author. The following publications have been primarily written by the PhD candidate.

[Journal] B. A. Sanchez, A. Zolotas, H. Hoyos Rodriguez, D. Kolovos, R. F. Paige, J. C. Cooper, J. Hampson, "Runtime Translation of OCL-like Statements on Simulink Models: Expanding domains and optimising queries", in *Software and Systems Modeling*, 2021.

[Conference] B. A. Sanchez, D. S. Kolovos and R. F. Paige, "To build or not to build: ModelFlow, a build solution for MDE projects", in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion Proceedings - MODELS '20*, 2020.

[Workshop] B. A. Sanchez, D. S. Kolovos and R. F. Paige, "ModelFlow: Towards Reactive Model Management Workflows", in *Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling*, 2019.

[Conference] B. A. Sanchez, A. Zolotas, H. Hoyos, D. S. Kolovos and R. F. Paige, "On-the-fly Translation and Execution of OCL-like Queries on Simulink Models", in *Proceedings of the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion Proceedings - MODELS '19*, 2019.

[Doctoral Symposium] B. A. Sanchez, "Context-aware traceability across heterogeneous modelling environments", in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion Proceedings - MODELS '18*, 2018, pp. 174–179.

The initial version of the Simulink driver was implemented by Dimitris Kolovos. The author continued its maintenance, added support for Stateflow blocks and provided optimisations for operations on collections.

This work was supported by the University of York and by the EPSRC through the National Productivity Investment Fund in partnership with Rolls-Royce under Grant EP/R512230/1.

1 Introduction

It might not come as a surprise to the reader that technology keeps evolving at an ever-increasing speed. This is the result of industry ambitions to reduce product costs and their time to market. Paradoxically, the rapid evolution of technology makes it more complex for the industry to meet these goals as new standards must be followed and more project dependencies —such as Components Off The Shelf (COTS), are introduced. Time pressures have forced development process to adopt agile practices that allow for quick defect detection and rapid adaption in case priorities change or unexpected events take place. At the same time, the increased complexity requires more domain specific experts to deal with the different dimensions of the product such as its real-time behaviour and its requirements. To cope with these challenges, many industries are now adopting Model-Driven Engineering (MDE) processes which advocate for automation and enable engineers to work with artefacts that are close to their domain of expertise.

The key idea behind MDE is to use models that capture the essential complexity of a system in a given context to make it easier to understand and manipulate. For example, a model can be used to capture the real time behaviour of the system to facilitate the work of Real Time Engineers. Traditionally, models used to be drawings of views of the systems which were used to guide the development process. However, in MDE processes, models are living entities at the core of the development process that capture information in a structured way so it can be processed. Models can be validated, compared, transformed into other type of models, and used to generate text, documents, code, and configuration artefacts. All these automated tasks reduce the time spent on manual and error prone activities. For example, generating code can reduce the number of programming errors, the time spent writing boilerplate code and the need for code reviews.

Usually, models are created, modified, and tested with dedicated tools that know how to manipulate them. Examples of such tooling include MathWorks Simulink, which manages dynamic system models, and IBM Rational DOORS, which is dedicated to requirement models. Modelling tools usually provide facilities to perform operations on the specific type of models they manage. For example, simulations can be executed for Simulink models in the MathWorks workbench and Word documents can be generated from requirement

models in DOORS. There are also generic model management tools that do not make assumptions on the type of model they act upon (e.g., requirements or dynamic system models) and are used to perform common operations such as transformations and validations on them. While running a simulation for a Simulink model is a preconfigured activity in the MathWorks workbench, running a transformation in a generic model management tool (e.g., OCL, ATL, QVT, EOL) usually requires the user to write a custom model management program. As such, MDE projects end up relying on a variety of domain specific models and model management programs that are managed by different tools. While individual models facilitate the comprehension of an aspect of the system, understanding how models are connected to each other and orchestrating their model management activities remains a challenge.

Coping with heterogeneity is one the main challenges in large MDE projects. The first challenge that it brings has to do with interoperability. Developers have proposed integrations with build tools to support the orchestration and execution of multiple and heterogeneous model management tasks with diverse persistence technologies. Build tools are almost a requirement in software projects these days and are well integrated with Continuous Integration (CI) tools. Both CI and build tools along with version management systems facilitate collaboration within and across teams, support agile processes and enable continuous testing and delivery of software. And yet, neither build tools nor CI tools are model-aware resulting in the need to adapt models and model management tasks as build tasks. Examples of such adaptations include the declaration of model loading and disposal tasks, which are normally steps configured within model management tasks as opposed to tasks on their own. Build tools and CI tools that orchestrate tasks often work with files. In contrast, model management workflows involve model management tasks that consume, modify, or produce heterogeneous models that may be backed by arbitrary technologies, including databases (e.g., PTC Integrity Modeller). Other developers have proposed dedicated model management workflow frameworks to orchestrate these tasks (e.g., MWE2, MTC-Flow). Nevertheless, these frameworks often lack some of the cutting-edge features that state-of-the-art build tools offer such as executions where only tasks that may be transitively affected by external changes to build resources are re-executed (e.g., by Gradle).

The second challenge that heterogeneity brings has to do with the provenance of generated artefacts. Aerospace and automotive are examples of industries adopting MDE in their processes and yet, as safety critical industries, they must not only produce the software that runs on their products but also certify it as safe. To do this, certifying authorities require evidence (e.g. in the form of test results and traceability) to check whether all requirements have been translated into code and if they have been successfully tested. Manually producing

traceability information, that is, a set of links (traces) that connect source and target elements through a semantically rich relationship, is not an easy task. Neither it is to produce this information when models involved in the traces are provided by different tools as they may conform to various *metamodels* and be persisted in different formats. In contrast, traceability is often a by-product of model management activities. A model management trace works as a *ledger* of the actions that occurred during the model management task execution. For example, a model-to-model transformation often returns traces of all model elements that are generated in the target model specifying the model elements in the source model that were involved in their transformation. Producing and maintaining rich traceability information from orchestrated model management tasks could support the establishment of end-to-end traceability, that is, traceability from artefacts at the early development process, such as requirements, to those at the very end, such as test results. This information can help companies provide evidence required in certification processes. At the same time it can also be used by developers to determine model coverage, debug complex workflows, perform impact analysis on potential metamodel or transformation changes, and to identify refactoring opportunities. However, while many model management tasks produce traces as a side-product of their execution, others do not produce any.

This thesis proposes an architecture that can orchestrate model management workflows that (i) support executions where only tasks that may be transitively affected by external changes to build resources are re-executed, (ii) produce model management traceability information from tasks involved in the workflow and (iii) reduce invocations of model loading while disposing models when they are no longer needed. An outcome of this project is ModelFlow, a prototype implementation of this architecture which is publicly available and has been tested with several example workflows. Our architecture supports the definition of the workflows through a declarative textual language that is based on the relationship of the models with the tasks which ensures that model-task and task-task dependencies are clearly stated. ModelFlow takes advantage of this information to schedule the task executions and to generate runtime views of these interdependencies. Another outcome of the thesis includes the implementation of a bridge between Simulink models and the languages of the Epsilon family of model management languages which offers a native integration with MATLAB as opposed to available solutions which require the models to be transformed into an open-source modelling representation format. This bridge enables ModelFlow workflows to query and modify Simulink and Stateflow models and to maintain traces that involve their model elements. This bridge has been integrated with ModelFlow, enabling the management of Simulink models within ModelFlow workflows.

1.1 Research results

This work proposes a novel architecture of a model management workflow interpreter that (a) combines features from build tools that allow for partial executions that only re-execute tasks transitively affected by external changes to workflow resources, while also (b) gathering and creating traceability information in a queryable format as a side product of its execution, and (c) using a novel approach to automatically handle models that loads them when first needed and disposes them when no longer required by the workflow. Additionally, to declare these workflows, this work has proposed a model management workflow language that differs to other frameworks in the way models are used to drive the execution and in its capability to generate tasks dynamically. To validate the architecture we have built ModelFlow which is a prototype of the model management workflow language and interpreter, which is publicly available and has been presented in [163].

These architectural features are evaluated through three case studies performed using the ModelFlow prototype. Overall, the model management workflow architecture was designed to be extensible to support multiple model and task types. The case studies demonstrate its extendibility when supporting heterogeneous tasks beyond Epsilon (e.g., GMF, EMF) and model types beyond EMF (e.g., HTML, Simulink).

The first case study presented a contrived case study of a workflow with three models and three tasks that evaluates the correctness and performance of ModelFlow executed under scenarios that change different resources used or produced by the workflow. This case study also compared the workflow language and execution in ModelFlow against those of the Gradle build tool. Overall Gradle outperformed ModelFlow in most scenarios but ModelFlow was better suited to provide fine grained responses to changes to workflow resources, particularly those affecting outputs.

The second case study reproduced a more complex workflow used by the EuGENia tool (which predates this work) to generate graphical editors from annotated metamodels. This workflow involved heterogeneous tasks from GMF and EMF. The case study evaluated the correctness and performance shown by ModelFlow compared against the original implementation and executed under realistic change scenarios. We were able to reproduce this workflow (which contained tasks that modified models) with ModelFlow and the results suggest that the performance of repetitive executions of the workflow can be significantly improved with ModelFlow, particularly when not all tasks need to be re-executed offering the ability to reduce execution times when only parts of the workflow are affected by external changes. At the same time, the results suggest that scenarios that need to be re-executed completely will inevitably

spend additional time processing inputs and outputs. Overall ModelFlow's features like execution graph generation, model loading and disposal, model management trace generation, and input and output processing took 3.3% of the total workflow execution in most of the change scenarios. In particular, tracking and comparing changes among input and output models were the most time-consuming activities (not including the core task logic e.g. the execution of a transformation program). Additionally, this case study showed that ModelFlow was able to collect and generate traces from relevant tasks without a significant impact in the performance. Furthermore, it showed that ModelFlow was also able to reduce the number of invocations of model loading and disposal procedures which is a valuable feature in workflows involving models that are time consuming to load such as Simulink models.

The third case study described a sanitised industrial workflow that involved heterogeneous models including Simulink and HTML. This qualitative study was used to validate the language and model management traceability support provided by the prototype but also to provide insights regarding the conciseness of the workflow specification, its visualisation and user interaction facilities, the recovery of model management traces, limits to conservative executions, and it also explored potential workflow language optimisations. This workflow demonstrated the utility of the dynamic task generation feature to reduce the number of tasks in the workflow declaration and the ability to collect traces from tasks in the workflow. Overall, further improvements can be made regarding user interaction (e.g. triggering only parts of the workflow), conciseness (e.g. nested workflows, abstract tasks, dynamic model generation) and visualisation support (e.g. making workflow views executable). Similarly, the case study highlighted potential conservative execution pitfalls when tasks contain implicit inputs or outputs that are not provided in the task declaration (e.g. when using a String to read a file in a model management program).

To demonstrate the extensibility of ModelFlow while also bridging the gap between proprietary modelling tools and open-source research tools, we set out to propose an integration between the Epsilon family of languages and MATLAB Simulink that could be integrated with ModelFlow. Our approach relies on the "on-the-fly" translation of OCL-like queries into MATLAB statements. As opposed existing solutions, our approach does not require an intermediate representation and can mitigate the cost of the upfront transformation on large models. We evaluated both approaches and measure the performance of a model validation process with Epsilon on a sample of large Simulink models publicly available. Our results suggest that, with our approach, the total validation time can be reduced by up to 80%. We also improved the approach to perform queries on collections of model elements more efficiently and propose an experiment that compares the performance of a set of queries on models

with different sizes. Our results suggest an improvement by up to 99% on some queries.

1.2 Thesis structure

Chapter 2 provides an overview of the background theory and current research in the domains of Model-Driven Engineering, traceability, and build systems. **Chapter 3** presents the analysis of the problem, states the research framework including the hypothesis, goals, and scope. **Chapter 4** presents the architecture and implementation of ModelFlow highlighting its main features: conservative executions, traceability, and lazy loading. **Chapter 5** presents the architecture and implementation of the Simulink bridge with the Epsilon family of model management languages and its integration with ModelFlow. **Chapter 6** contains the evaluation of ModelFlow which is composed of three case studies. **Chapter 7** summarises the thesis contributions and proposes lines of future work.

2 Background

This chapter introduces the key domains for the proper understanding of this research work. Section 2.1 provides the background of Model-Driven Engineering. Section 2.2 offers an overview of the background on Traceability. Finally, Section 2.3 offers background on Automated Task Processes.

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is a software development approach that places models at the core of the development process. Models are structured entities that capture the essential complexity of a system required for a given context or purpose. As models are inherently structured, they can be processed to be validated, transformed or to generate code from them.

MDE has been successfully adopted in industries such as aerospace, automotive, and telecommunications [128, 198]. In this section we provide an overview of the key concepts, challenges, and opportunities in MDE, which is at the core of this thesis as it defines its context and powers the proposed solution.

2.1.1 Models and metamodels

Consider the activity of modelling a society where people either like or dislike each other. Before we can name and identify which people who like or dislike each other, we need to define the concept of a *social network* that contains a list of *persons* which have a *name* and can specify two relationship types: *likes* and *dislikes*. These concepts would be captured in a metamodel (Listing 2.1) which would be used to specify whether a model has a valid structure or not [149, 171, 19]. In this case an example of a valid model (Figure 2.1) would be one with two people: Alice and Bob who, despite their polite appearance, dislike each other. This model is said to *conform to* the metamodel as it is using correctly the constructs defined in a metamodel. In contrast, we could invalidate this model by including an additional person called Charlie that has an email `charlie@mde.com`. The model would be invalidated as *email* is not defined in the metamodel as a known property for a person.

```
1 @namespace(uri="socialnetwork", prefix="sn")
2 package socialnetwork;
3 class SocialNetwork {
```

Figure 2.1: Social network sample model

```
 4     val Person[*] people;
 5   }
 6   class Person {
 7     attr String name;
 8     ref Person[*] likes;
 9     ref Person[*] dislikes;
10   }
```

Listing 2.1: Social network metamodel in Emfatic

The complexity of the metamodel can be influenced by how we intend to use the models. The metamodel from the example contains very little personal details but may have been defined with sufficient complexity if our only intention is to determine whether people like or dislike each other. However, if we intended to use the model to send emails to the people in it, then the metamodel would have lacked the required constructs to capture this information.

The modeling process is confined in a layered architecture (see Figure 2.2) with an abstraction layer and three modeling layers [16]. This architecture was proposed by the Object Management Group (OMG) Model-Driven Architecture (MDA) [138] and first achieved by Meta-Object Facility (MOF) [140] . The abstraction layer represents a real-world object that exists at level M0 and is represented by a model at level M1. From our example, the real Alice and Bob would exist in M0 while only their names and people preferences would be captured in the M1 model. The first modeling layer at Level M1 is the model which conforms to a metamodel at level M2. This is the conformance relationship discussed in the example where the properties and relationships of a Person are defined. At this level we can observe that metamodels are used as *modeling languages*. The next modeling layer at level M2 is the metamodel which conforms to a meta-metamodel at level M3 (which can conform to itself). In the example, we have the class *Person* that has three properties: *name*, *likes* and *dislikes* so the M3 level would define the concept of a *class* and of *property*. At this level we can say that the metamodels are defined by *metamodeling languages*. Examples of metamodeling languages include Ecore from the Eclipse Modeling Framework [175] and the MOF [140]. Another example of this architecture are XML documents (M1) used to represent a real-life system (M0) which at the same time conform to a custom XML Schema Definition (XSD) (M2) which in turn conforms to XSD (M3).

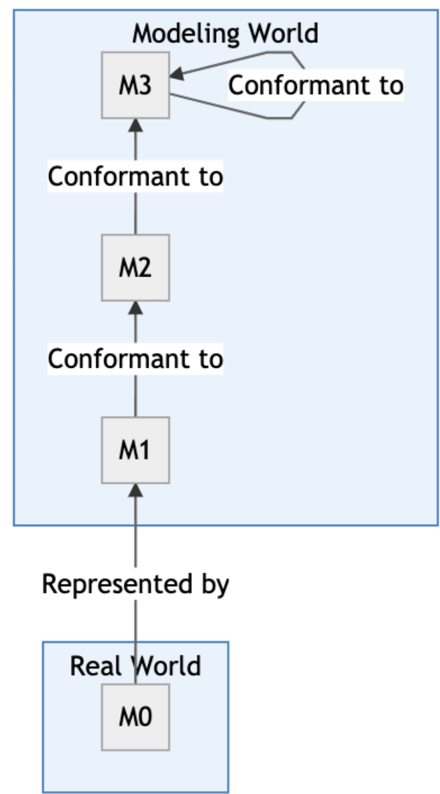


Figure 2.2: Modeling layers in a Model-Driven Architecture

In the example, the metamodel is defined using Emfatic, a textual representation of Ecore while the model instance is persisted in XMI although Figure 2.1 shows a graphical visualisation in which people are boxes and dislike relationships are red arrows.

Having explored the concept of models and metamodels, we now explore the characteristics of modeling languages, that is, metamodels.

2.1.2 Modeling languages

The previous section mentions that metamodels at level M2 can be used as metamodeling languages. Indeed, metamodels describe the *abstract syntax* of a modeling language, that is, the key concepts, properties, and relationships to be captured. A modeling language is not only defined by its abstract syntax but also by its concrete syntax and the semantics of them both [19, 90].

Abstract syntax. This aspect describes the structure of the language and its grammatical rules, that is, its constructs and allowed connections with each other [19, 90]. In modeling languages, the abstract syntax is commonly specified in a metamodel [90].

Semantics. This aspect provides meaning to the modelling constructs defined in the abstract syntax of the language and to their combinations [19].

The semantics of a language can be defined using natural language or, more rigorously, through a language such as Z ([167]) [171].

Concrete syntax. This aspect describes a specific representation of the modeling language along with a notation [90]. The concrete syntax allows the construction of models that conform to the language (metamodel) [171]. The representations of modeling languages are commonly graphical or textual but can also be based on arbitrary forms such as matrices, tables, and forms [90].

Classification of modeling languages

According to the purpose, modeling languages can be classified as *general-purpose* or *domain-specific*.

General-Purpose Modeling Languages (GPLs). These provide constructs and notations that can be applied to any domain for modeling purposes. GPLs are intended to be universal. Examples of GPLs include UML [142], MERISE [6], SSADM [5], IDEF [93] and SysML [143]. Although the Unified Modeling Language (UML) [142] is intended for analysis, design, and implementation of (object-oriented) software-based systems, its application domain is so broad that can be classified as general-purpose [19]. A similar argument can be made for the other GPLs such as MERISE which is intended for information systems.

Domain-Specific Modeling Languages (DSLs). These languages are tailored to the requirements of a specific domain, context, or company [19]. According to Kelly and Tolvanen [90] these languages are often more productive and easier to create than general-purpose languages because they raise the level of abstraction and use concepts from the specific problem domain. An example of a DSL is the Goal Structured Notation (GSN) [168] used to structure arguments and their relation to evidence which can be used to support assurance cases. Another example is the VHSIC Hardware Description language (VHDL) [82] intended for modeling electronic systems. The Business Process Model and Notation (BPMN) [136] is another graphical modeling language used to represent business processes. The minimal social network example can also be classified as a DSL as the concrete syntax indicates that models are captured in XMI, and the semantics and abstract syntax (metamodel) have been described in the previous section. Finally, the Web Modeling Language (WebML) [26] is a graphical DSL intended to specify the content, composition, and navigation features of web applications.

Classification of DSLs

DSLs can be classified according to their focus, style, notation, internality, and execution characteristics [19].

Focus. Depending on their domain of applicability, DSLs can be classified as *vertical* or *horizontal*. Vertical focus implies that the domain, industry or field of application is clearly defined e.g., VHDL. In contrast, horizontal focus implies that the DSL has a broader range of applicability e.g., SQL, WebML.

Style. Depending on the specification of control flow, DSLs can be classified as *declarative* or *imperative*. A declarative style expresses the logic of a computation without specifying *how* to achieve it i.e., the control flow. In contrast, an imperative style explicitly indicates the steps that need to be taken to achieve a specific computation.

Notation. The notation of a DSL can be *graphical* or *textual*. Graphical DSLs result in graphical models with graphical model elements such as blocks and arrows. Textual DSLs rely on structured text notations such as XML-based notations.

Internality. Depending on the use of a host language, DSLs can be either *external* or *internal*. Internal DSLs either give the feel of a domain to a host language either through the insertion of fragment DSL in the host language or by building on top of it to provide more abstractions, structures, or functions. In contrast, external DSLs are independent and have their own syntax and parser.

Execution. Modeling languages can be classified as *executable* when they provide execution semantics i.e., execution behaviour specification, that can be used to define and execute models [76]. Examples of executable modeling languages include Petri Nets [126], fUML [172], and BPMN [136]. An executable model must conform to an executable modeling language and define its behaviour in sufficient detail for it to be executed [76].

According to Hojaji et al. [76], execution semantics of an executable modeling language can be defined using three different approaches. The *denotational* approach uses algebraic and mathematical constructs. The *translational* approach defines these semantics by the translation of the model into another executable language through a model transformation. Finally, the *operational* or *interpretational* approach relies on an interpreter to create an initial representation of the execution state of a model and then modifies it through transitions from one execution state to another that result from the model execution.

Information about the execution of these models may be captured within a *model execution trace* which may include events, state transitions, and input or output parameters [76].

2.1.3 Model management operations

Model management operations are actions defined at metamodel level that apply at model level [149, 155]. It is through model management operations

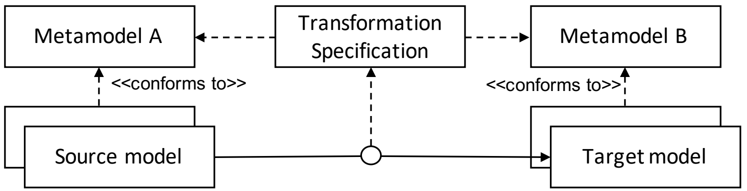


Figure 2.3: Exogenous Model-to-Model transformation

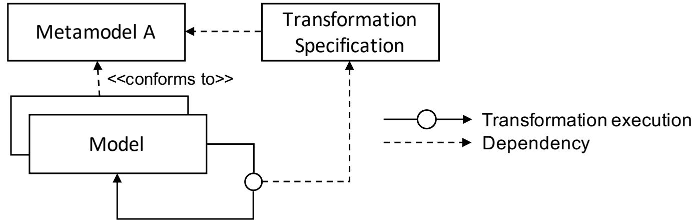


Figure 2.4: Endogenous Model-to-Model transformation

that concrete software development artefacts can be produced in an automated fashion from models [104]. Some model management operations can be specified through executable model management languages. In the following we introduce several kinds of model management operations.

Model transformation

Model transformations are operations which generate an arbitrary number of *target* artefacts from a number of *source* artefacts based on a transformation specification. The kinds of model transformations based on the nature of the source and target artefacts are presented below.

Model-To-Model Transformation. Model-to-Model (M2M) transformations can produce one or more target models from one or more source models. The nature of M2M transformation languages varies greatly as they can be declarative, imperative or hybrid, textual or graphical, unidirectional, or bidirectional, and semi-formal or formal. Moreover, if transformations produce models conforming to a different modeling language from the source model, they are called *exogenous* (Figure 2.3), while if the output and input models conform to the same modeling technology, the transformations are called *endogenous* (Figure 2.4).

Examples of M2M transformation languages include the Epsilon Transformation Language (ETL) [99], the ATLAS Transformation Language (ATL) [87], the VIsual Automated model TRAnsformations (VIATRA) [15], and the Query/View/Transformation (QVT) [141].

Model-To-Text transformation. Model-to-Text (M2T) transformations enable the generation of text such as documentation or source code, from models. Some source code generators are built atop general purpose languages and use

the model API to produce source code for a target programming language. Examples of this generation approach are the Ecore-to-Java transformation provided by EMF, and the C code produced with the Simulink Coder. The main issues with this approach are that static and dynamic code are intermingled and it is hard to grasp the final output structure [19].

Model transformation languages —as opposed to general purpose language code generators, alleviate these problems by using a configurable template-based approach. In this approach, the template allows to explicitly represent the structure of the output and clearly indicates where to insert the dynamically generated parts. Examples of M2T transformation languages include Acceleo [179], the Epsilon Generation Language (EGL) [156] and Xpand [176]. A particular case of M2T transformations are High Order Transformations (HOT) whose outputs are transformations.

Text-To-Model transformation. Text-To-Model (T2M) transformations can extract a model from a text. This kind of transformation is used for reverse engineering [19] but is also used by parsers that can produce models [171]. T2M tools require a grammar, a target metamodel, and a text-to-model parser. Example of T2M tools include ANother Tool for Language Recognition (ANTLR) [152], Xtext [184], and EMFtext [74].

Model validation

Modeling validation languages specify the structure of "valid" models, that is, models that conform to their specification and other wellformedness rules. The well-formedness specification that a modeling language can impose is limited by the expressiveness of the meta-modeling language by which it is defined [103, 19]. Model validation languages provide the means to apply more complex structural constraints to models that may not be possible to enforce or express through the basic constructs of a meta-modeling language. The ultimate goal of model validation programs is to check that a model complies with constraints imposed at metamodel level to detect errors in the model.

In general, model validation languages define *invariants* (i.e., Boolean expressions) associated to a *context type* (i.e., a model element type) and these are evaluated against all model element instances of the context type. Examples of model validation languages include the Object Constraint Language (OCL) [137] which is an OMG standard, and the Epsilon Validation Language (EVL) [103].

Model comparison

Model comparison languages enable the detection of differences between models. Model comparison programs produce a *difference-model* or *correspondence model* which contains the found differences. The comparison process consists of two phases. *Model matching* is the first phase and consists in matching two corresponding model elements using a matching strategy [19] which, according to Kolovos et al. [102], can be language-specific or identity-, signature-, or similarity-based. The second phase is *model differencing* which consists in the execution of differentiating algorithms that apply comparison on the previously found pairs of correspondent model elements. Examples of model comparison languages include EMFCompare [180], and Epsilon Comparison Language (ECL) [98].

Model composition

Model composition is the activity of taking a pair of source models and combining them into a new target model with the aid of a correspondence model [24]. A model composition framework must be able to identify corresponding elements in the models that are to be composed, indicate how these elements are to be merged and how to transform non corresponding elements to avoid losing information. Model merging is a specific composition scenario where all information from the input models is present in the output model and where there is no information duplication [24]. Examples of model composition frameworks include the Atlas Model Weaver (AMW) [38], the Glue Generator Tool (GGT) and the Epsilon Merging Language (EML) [96].

2.1.4 Global model management

Managing models can be done at two levels which in the literature are referred to as *modelling in the small* and *modelling in the large*. *Modelling in the small* refers to the activity of managing *elements* of models and metamodels. In contrast *modelling in the large* refers to the activity of establishing and managing relationships among models *as a whole* [23]. A model that has interrelated models as elements is called a *megamodel* [95]. The activity of modelling in the large can also be referred to as megamodelling or global model management. Whilst megamodelling is interested in mappings and operations over *models*, intermodeling is a particular case where the models of interest are *modelling languages* i.e., *metamodels* [72].

Megamodeling

The term *megamodel* was coined by Bézivin et al. [17] which defined it as a terminal model in which elements themselves are models. One key aspect around *megamodeling* is the management of these models, which may include browsing and editing these models (e.g., AM3 [7]) or performing operations on collections of models such as *map*, *filter*, *reduce* [161] and *slicing* [160]. However, one of the challenges for the management of megamodels is taming the technological heterogeneity of tools, frameworks and languages used by MDE projects [44].

Regarding the theory behind megamodelling, Diskin and Maibaum [40] highlighted an overlap between Category Theory and model management activities. Using a mathematical framework based on category theory, Diskin et al. [42] proposed a mega-modelling framework based on graphs, graph mappings and operators along with a library of structural design patterns for megamodel engineering. This theoretical framework has been used to set mathematical principles for model synchronisation [39] based on models and model mappings, and bidirectional transformations [40], among other operations. Another mathematical framework that has been used to support model synchronisation and bidirectional transformation is algebraic *lenses* [55]. In contrast to the approach presented by Diskin [39], lenses only use models as input (leaving out model mappings) which can make their synchronisation results erroneous [39].

Another approach to global model management was proposed by Melnik [122] which defined high-level algebraic operators such as *match*, *merge*, and *compose* to manage models based on SQL expressions. In addition, Melnik [122] also provided a prototype of a model management platform, called Rondo, which can manipulate models and mappings as first-class objects and execute model management scripts using the defined operators.

MDE projects

Metamodels, models and model management languages are at the core of MDE projects. MDE projects can use a wide range of technologies and artefacts (e.g., models, metamodels, model transformations). Without any structured representation of these artefacts or proper documentation, users can have difficulties understanding their classification, the flow in which they are used and other properties and relationships within the projects which in turn makes projects difficult to analyse, build, test and reuse [44].

Butting et al. [22] proposed the use of an *artefact model* to explicitly capture information about the artefacts in a project, their languages and the tools producing and consuming them. This artefact model would be able to describe

all possible interactions with other artefacts and tools in the project during its lifetime. The main goal of this artefact model is to represent an MDE project in a structured way to facilitate dependency analysis, communication, impact analysis, compliance checks, data-driven decision making and metrics computation. The artefact model itself would be composed of artefacts in the MDE project, the tool chain configuration along with traces of any tool chain executions and artefact relationship knowledge.

An alternative approach to structure and visualise artefacts of an MDE project is taken by Di Rocco et al. [44] which uses a megamodel-based approach combined with reverse engineering heuristics to identify specific artefact types or relationships in an MDE project.

2.1.5 Model management platforms

This section introduces some of the available model management platforms and the tools and languages that are supported by them.

Atlanmod

Atlanmod is a platform that proposes a set of model management tools including ATL, a model-to-model transformation language, NeoEMF an efficient model persistence tool for EMF models that can be backed by various NoSQL datastores, and MoDISCO a tool that extracts models from legacy systems to describe them in a structured way.

Epsilon

Epsilon is a framework of inter-operable languages and tools designed for model management tasks like model navigation, validation, and transformation. The Epsilon Object Language (EOL) [97] is an OCL-like model query and transformation language that all other Epsilon languages are built on top of. Among these model management languages we find the Epsilon Validation Language (EVL) [103] — designed to evaluate invariants on model elements, and the Epsilon Transformation Language (ETL) [99] — targeted at model-to-model transformations.

Epsilon has a layered architecture (see Figure 2.5). The Epsilon Model Connectivity (EMC) layer provides abstraction facilities that allow models of arbitrary technologies (e.g., EMF, XML) to be managed in a uniform manner in any of the Epsilon languages. Concrete EMC implementations for different modelling technologies such as EMF, or PTC-Integrity Modeler, are known as (epsilon model) drivers.

Listing 2.2 shows a sample EOL program that navigates and manipulates a

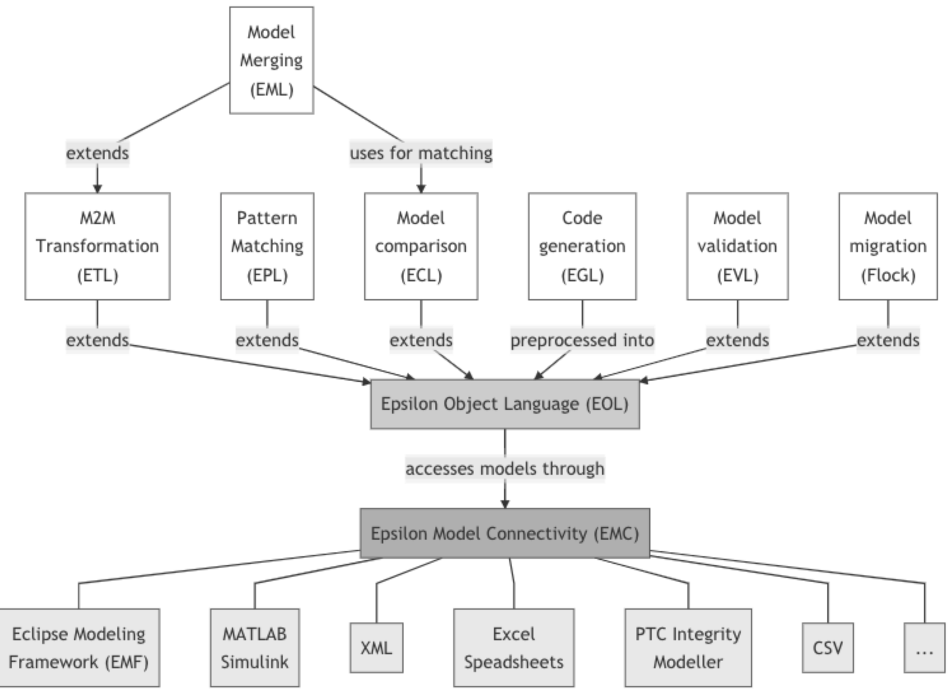


Figure 2.5: Epsilon architecture (Image from [52])

model `M`[1] of arbitrary underlying modeling technology (e.g., EMF, XML). In the first line, the first of all the elements of type `Block` contained in the model is selected and then assigned to a new variable named `element`. In line 2 the value of its `name` property is retrieved while in line 3 its `evaluate()` method is invoked. Further down, line 4 shows how a new element of type `Block` is created and assigned to the `newElement` variable while line 5 sets its `name` property.

```
1  var element = M!Block.all().first();
2  element.name;
3  element.evaluate();
4  var newElement = new M!Block;
5  newElement.name = "My Block";
```

Listing 2.2: Sample EOL program

The EOL program in Listing 2.2 can be executed on models of arbitrary technology because the model is injected to the EOL interpreter at runtime by an arbitrary driver. The syntax that an EOL program uses to create and delete model elements, to set and get their properties, or invoke their methods does not depend on the driver. The contribution of a driver on any Epsilon program is the availability of model element types, their properties, and additional methods at runtime. For example, the `first()` operator works on collections

[1] The character “!” is used in Epsilon to separate the runtime name of the model from a model element *type* or *kind* available in that model.

and is handled by the EOL engine by default[2]. In contrast, the `all()` method in Listing 2.2 delegates the collection of all elements of type `Block` to the driver that handles model `M`. For Listing 2.2 to terminate successfully, the driver that provides and manages model `M` would need to know how to handle model elements of type `Block` with a `name` property and an `evaluate()` method.

Epsilon currently provides drivers for a variety of modelling technologies including EMF, XML [97] and Spreadsheets [58]. Sec. 5 presents the architecture of the Simulink driver which is another contribution in this work and was first introduced in [162].

2.1.6 MDE challenges

As MDE popularity increased, several limitations in terms of its efficiency and capacity are hampering wider adoption [62, 8, 101]. In particular, MDE has shown scalability, co-evolution, heterogeneity and interoperability challenges. Scalability becomes an issue when models become very large as they require more storage, more memory, and collaboration becomes more complex. In parallel, preserving the consistency of model, metamodel and model management programs, which are disjoint artefacts but mutually related, requires efficient co-evolution mechanisms. Additionally, the heterogeneity challenge comes from the use of different domain specific modeling languages at the various stages of the development process. Moreover, when the domain specific modeling languages are based on different technical stacks or are managed by different modeling tools (e.g., EMF and MATLAB/Simulink) interoperability becomes imperative. In the following we discuss the aforementioned challenges in more detail.

2.1.7 Scalability

Modeling technologies have shown scalability issues when dealing with large models (millions of model elements) [101, 104, 62] in particular regarding model persistence, but also in model management activities like model querying and transformation [104]. We introduce some of the strategies that have been proposed to address the scalability challenge and highlight research works that support them.

Static analysis. Programs can be analysed both at runtime and before execution. The former case is referred to as *dynamic* analysis while the latter is called *static* analysis. Static analysis usually is done at the source code level or at some form of object code. This analysis can be used for multiple purposes

[2] Other collection operators such as `select()` and `collect()` are provided in EOL by default although drivers may override their behaviour.

including the notification of compilation errors, code linting, and detection of sub-optimal controls or unused variables. Static analysis can be used as a performance enhancement tool when it can detect parts of the code that can be optimised (and even change them) to improve runtime performance. In MDE, static analysis has been used to optimise queries and filters in object collections e.g., Ali et al. [2] and to partially load models to reduce their memory footprint e.g., Wei and Kolovos [192], Wei et al. [193] and Jahanbin et al. [85].

Laziness. This approach relies on delaying the evaluation of an expression until its result is needed. This can reduce the invocations of functions and improve the overall program performance. In addition, laziness can also be combined with *caching* which consists of storing the value of a computation so that future invocations can skip the re-computation. While caching can also improve performance by reducing computation times, it must be handled correctly to invalidate and re-compute cached values when the stored ones are no longer valid. Lazy computations are common in model management programs both for model navigation and other declarative operations such as model-to-model transformations. Functional languages e.g., Haskell and functional constructs such as *Streams* in languages like Java make use of lazy computations. Consequently, model navigation languages that rely on functional constructs such as OCL and EOL have been adapted to adopt these strategies in works like Tisi et al. [187] and Madani [109]. For hybrid languages like ATL and ETL, laziness has been adopted by compartmentalising computations as on demand operations but also by enabling some transformation rules to be invoked on demand (e.g., Tisi et al. [185]). For example, in ETL some transformation rules can be annotated as *lazy* so that the target model elements are only created if the program explicitly calls that rule.

Incrementality. Another strategy to cope with scalability issues is incrementality. This strategy relies on detecting artefact changes so that computations that use those artefacts can be re-executed but only for the affected parts, reducing computation times and improving efficiency. In practice, incrementality minimises the execution of redundant computation by responding to changes in resources like models [135]. In MDE systems, incrementality can reduce the number of artefacts re-generated after models or model management programs are modified. This can reduce the computational efforts by limiting the number of artefacts that need to be re-compiled, tested or analysed [60, 75].

However, when speaking about incrementality support, it is important to clarify what *type* of incrementality is supported. In model transformations alone, incrementality can be of different types, such as edit-preserving incrementality, target incrementality, and source incrementality [30]. Edit-preserving incre-

mentality is focused on preserving manual changes to generated artefacts [132]. In contrast, target incrementality focuses on updating a target artefact based on changes in a source model by re-executing a transformation whose output needs to be merged with the previous target [132]. Source incrementality is similar to target incrementality but it limits the execution of the transformation to only parts affected by the changes in the source model, seeking to eliminate the need to perform a merge of the old and new target [132].

To support source incrementality, model transformation tools may rely on recording property access traces, using model differencing techniques or employing static analysis [135]. Frameworks such as EMFCompare [180] or languages like ECL can be used to compute the differences between model versions. Property access traces are commonly used by rule-based model transformation languages e.g., Hearnden et al. [73], Rose et al. [156], and Tratt [188]. Regarding model-to-text transformations, another strategy to support incrementality involves manually or automatically computing source model signatures associated to templates e.g., Ogunyomi et al. [133].

Model transformation incrementality can also be classified as *live* and *offline* depending on how the change detection occurs. Live incrementality directly propagates events between models already loaded in memory relying on change events emitted by the modeling framework holding the source model [86]. In contrast, in offline incrementality models are not loaded and the approach needs to keep track of the original unmodified source to later compare it with the new source [86]. Once the changes have been identified source and target artefacts are loaded to propagate the changes [86].

Incrementality is also available in the domain of model querying. For example, EMF-IncQuery [14] is a environment for incremental queries over EMF models that is based on an incremental pattern matching system for graph patterns relying on the RETE networks algorithm. Similarly, Cabot and Teniente [25] has investigated whether OCL expressions can be incrementally evaluated to detect if OCL constraints are remain valid after modification to a UML model.

Reactiveness. The reactive manifesto [18] is a document that defines the core principles of reactive programming such as responsiveness, resilience (responsive upon failure), elasticity (responsive under various workloads) and message-driven (asynchronous and non-blocking). In software, reactive programming is a programming paradigm that consists in reacting to changes such as events or values [131] and has been mostly studied in relation to functional languages. As such, the behaviour of components is determined and triggered by instances observed on event streams [15]. A reactive execution requires both the ability to detect updates or requests as events and to incrementally execute based on the changes. Reactiveness in MDE has been adopted in model transformation

languages such as VIATRA [15] and Reactive ATL [115]. In the case of VIATRA, transformations are written as actions that are triggered when a specific change event occurs. In the case of Reactive ATL the program is the same as a regular ATL but some rules are triggered in response to changes in the source model.

Concurrency, parallelisation, and distribution. Concurrency can be described as the *potential* for parallelism [194]. In practice when programs exhibit concurrent behaviour, they allow parts of themselves (or *threads*) to be executed in arbitrary order without affecting the program's outcome. When these threads are effectively handled by different processing units, in a way that individual threads could potentially be executed simultaneously, the program is said to be executed in *parallel*. A particular case of parallelisation is *distributed computing* in which threads are executed on different network computers where they can independently fail. Parallelism and distribution have been investigated in the MDE community to tackle scalability issues. Tisi et al. [186] proposed a parallel implementation of the ATL compiler and virtual machine. Similarly, Madani has worked on parallelisation of Epsilon languages including EVL (validation) [110], and EOL (navigation) [111]. Other examples that execute parallel and distributed ATL model transformations using frameworks for parallel processing include works by Burgeno [21, 20] which are implemented atop the Linda framework and work by Benelallam et al. [12] implemented atop MapReduce. Benelallam et al. [13] later proposed two improvements of the ATL execution atop MapReduce which include a model partitioning algorithm taken from graph theory and an algorithm to distribute data from declarative model transformations based on static analysis in a greedy and prioritised way.

Model persistence

The achievement of intensive and fast model queries and transformations on large models is closely related with the model persistence mechanism [62]. The ability to store, access and update large models with a low memory footprint can be achieved through file-based model fragmentation (e.g., XMI/JSON-based), through a model persistence layer backed by a database (e.g., Teneo/Hibernate, NeoEMF, MongoEMF) or through the use of model repositories (e.g., CDO, Morsa, EMFStore) [62]. Alternatively, model indexes can make file-based model storage more efficient by monitoring models and mirroring them in a model index backed by a scalable database [62, 104]. These indexes enable efficient global queries as they are kept synchronised with the latest version of the models without having to copy files locally or load them into memory. We now discuss some of the strategies for model persistence and their scalability.

XMI. The XML Metadata Interchange (XMI) format is an Object Management Group (OMG) and an ISO/IEC (19503:2005) [104] standard that

is used to describe objects by representing them as XML elements and/or attributes [139]. In addition, XMI specifies how to link objects within and across XMI documents, how to validate these documents using XML schemas and how to identify objects within the documents [139]. XMI can be used to describe models conforming with the Meta-Object Facility (MOF) including UML models, and models from the Eclipse Modeling Framework (EMF). Introduced to support modeling tool interoperability and prevent vendor lock-in, XMI has become the most widely adopted model persistence format [104].

One of the limitations of XMI, inherent to XML, is that the models need to be fully read and loaded in memory before they can be queried. While this is not a limitation for small models, it negatively impacts loading times and memory usage in large ones [104]. Moreover, despite having the ability to store models across a range of XMI files, many tools store the models in a single file by default [62]. The *fragmentation* of resources can be used as a strategy to reduce scalability issues [9]. Another limitation of XMI is that its serialization is inherently verbose (because it extends XML) which ends up producing model files that are larger in size compared with the actual amount of information they need to store [104].

Database-backed persistence. To deal with the scalability issues of file-based persistence, several solutions have been proposed where models are serialised into a database. Most of the solutions target EMF models and they usually deserialise the database contents into a memory representation that is compatible with the EMF API. Originally, relational databases were the preferred solutions in frameworks like Teneo/Hibernate [174]. While this solution can be more efficient than file-based persistence, extensive join operations can be required to navigate the models, impacting performance. Similarly, migrating models based on metamodel changes require updates in the database schemas that can be hard to maintain/orchestrate. This motivated the development of alternative solutions backed by NoSQL databases such as Morsa [146], MongoEMF [79] and NeoEMF [32].

Model repositories. An alternative strategy to cope with scalability issues derived from model persistence is the use of model repositories. A model repository offers remote model access and enables users to concurrently access the model while also providing model versioning and transaction support [146]. Examples of model repositories include CDO [181], EMFStore [177], Modelio [124], MagicDraw and MetaEdit+. These repositories may offer different choices to use as backend for the model storage. For example, CDO supports both relational and NoSQL databases as store.

Model Indexes (Hawk) An example of a model indexing framework is Hawk [10] (Figure 2.6) which enables developers to perform efficient global queries [62] on a NoSQL model-element-level graph database which mirrors the

contents of (possibly fragmented) models stored in file-based version control systems, without needing to maintain a complete copy of all model fragments in their local workspace [10, 61]. Figure 2.7 provides an example of a Hawk model index where the top-left box represents an Ecore metamodel, the bottom-left box represents an XMI instance of such metamodel and the right box represents the Hawk index. The Hawk index contains nodes of different nature i.e., `Metamodel`, `Type`, `Element` and `File`. Hawk keeps an index of the `Metamodel` and `File` nodes so that they can be efficiently accessed for querying. `Type` nodes belong to `Metamodel` nodes whilst `Element` nodes are linked to `Type` nodes through the *isOfType* relationship and to `File` nodes through the *file* relationships. In addition, `Element` nodes may have metamodel-specific relationships to other `Element` nodes (e.g., book) as defined by their `Metamodel` file (top-left box). `File` nodes are also related to the `Repository` nodes which are used by Hawk to detect repository changes (e.g., Git, SVN, Local Directory) and trigger index updates.

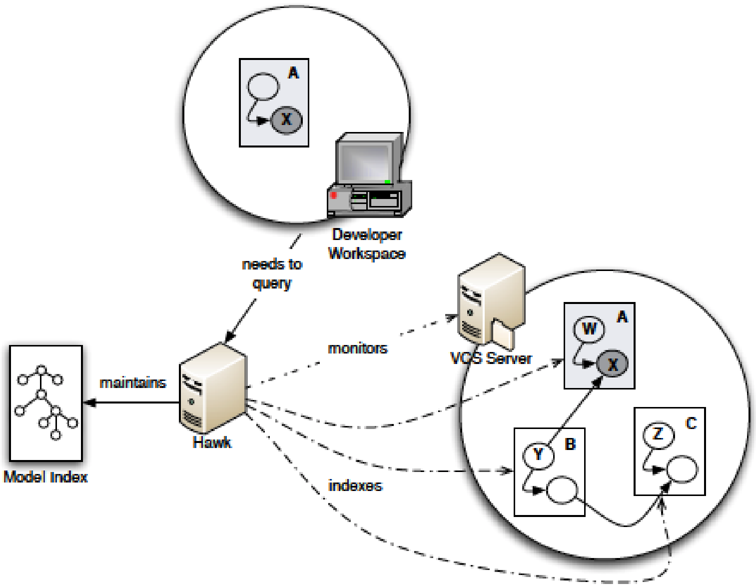


Figure 2.6: Model indexing framework architecture (Image from [10])

2.1.8 Co-evolution

In addition to scalability challenges, Model-Driven Engineering tools have had issues with managing the co-evolution of its core artefacts—models, metamodels and model management operations, to keep a system of inter-related models in a mutually consistent state [149, 41].

Approaches to manage their co-evolution have been broadly categorised as approaches where metamodels change or not [149]. Paige et al. [149] uses the following characteristics to describe current co-evolution solutions: *scope* —used to define whether the solution applies to a single MDE artefact or if a change

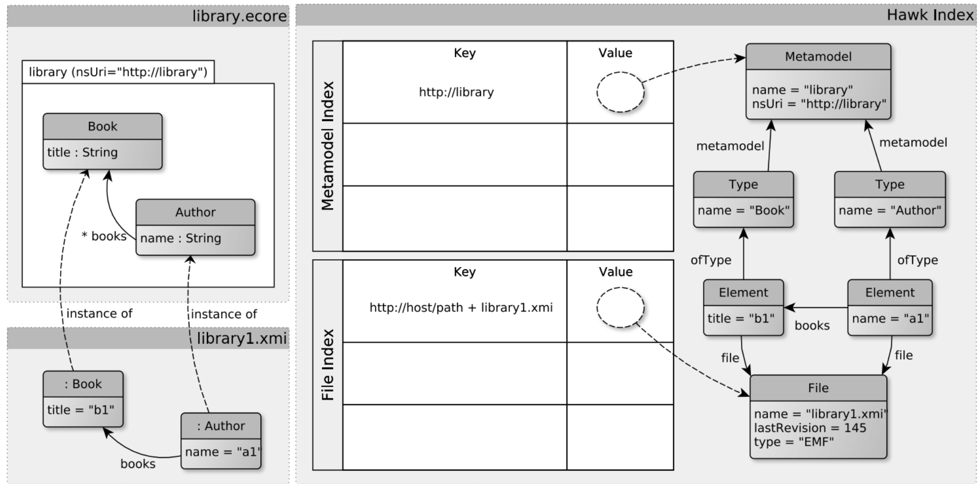


Figure 2.7: Example of a Hawk model index (Image from [61])

in the metamodel triggers changes in other MDE artefacts, *automation* —to determine whether the solution is manual or fully automated, *environment* —whether a specialised model editor or operation recorder is required, *conformance* —if it provides means to restore metamodel conformance when a metamodel changes and whether constraints are required to verify conformance.

Research on bidirectional model transformations (BX) and lenses addresses the challenge of preserving mutual consistency of a pair of inter-related models by discovering deltas and propagating the changes [41, 55]. Stevens [169] highlights the need for bidirectional transformation in Model-Driven Engineering and points out how the QVT standard for model transformations provides semantics based on lens-like structures. Paige et al. [149] highlights that another challenge to support modeling artefacts co-evolution is managing the heterogeneity of inter-dependencies between MDE artefacts.

2.1.9 Heterogeneity and interoperability

As systems grow and become more complex, several models may be required at different stages of the development process. Models may vary greatly depending on the development stage they are used on or their purpose. For example, at early stages of the development process a project may use requirements models (e.g., ReqIF or IBM Rational DOORS) to capture the application requirements, while at the development stage the project may use system models (e.g., SysML) to capture the composition of the various systems to develop.

The various models used in the development of a system are likely to conform to different modeling languages which may not be based on the same technical stack. For example, a SysML model created with Papyrus would conform to an EMF-based SysML metamodel, but a system model created with

MATLAB/Simulink would only be compliant with the MATLAB's modeling language. Considering that there are many modelling frameworks available e.g., MOF and EMF, a broad range of model management languages e.g., Epsilon, Acceleo, ATL, Kermeta, QVT, OCL and multiple tools to manage the models, the integration of the tools and formats may involve significant effort [28].

A significant interoperability challenge comes from the gap between tools used in industry and academy. Proprietary modelling tools (e.g., MATLAB Simulink and PTC Integrity Modeller) are used predominantly in industry, while most of the Model-Driven Engineering research is centred around open-source modelling frameworks such as EMF [198]. To bridge this gap, proprietary tools must offer exchange mechanisms of their models from and into standardised interchange formats (e.g., XMI) [92] and model management platforms must be able to manipulate models beyond the most common research metamodeling languages such as EMF. The Open Services for Lifecycle Collaboration (OSLC) [144] is an initiative that aims to simplify tool integration through a set of specifications for different aspects of application and product life-cycle management. Several proprietary software vendors are exposing a range of services through OSLC, including PTC Integrity and IBM Rational DOORS.

2.2 Traceability

Gotel et al. [67] defines traceability as "the potential to relate data that is stored within artefacts of some kind, along with the ability to examine this relationship". Traceability has many applications including requirements management, change management, impact analysis, verification, reuse, system understanding, audit and certification [195]. In software processes there is great interest in achieving *end-to-end traceability* where software development products are inter-related by trace-links through all the phases of the development process [148]. This section introduces key traceability concepts and classifications, and then discusses the key features and limitations of a selection of traceability tools. Finally, the section introduces how traceability is used in Model-Driven Engineering activities.

2.2.1 Defining traceability

There are multiple definitions for the term *Traceability* (e.g., [165, 148, 67]) which seem to vary according to the context and purpose of the research. The aforementioned traceability definition is stated at a fundamental level which highlights the storage of the traceability information and the importance of being able to navigate those relationships. However, the same authors provide a more formal definition for traceability as "the potential for traces to be

established and used" [67].

The latter definition uses the term *trace* which refers to a "triplet of elements comprising a source artifact a target artifact and a trace link associating the two artifacts" [67]. The term *trace* can also be used as a verb, in which case it refers to the ability to "[follow] a trace link from a source artifact to the target artifact or vice-versa" [67]. Figure 2.8 illustrates how a source and a target artefact are related through a trace link which is navigable both ways. Note that the definition of *trace link* varies in the literature depending on its typed/untyped, binary/n-ary and interconnected/isolated nature [113].

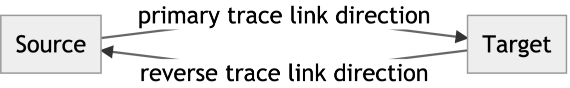


Figure 2.8: Trace triplet [67]

Types of traceability

There are various types of traceability and some of the most common types are shown in Figure 2.9. *Backward* traceability follows links of an artefact back to the artefact from which it was derived, in contrast, *forward* traceability is concerned with following links of an artefact to find those artefacts derived from it. *Horizontal* traceability follows traces between artefacts that are at the same project phase or abstraction level, whilst *vertical* traceability applies to artefacts that do not satisfy the previous condition. As traceability is intensely researched in the software requirements community, *Pre* and *Post Requirement Specifications* deal with traces created before or after said specification is formalised. Other kinds of traceability not represented in Figure 2.9 are *functional* and *non-functional* traceability, the former concerned with artefacts being transformed into other artefacts and the latter with informal traces which provide reasoning or context information. Finally, *implicit* traceability results from an inherent relationship between traced artefacts whilst *explicit* traceability has to be created as it cannot or should not be inferred.

To make the trace links more useful they can be enriched with attributes or by classifying trace links into types with richer semantics. There are multiple propositions for traceability link classifications which may be flat (e.g., Spanoudakis et al. [166], Ramesh and Jarke [153]) or hierarchical (e.g., Dahlstedt and Persson [31]), while some researchers advocate that trace link classifications should be built on a project-specific basis e.g. Paige et al. [147].

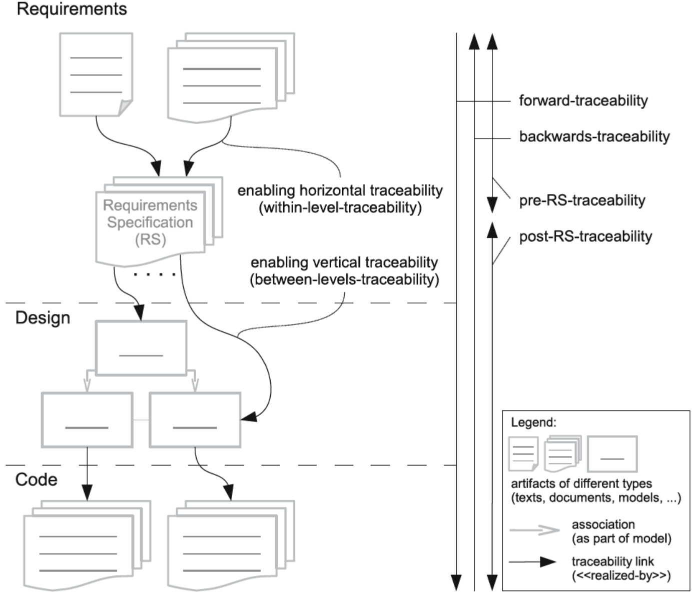


Figure 2.9: Types of traceability (Image from [195])

Activities

There are several activities required to achieve traceability. Gotel et al. [67] identify four activities in their proposed generic traceability process model which are planning, creating, maintaining, and using traceability. The first activity is about the identification of traceability needs and resources and the definition of a Traceability Information Model (TIM) which defines the information that should be captured to support traceability i.e., traced elements, their granularity and trace link types. There is a broad range of TIM propositions (e.g., Drivalos et al. [45], Katta and Stålhane [89], Mustafa and Labiche [130], Ramesh and Jarke [153], Taromirad [171]), and only recently Mustafa and Labiche [129] proposed a set of requirements for traceability model solutions and evaluated the completeness of several published TIMs. Creating traceability involves the acquisition, representation and storage of traces and may be achieved with two possible approaches: trace creation or trace recovery (discovery) [67]. Traceability maintenance is interested in preserving the traceability information relevant and consistent whilst the artefacts being traced evolve. Finally, traceability usage involves traceability visualisations such as matrices, graphs and hyper-links, and traceability retrieval through queries.

2.2.2 Demanding traceability

Regulated industries that develop safety-critical systems often must comply with guidelines and standards to certify their systems as safe and secure [154]. Usually, these guidelines prescribe activities and deliverables around software de-

velopment and verification processes which may also consider quality assurance and configuration management [154]. Examples of guidelines and standards that demand traceability are described below.

DO-178C is a standard for satisfying airworthiness requirements in software of airborne systems and equipment used on aircraft and engines provided by the US Federal Aviation Authority (FAA). DO-178C is an update of DO-178B [157]. The DO-178C guidelines require bi-directional traceability of the software development process which includes trace data between (a) system requirements allocated to software and high-level requirements, (b) high-level requirements and low-level requirements, and (c) low-level requirements and source code [157]. The objective of keeping these traces is to ensure that the functional, performance, and safety-related requirements of the system can be traced to source code passing through high- and low-level requirements [157]. Additionally, DO-178C also requires bi-directional trace data about the software verification process which includes trace links between (a) software requirements and test cases, (b) test cases and test procedures, and (c) test procedures and test results [157]. These traces are required to verify that the complete set of test cases was developed into test procedures and that all of these were executed. Trace links may be shown through naming conventions or by using references either embedded or external to the software data [157].

ISO 26262 is a standard for safety critical systems with electrical and/or electronic systems that are installed in road vehicles [84]. This standard requires traceability between safety related artefacts, for example from hazards to safety goals, to safety requirements, to the structure and behaviour of these safety requirements, to the code and to tests [114]. ISO 26262 recommends bi-directional traceability and requires artefacts to be versioned and have unique identifiers [114].

2.2.3 The challenges of traceability

Despite its many advantages such as change impact analysis, system understanding and regression testing [65, 114], traceability is known for being hard to achieve [29]. Particularly in large and continuously evolving software systems, creating and maintaining trace links can be a costly activity that requires a lot of effort and discipline [29]. Furthermore, poorly defined traceability processes, inadequate user training, and lack of effective tooling can prevent the exploitation of traceability [65, 88].

2.2.4 Traces in Model-Driven Engineering

Traceability in the context of Model-Driven Engineering is used to capture relationships among modeling artefacts and generated code.

Model management traces capture information derived from the internals of a single model management operation or from the relationships among a group of model management operations. Model management languages often produce model management traces as a side product of their execution. To illustrate the variety of model management traces at model element level offered by different model management languages, we focus on the traces offered by the languages of the Epsilon family.

As a result of the execution of a model-to-model transformation with ETL [99], the language provides a *transformation* trace which contains information about the source and target model elements consumed and produced by each transformation rule. In contrast, the EVL [103] model validation language produces a *constraint* trace which links model element instance, the rule they are validated against and the result of the validation. Similarly, the ECL [98] model comparison language produces a *comparison* trace which contains the pairs of model elements being compared, the rule that compares it and the result of the comparison. Merging activities in EML [96] require the execution of model comparisons and provide a *merge* trace for each match containing the resulting merged elements and the merge rule which produced them. For model-to-text transformations, the EGL [156] language produces a *template* trace which contains the information of the templates and variables used to produce output files.

Traceability at model level occurs when relationships are established among models as a whole. For example, the transformation of model `A` (source artefact) into model `B` (target artefact) through transformation `T` (trace link) illustrates such a trace. Traceability at this level is tightly related with the domain of mega-modeling and global model management discussed in section 2.1.4. At this level, traceability may involve non-model management relationships such as the *conforms to* relationship between a model and its metamodel.

2.2.5 Tools

Tools are important to support traceability management activities. Maro et al. [113] recently proposed ten guidelines for traceability tool developers that facilitate the trace maintenance activity. Their work was inspired by the set of guidelines proposed by Gotel and Mäder [66] regarding what to look for in traceability tools to assist engineers in their decisions to adopt a traceability solution. These guidelines are based on identified factors that affect traceability maintenance: versioning, tool boundaries, configurable semantics,

and consistency specification. Broadly, their guidelines propose that tools should support versioning of their internal traceability models, enable the extraction of deltas for all traced models, expose interfaces to access the managed models, and provide common interfaces for tool adapters.

In the following we introduce a set of relevant traceability tools and discuss key features regarding their interoperability and architecture.

DOORS. One of the most widely used requirements management tool in industry [113] is the commercial IBM Rational DOORS [80] product which can manage traces to requirements. This tool stores requirements and traceability links in a database and can compute deltas on its requirements which are used to notify of possible inconsistencies with their traced artefacts. DOORS enables external access to its requirements via OSLC services (Requirement Management) and can also be extended to consume services of external tools that provide OSLC services regarding change, quality and/or architecture management [81]. One of the main deterrents of the adoption of DOORS is its onerous acquisition cost.

YAKINDU Traceability [1] is a commercial traceability solution. In YAKINDU, traceability models are processed as EMF models that can be configured for each project. Traced model types require an EMF representation which if unavailable must be produced by tool adapters. YAKINDU provides a long and varied list of adapters which range from Google products such as Gmail and Calendar, and commercial tools such as IBM Rational DOORS, RHAPSODY, PTC Integrity and MATLAB Simulink/Stateflow to code e.g., Java (JDT) and C (CDT), and Eclipse projects e.g., Mylyn, Papyrus. Supported artefact types determine how the versions of its artefacts are handled. YAKINDU uses artefact versions to detect suspicious traceability links and provides a *snapshot* feature that allows browsing traceability information at different points in time [77]. The tool provides *diff* and *merge* actions on its traceability models but not for its traced artefacts. In addition, YAKINDU provides a rule-based language to specify link derivation which can be stored or computed at memory.

Capra [182] is an open-source traceability tool developed under the Eclipse Foundation. The architecture of Capra is presented in Figure 2.10 where four components are connected to a generic component through Eclipse extensions and extension points (service provider and consumers). As traceability needs to change from project to project, Capra allows the definition of custom trace links through the Traceability Metamodel extension point. Its artefact handler component allows the registry of new artefact formats to be supported in the traceability solution which (like YAKINDU) also require an EMF representation. Capra's persistence extension point enables the storage of the traceability model e.g., per project or per workspace, and allows the integration of the traceability

models with version control solutions like EMF Store, CDO or Git [112]. Capra provides a set of artefact extensions including EMF, Microsoft Office and Excel, Java code and the Hudson continuous integration tool. In addition, Capra offers two visualisation features: matrix- and graph-based. Capra offers Java APIs that can be used to access information about artefact wrappers, traceability links, and their contents. This API is used by its visualisation features and can be used for other navigation scenarios.

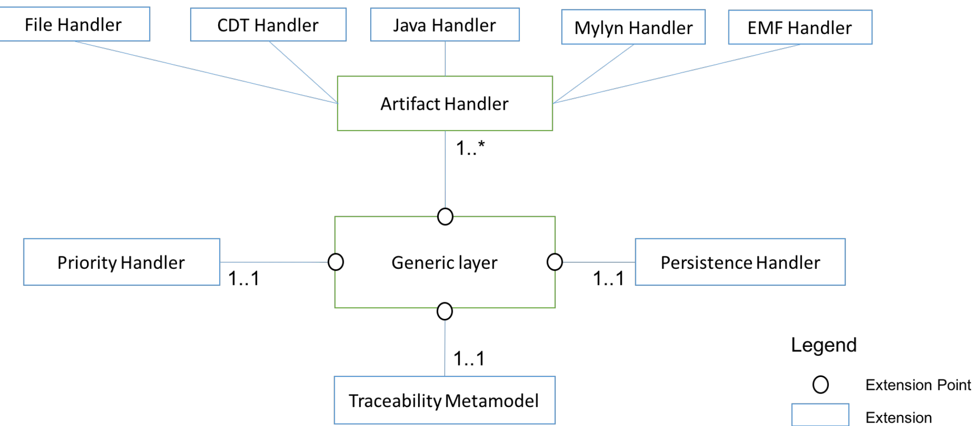


Figure 2.10: Capra architecture (Image from [182])

VeroTrace [190] is a commercial traceability solution produced by Verocel. It supports the creation and maintenance of bi-directional traces between requirements, design, and verification artefacts. Additionally, it provides validations and coverage reports. VeroTrace also supports impact analysis and can generate document and web reports.

Simulink Traceability [121] is a MathWorks plugin which enables the establishment and management of traces among Simulink requirement, design, and test artefacts. Traceability information is embedded in the environment where the artefacts are developed. Like other MathWorks add-ons, this plugin can manage the traces programmatically which may be used for custom and automated trace management procedures.

ChainTracker [69, 70, 71] is a state-of-the-art model-to-model and model-to-text transformation analysis tool. The main contributions of this tool are trace information collection and analysis in the form of visualisations. Within its traceability model, ChainTracker considers not only model resources but also how metamodel constructs at attribute level are used by invoked rules in the model management tasks. Figure 2.11 shows the main screen of ChainTracker. The top left panel shows the model transformation composition visualiser which uses vertical lines to represent models, blue rectangles to represent model elements (yellow when selected), black dots for element attributes connected through implicit and explicit transformations using red and green arrows [71].

Then the right panel shows the transformation code viewer which shows the transformation programs and how they involve model elements (which can be highlighted). Then the bottom panel shows model element information. As of the time of writing, the traceability tool is not publicly available.

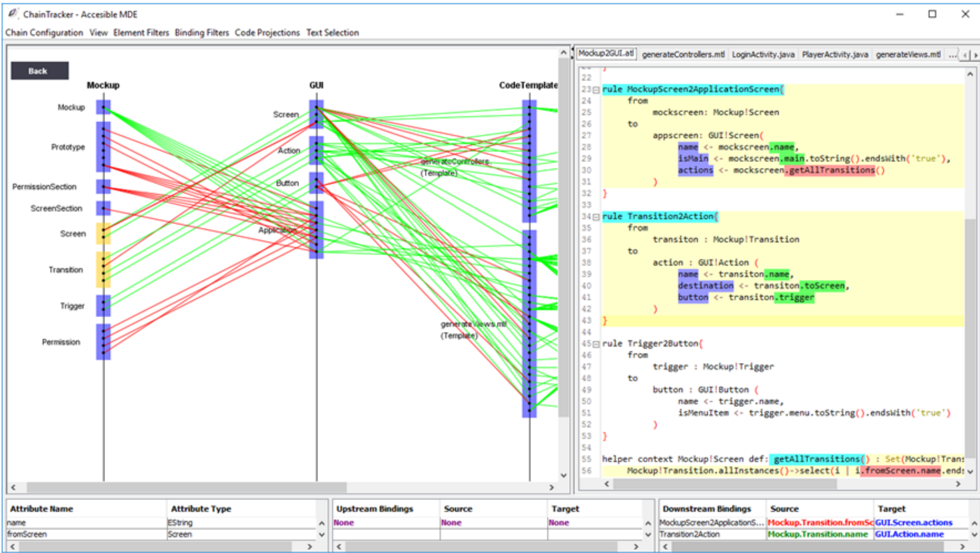


Figure 2.11: ChainTracker main screen (Image from [68])

2.3 Automation of task processes

2.3.1 Business processes

In complex model-driven software development processes, executing multiple model management tasks of different kinds is a common requirement. For example, before transforming a database model into code, we might want to verify that the model is well formed. In addition, model management tasks often need to be triggered in response to a modeling resource update, such as a model or a transformation being modified. For the different actors that need to understand how these tasks are related to each other and to other modeling resources, and for them to execute the required groups of tasks in response to resource updates, we need to introduce the domains of *Business Processes* and *Workflows*.

The International Standard for Systems and Software Engineering and Software Life Cycle Processes (ISO/IEC 12207:2008) defines a *process* as "a set of interrelated or interacting activities which transform inputs into outputs" [83]. A *business process* is used to represent, understand, and communicate how business-related activities must be carried out within an organisation. The Business Process Model and Notation (BPMN) standard defines a business process as "[a] defined set of business activities that represent the steps required

to achieve a business objective" [136].

The Workflow Management Coalition defines a *workflow* as "[t]he automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules" [196]. In other words, the automated aspects of a business process definition can be transformed into an executable process enacted by a workflow management system [196].

There are several aspects that characterise a process, in particular those defining the control-flow, data, and resource management viewpoints [159]. The *control flow* viewpoint determines the sequencing of activities. The *data* viewpoint describes the information visibility and interaction among workflow components. The *resource* viewpoint describes the way in which tasks can be allocated to human and non-human resources. The Workflow Patterns Initiative [159] provides a comprehensive description of common patterns for each of these workflow aspects.

The first part of this section introduces some of the most popular business process modeling languages. Then, build systems are introduced as tools that can support workflow management systems. Finally, the section ends with an overview of model management workflows in Model-Driven Engineering.

In the following we describe a set of popular tools/frameworks that are used for the orchestration and execution of programming tasks that have been used or have the potential to be used for model management tasks.

BPMN 2.0. The Business Process Model and Notation is a mature and widely adopted [108] international standard issued by the Object Management Group (OMG). This specification was built as a compilation of the best ideas and practices of the business modeling community including UML Activity Diagrams, Event-Process Chains (EPCs), etc.

YAWL. Yet Another Workflow Language (YAWL) [158] is an open-source workflow language based on formal Petri net foundations and designed to provide support for many workflow patterns (control-flow, resource, data, exception handling).

2.3.2 Model management workflows

In Model-Driven Engineering, workflows are understood as frameworks or tools able to define and execute model management operations in a predefined order. Diskin et al. [42] observed that megamodels can be used to process models in the form of abstract workflow languages. There are several tools that support the execution of model management workflows but only recently Kokaly [94] proposed switching their task-oriented paradigm into a declarative style that guarantees correctness by construction. Kokaly [94] proposes that

the specification of the inter-relationships between models is based on graphs, graph mappings, constraints and operations which can be composed to form complex chains of operations that can be parsed into a Directed Acyclic Graph (DAG) which verifies their correctness.

In the following we introduce some of the model-driven workflow frameworks and discuss their strengths and limitations.

MMINT [37] is an extensible and graphical model management tool for exploration and experimentation [37]. At its core MMINT builds a megamodel that is described at two levels of abstraction: the *type-level* where metamodels are interrelated through relationships and megamodel operators (e.g., filter, map, reduce, merge) [37] defining the relationships and operations allowed for models at the *instance-level*. From their definition model management operations are strongly typed and to execute them they must be invoked manually and individually. In this context, only relevant model loading activities are triggered when editor views are opened and when model management operations are invoked. Similarly, the strong and explicit typing of model relationships allows model management operations to produce trace links at model element level that become part of the megamodel. While the tool can be extended to support different metamodels, its scope is limited to EMF-based models.

MTC-Flow is a graphical tool that enables the definition and total or partial execution of chains of model management operations [3]. A workflow definition consists of the declaration of models and files that are consumed or produced by transformations (operations which use a model as input, output, or both). These chains of operations are executed by identifying the input resources of the workflow and invoking the tasks that consume them. When a task finishes its execution, it notifies that the output models and files are ready to be used by the tasks that use them as input. Before each operation is executed, validations may be performed on the models that it uses. In MTC-Flow, the workflow definition itself works as an explicit dependency graph. This tool supports a variety of model management tasks from different frameworks and its notion of a model is sufficiently abstract so that each task can implement its own model interpretation. However, for each task execution, models are created, loaded and disposed regardless of whether they are later reused by other tasks. An overview of the MTC-Flow metamodel is presented in Figure 2.12. Similarly, there are no validations to check whether an input or output file or model has changed from a previous execution to determine whether a re-execution is required. Regarding model management traceability, MTC-Flow does not seem to support it at any level.

Workflow+ is a theoretical modelling framework with a prototype that has been targeted to build safety cases [43] but has wider applications. Workflow+ uses a process-driven approach to model and analyse cyber-physical systems

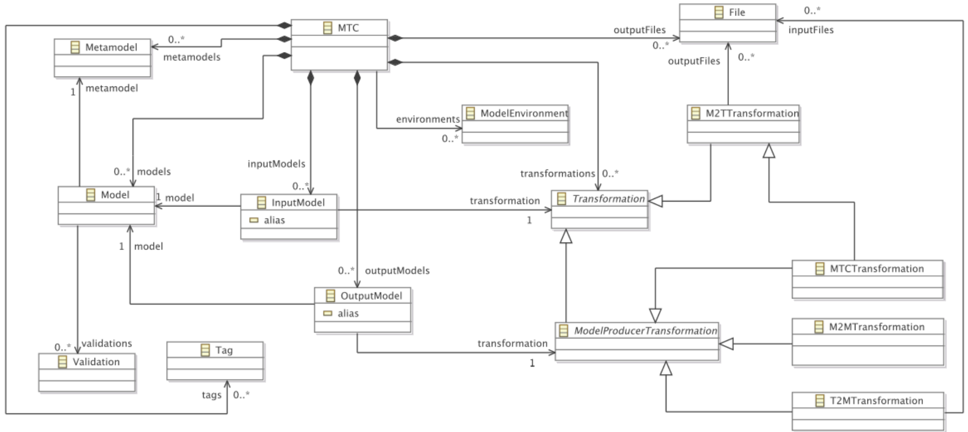


Figure 2.12: MTC-Flow metamodel (Image from [3])

and safety engineering processes with complex data flow and control flow [43, 4]. The framework is proposed as an alternative to GSN [91] that uses data to drive the argument flow. Workflow+ uses UML class diagrams to capture process and data definitions, control flow and constrains of data and processes, and traceability between model elements, dataflow, and process flow [43, 4]. Figure 2.13 illustrates an example Workflow+ metamodel describing a baking process and of the resulting workflow instance. In this figure, green boxes are processes, yellow boxes represent data, red lines are constraints, and green arrows represent dataflow to/from processes and black lines denote regular associations.

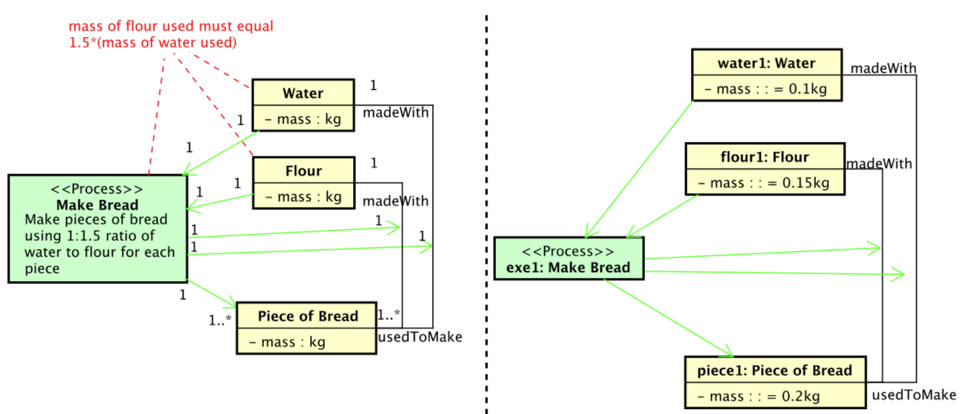


Figure 2.13: Example Workflow+ metamodel (left) and an instance (right) (Image from [4])

MWE2 [197] is a textual declarative workflow language and execution engine that allows the definition of tasks that read/write EMF resources, perform operations on them and generate artefacts from them. It is worth noting that MWE2 is a language designed to be used by the Xtext language generator

to configure itself. To the best of our knowledge MWE2 is only performs batch workflow executions, and is not concerned with the production of model management traces, or with dealing with non-EMF models. The execution life cycle of MWE2 consists of three phases: pre-execution, execution, and post-execution. At each of these phases, all tasks and sub-workflows invoke the method that corresponds to the phase in the order in which they are declared i.e., sequentially. This execution process is therefore not engaged with task interdependencies. Regarding model handling, MWE2 relies on explicit tasks to read and write EMF models.

Epsilon Workflows. The Epsilon modelling framework provides a family of interoperable languages and tools designed for specific model management tasks. Epsilon provides Ant Tasks [100] to execute each of its languages. The limitations of this workflow framework are bound to those of the ANT build system. While no dependencies can be established among model loading/disposal tasks or model management tasks, dependencies can be established through the Ant targets containing tasks.

ChainTracker was discussed in Sec. 2.2.5 as a traceability tool for model transformation analysis. However, ChainTracker also supports the execution of model-to-model and model-to-text transformations using ATL and Acceleo, correspondingly. We are not aware of any extensibility mechanisms to support more model management tasks and to our knowledge the workflow executions are performed in batch.

2.3.3 Build systems

Build systems or tools are used to automate the software compilation process. They are an integral part of Continuous Integration systems as they are often used to execute tasks jobs or steps in their pipeline. Mokhov et al. [125] describes a build system as a tool that "takes a task description, a target key, and a store, and returns a new store in which the target key and all its dependencies have up-to-date values". The *keys* and *stores* vary according to the build system, for example, keys are filenames when the store is a filesystem as in software build systems [125]. Task description are specifications that indicate how to compute new values for a given key.

According to Mokhov et al. [125], build tools can be compared based on several criteria including whether the build system is local or executes in the cloud; whether dependencies are known before the build (static) or are resolved as the build progresses (dynamic); the build determinism; whether tasks monitor changes to the task itself in addition to its dependencies (self-tracking); and whether the build can be stopped when outputs do not change (early cut-off). More importantly, Mokhov et al. [125] identifies two key design choices at the

core of build tools: the order in which tasks are built (scheduler) and whether a task can be rebuilt (rebuilder). Table 2.1 (influenced by [125]) shows some popular build tools classified by their scheduler, rebuilder, cloud-support and build activities (e.g., task execution, project configuration based on natures[3]).

Minimality. In addition to supporting the construction of outputs, build tools aim to be as efficient as possible. Mokhov et al. [125] recognises *minimality* as a guiding principle for build systems in which the build "executes tasks at most once per build, and only if they transitively depend on inputs that changed since the previous build". Some build tools refer to minimality as *incrementality* usually to indicate that only part of a build script needs re-execution after changes to source artefacts as other parts can be reused from a previous execution e.g., Konat et al. [107].

Correctness. Mokhov et al. [125] proposes a definition of correctness in which an acyclic and deterministic build is correct if it produces a *correct result* for any tasks, key, and store in which the *result* is the store obtained by running the build system with a given key, store and tasks. In other words, the result is correct if result and store agree on all input keys and if the value of all non-input keys stored in the result match one computed by the corresponding task [125]. For shallow builds the correctness is only needed for the target itself and the input keys reachable from the target, not those of their dependencies.

Schedulers. Mokhov et al. [125] distinguishes 3 types of schedulers among build systems. The *topological* scheduler pre-computes the order of the tasks before execution ensuring that when executed their dependencies will be satisfied. Computing this order requires the construction of an acyclic graph with the dependencies of a given key and then the iteration in topological order. The *restarting* scheduler interleaves the execution of tasks with their ordering. In this approach, tasks are executed and if out-of-date dependencies are found during their execution, the tasks are aborted and their dependencies are executed. The *suspending* scheduler follows a similar approach but instead of aborting, it suspends the task execution. Compared to the restarting scheduler, the suspending one supports *minimal executions* as it avoids re-execution of aborted tasks. In practice, the suspending scheduler is only better than the restarting only "if the cost of avoided duplicate work outweighs the cost of suspending tasks" [125]. The last two types of schedulers enable dynamic dependency discovery as opposed to the topological scheduler where dependencies must be known in advance.

[3]Project natures are an opinionated approach to configure a project. For example, a project configured with a Java nature might require a source and test folder in a specific (but configurable) location.

Table 2.1: Classification of existing build systems.

Approach	Scheduler				Rebuilder					Cloud		Build	
	Topological	Restart	Suspend	Parallelism	None	Dirty bit	Verifying Trace	Constructive Trace	Deterministic Constructive Trace	Cloud	Local	Tasks	Natures
Ant [56]	✔			✔	✔						✔	✔	
Bazel [64]		✔		✔				✔		✔	✔	✔	
Buck	✔			✔					✔	✔	✔	✔	
CloudBuild	✔			✔				✔		✔	✔	✔	
CloudShake			✔					✔		✔	✔	✔	
Excel		✔				✔					✔	✔	
Gradle [34]	✔			✔			✔				✔	✔	✔
Make	✔			✔		✔					✔	✔	
Maven	✔			✔	✔						✔	✔	✔
Ninja	✔			✔			✔				✔	✔	
Nix			✔	✔					✔	✔	✔	✔	
Pluto [53, 107]			✔				✔				✔	✔	
Redo			✔	✔			✔				✔	✔	
Shake			✔	✔			✔				✔	✔	
Tup	✔			✔		✔					✔	✔	

The challenges supporting the parallel execution of builds are dependent on the type of scheduler [125]. The most straightforward implementation is for the topological scheduler in which tasks can be started once their dependencies are completed. An approach to parallelise the suspending scheduler is to start multiple dependencies in parallel. For the restarting scheduler, parallelisation can involve the creation of as many threads as keys in the build and moving tasks with non-built dependencies to the end of a queue. The latter approach can lead to race conditions, but these can be mitigated by storing the build order across executions.

Rebuilders. Regarding rebuilders, Mokhov et al. [125] distinguishes 4 broad categories among build systems. The *dirty bit* rebuilder consists in saving information of a key (e.g., a timestamp) as a *bit* that encodes if it is dirty or clean. After an execution all bits are cleaned and key changes in subsequent executions are marked as dirty. This rebuilder only rebuilds what has changed including any affected transitive dependencies. The *verifying traces* rebuilder uses a *trace* to record values or hashes of the task and its dependencies, and only re-executes when these values have changed. To support this type of rebuilder, two operations are required: one for recording hashes in the trace, and another to verify them. Alternatively, the *constructive trace* rebuilder records actual values instead of their hashes. This type of rebuilder is useful for cloud build systems that can reuse results computed in other machines when local inputs are out-of-date. The last type of rebuilder is the *deep constructive trace* which in contrast to the regular constructive trace, it does not record values of dependencies. This rebuilder (also known as shallow build) relies on tasks being deterministic.

Build tool examples

This section describes the main features of some build tool examples and compares their dependency resolution procedure and whether they support *minimal* executions as defined by Mokhov et al. [125].

Apache Ant is a widely used build tool written in Java. A build definition in Ant is captured in an XML file and it starts with a root project which contains one or more targets. Each target defines one or more tasks which are sequentially executed. Ant does not statically declare file dependencies, instead it uses target inter-dependencies to compute its execution plan. As such, the ANT rebuilder does not support minimal executions by default [53]. To improve the efficiency of workflow executions, targets can be marked as conditional using the `uptodate` macro which checks whether a set of target resources are more up to date than their source to trigger a re-execution [53].

Gradle is also a task-based build tool, language, and dependency manager. In contrast to Ant, tasks in Gradle must not be contained in targets and they can directly depend on other tasks. Its build life cycle consists of three phases: `initialization`, `configuration`, and `execution`. After resolving project dependencies in the `initialization` phase, the `configuration` phase builds a graph of the tasks that are part of the build and computes which of them are required to be executed in the *execution* phase [127]. Gradle was designed to support minimal execution of build scripts. The task execution graph is not only influenced by task interdependencies but also by their inputs and outputs [127]. These values are typically evaluated at the `configuration` phase, but some inputs may be `evaluated` at the execution phase [127]. If the inputs of a task have not changed, it is considered up-to-date and skipped, otherwise it is executed [127]. In Gradle, properties of type file, directory or file collections can be declared as inputs or outputs, but properties of arbitrary nature such as strings can only be used as inputs.

Apache Maven is a build tool and dependency manager that favours convention over configuration. Maven configures the build process using one or more XML files called POMs. In contrast to ANT, Maven has three predefined lifecycles [57] which go through specific phases in a predefined order. For example, its default lifecycle includes the phases `validate`, `initialize`, `compile`, and `test`, in that order. Invoking any of those phases will implicitly call those that precede it. Custom tasks can be defined but they must be attached to a specific phase of a life cycle. Similarly, the *archetype* of a maven project defines different tasks which are executed by default at different phases of the life cycle. If more than one task is attached to the same phase, they are executed in the order in which they are declared. While some Maven tasks can execute themselves incrementality, the build execution is performed in batch.

GNU Make is a popular build tool that has been around for a long time. Make describes tasks and their dependencies in *Make files* with the form:

```
<provide> : <require>* <command>*
```

where `provide` is the file to be generated, `require` is the list of input files dependencies and `command` describes the task execution. A build is invoked by specifying a build target (any of the `provide` constructs) which triggers the recursive invocation of its dependent targets (any of the `require` constructs) if they have a more up-to-date value than last time the main target executed. Make uses the files timestamps to determine the *dirty* state of build targets and executes using a topological scheduler [125]. Other build tools like *CloudMake* and *Ninja*, manage dependencies in a similar fashion to Make [53]. These

characteristics allow it to perform minimal executions. However, Make has faced scalability challenges [125] as dependencies must be specified statically in the build file which also can lead to missing dependencies making the build unsound [53]. Furthermore, the use of the timestamp as indication of the dirty state of a file can lead to re-executions that are not really needed [53].

Shake is a build tool that aims to provide dynamic dependencies while still allowing for minimal executions [125]. It achieves this goal by using a suspending scheduler and by using a constructive trace that stores the dependency graph of the previous execution [125]. Like Make, Shake must statically declare the files that are provided but can discover and register dependencies at runtime allowing it to support incremental re-executions that consider these [53]. However, Erdweg et al. [53] indicates that there are three issues with Shake: that it is dependent on timestamps to determine the *dirty* state of files (irrespective of their contents); it requires clean builds when the build program is updated; and provided files cannot be computed, they must be statically declared.

Pluto is an incremental build tool that performs dynamic analysis to enforce invariants on its dependency graph [53, 107]. This graph connects file nodes with built units (i.e., tasks) through edges that indicate whether the build unit produces or requires the file [53]. The initial version of the algorithm *pluto* [53] interleaved dependency analysis with task execution. The *hybrid* version of the algorithm considered the full dependency graph traversal unnecessary in subsequent executions and proposed the use of file changes to only select potentially impacted tasks and check their consistency to decide whether to re-execute them [107]. To check whether a file is up to date, *pluto* uses the notion of *stampers* which are functions which take a file and produce a value or *stamp* based on some criteria such as its last modification date, contents' hash, or existence [53]. These *stamps* are saved in the edges between a file and a task in the dependency graph. Because of the stampers, Pluto offers minimal task execution. Pluto additionally supports cyclic executions if a resolution strategy is provided by the user.

Bazel [64] is a build tool developed by Google. Bazel maintains in the cloud a map of file hashes to file contents along with a log of executed build commands along with their input and output file hashes [125]. By using the build log, Bazel can predict the hash of a build result by examining the hashes of dependencies with matching inputs [125]. Additionally, if the computed hash is available in the map of file hashes to file contents, Bazel enables local machines to download the result and skip intermediate outputs [125]. Bazel supports dynamic dependencies only in built-in tasks for which it uses a restarting

scheduler [125].

2.3.4 Continuous integration

Software development processes have been moving away from waterfall methodologies into more agile processes that allow for rapid feature integration, error detection and release cycles. Continuous Integration (CI) is a software development practice that is used to support such processes, which has been shown to increase productivity [47]. Generally, the CI build process involves compiling and testing the code, creating executable artefacts from the source code, and any other tasks prescribed in the build process [60]. Usually, these tasks involve build tools like ANT or maven that are used to compile, test, and deploy code.

CI tools are normally linked to a Version Control System repository which developers work against. A CI build is triggered whenever the repository is updated. In a distributed team, CI tools are a centralised view of the project that can provide metrics (such as code coverage) of current and past builds and report to the team when builds fail. CI offers many advantages including reactivity, parallelisation and visibility of the process and state [59]. Example of popular CI tools include Jenkins, Travis, TFS, CicleCI and GitHub Workflows.

García-Díaz et al. [60] identified two issues while working with MDE projects in CI. One regarding the lack of model-driven tools integrated with CI tools. The other is that domain experts modifying the models were able to re-generate code artefacts and deploy to CI while other engineers that needed to modify the code could not do it without the help of the model experts.

Later Garcia and Cabot [59] proposed the invocation of MDE tools directly from the CI pipeline. [59] claims that if tools are executable in standalone mode, they can be integrated with the CI tools and provides examples of tools that can be readily integrated. In their paper, they use an example of a co-evolution scenario in which a metamodel is evolved and where the workflow involves change detection, impact analysis and co-evolution and testing before the execution of model-to-model and model-to-text transformations. While having the model management tasks defined in the CI would be useful for workflows that run in the CI, for those that also need to be executed locally this configuration might lead to duplication in the build/task configuration. While CI tools are meant to run in the cloud, build tools can be executed locally and in the cloud through the CI triggering the same results in both mediums, which is why they are commonly the main task in the CI processes.

2.4 Summary

The first part of this chapter introduced key concepts of Model-Driven Engineering. The definitions of models and metamodels were provided and then the classification of modeling languages and key components (semantics, abstract and concrete syntax) were provided. Later, this section introduced different kinds of model management operations such as model-to-model/text transformations, model validations and model comparison. Then, the notion of global model management and the differences between modeling in the small and modeling in the large were presented. This section finished with an overview of current MDE challenges such as scalability, co-evolution, heterogeneity, and interoperability.

The second part of this chapter introduced *Traceability* for software development. This section started with key terminology, classifications, and activities and then moved to discuss the traceability use within Model-Driven Engineering. The section finished with an overview of available tools that support traceability.

The third part of the chapter was an overview of business processes, build systems and MDE workflow tools. The section started with the terminology followed by the introduction of some of the most common business process modeling languages. Later, the section moved on to introduce and discuss some of the most popular and relevant build systems. Finally, the section concluded with an overview of some of the model management workflow languages used in the Model-Driven Engineering community.

3 Analysis and hypothesis

This chapter provides an overview of the limitations of state-of-the art build tools as MDE tools which motivate the proposition of a dedicated MDE workflow system, which is the main contribution of this thesis. After analysing the literature and identifying current challenges in Sec. 3.1, we present the research hypothesis, objectives, and scope in Sec. 3.2.

3.1 Analysis

General purpose business process and workflow languages such as BPMN [136], YAWL [189], UML Activities [46] can be used to capture model management processes. These tools are specialised to capture processes where control flow, resource interactions and data flow are precisely specified. As such they offer a wide range of control flow (e.g., branching, multi instance synchronisation, state-based triggers, iterations), data flow (e.g., between task and environment, push or pull strategy) and resource (e.g., authorisation, role-based access, distribution, selection) patterns. To take advantage of the previous patterns, these business process tools, and workflow languages are best suited for complex workflows that need to be precisely defined and orchestrated.

In contrast to business process and workflow languages, build systems which consist of a set of tasks executed to achieve a specific target, have a more limited range of available control flow patterns while often offering more efficient executions. The limited set of control flow patterns available to tasks in build systems is simplified to either task interdependencies (e.g., *dependsOn*, *after/before*) or build phase attachment (e.g., “compile phase”). In other cases, like Make, the dependencies are specified at file level rather than task level, for example: *fileC* requires *fileA* and *fileB*. Overall, build systems can be seen as domain specific workflow execution engines that offer a narrow set of control patterns to the users that are sufficient to achieve a successful build.

In addition, build systems have become more popular as the complexity of software development increases. As such, more build tools are available and these are increasingly efficient, reducing overall build times. The strategies that have facilitated this include the establishment and resolution of task and resource dependencies and the use of execution traces or dirty flags to ensure that the builds are only re-executed for the items that change. For example,

build tools such as Ant [56], Gradle [34], and Pluto [53] have the ability to check for changes in input resources to determine which tasks need re-executing. These tools use different mechanisms to identify resource changes that enable partial executions that respond to changes in the resources. For example, Pluto allows the user to indicate how to determine if a resource is up to date, while Ant may use either a file checksum or timestamp and Gradle will use exclusively file checksums or string values[1]. However, despite its flexible change detection criteria, Ant relies on excessive specification of the `uptodate` check in build definitions [53] to identify the resources that changed. More recently execution traces and generated artefacts can be shared in a remote location so that if a local item of a user matches a remote item built by a team colleague, the user can download the previously built artefact rather than rebuilding it locally.

Another approach that can be used to support model management workflows is megamodelling. This approach requires the specification of artefact relationships at a more specific level than how they are consumed (e.g., read/write) or their dependencies, possibly indicating the logic of the operator that binds them. For example, artefact relationships may be of type *matches*, *transforms*, *slices*, etc. Evidently, this requires models and model operators to be precisely (and even formally) specified to chain operations into a workflow. Because of its precise construction, megamodelling has the advantage of producing model management traces as a side product of an operator execution. Currently, MMINT is an example of such a megamodelling tool, although it does not support workflow executions as it expects the user to manually trigger operators on selected model resources of a project.

There are several dedicated tools that support model management workflows such as ChainTracker [69], MTC-Flow [3] and MWE2 [197]. ChainTracker is primarily a model management trace analysis tool, but it also supports the execution of chained model-to-model or model-to-text transformations. It is unclear how complex these workflows can be as many examples include at most two tasks in the trace analysis. Similarly, because workflow execution is not its primary focus, it is anticipated that executions are performed in batch. Another tool is MTC-Flow which supports model management workflow executions and captures the workflows in a graphical domain specific language. While MTC-Flow supports multiple model management tasks, it only supports EMF/Ecore models. One key functionality of MTC-Flow is support for alternative execution paths, which is a feature that is rarely available in build systems. However, MTC-Flow also executes workflows as batch processes (disregarding optimisations seen in build systems) and loads and disposes models before and after the execution of each task that uses them. One of the missing features of this tools is the lack of traceability produced as a side product of the workflow execution. Another

[1] For non-file-based properties

tool that can execute model management workflows is MWE2, a declarative and extendible textual workflow language and execution engine. Just like MTC-Flow, MWE2 can define workflows involving arbitrary model management tasks although it mostly handles EMF models. Similarly, no traceability can be produced as a side product of the execution. However, in contrast to MTC-Flow, MWE2 does not automatically read or write models before and after they are used, but rather expects the user to use dedicated loading and disposal tasks before and after a model management task. Another difference with MTC-Flow is that MWE2 can only define workflow tasks sequentially and execute them in the order that they are defined.

Models can be handled by multiple modeling frameworks. For example, MATLAB is used to manage Simulink, Requirements and Dictionary models, while EMF models are commonly manipulated in Eclipse. Additionally, MATLAB has its own set of model management tools for its own models while open-source model management frameworks such as Epsilon can support a multitude of modeling formats like spreadsheets and databases. However, most model management frameworks such as ATL tend to support EMF models exclusively. An important aspect of a model management workflow framework is the ability to support heterogeneous modeling formats and model management tasks. As such, extensibility is an important feature. Most of the dedicated model management workflow tools described above can support multiple model management tasks. However, as most of the model management languages and tools, they seem to only support EMF models.

Considering the improved efficiency efforts, omnipresence, and popularity of build tools, it is no surprise that there have been attempts at integrating model management tasks with these. Examples of these integrations include Epsilon and ATL tasks in Ant and more recently Epsilon tasks executed from Gradle. One of the challenges in these integrations, as in MWE2, is that model loading and disposal are usually separate tasks. While this in itself is not a problem, the user is left to handle how model management tasks will use the models (read/write or both) and this configuration may change based on how the tasks are invoked (e.g., an arbitrary *target* in ANT, a single or a group of tasks in Gradle). But more importantly, models are also dependencies of model management tasks, that is, if a model changes as a result of a previous model management task, the next task that uses it may need re-execution even if the model management program remains the same. By not considering model resources as dependencies of model management tasks (but rather as dependent model loading/disposal tasks), build systems could skip model management tasks executions when the models they use change. Moreover, model loading/disposal tasks cannot be skipped if the resources do not change as they must always be executed in case any model management task needs

them. A related challenge is that build systems are not supposed to modify any of their inputs while in model management workflows models are often used as both inputs and outputs. According to [125], a build tool cannot be correct if inputs are modified. We argue that if the interpreter keeps track of these models and of their use by different tasks in the workflow, it can ensure that subsequent executions are correct if the model is compared against its latest state from the previous workflow execution. To do this, task outputs should be tracked, and yet, only some build tools do this, and they differ in *how* they are used by the engine and *when* are they tracked. For example, Gradle can handle outputs that are known before a task execution, Pluto only knows about them when a task is finished (which may trigger a task reordering) and Ant does not track them at all. Another challenge in build systems is the generation and maintenance of model management traces. While integrated tasks can sometimes produce this information individually, build systems have no interface to collect or homogenise them because it is not their intended purpose, although global accessible variables could be used to capture these during an execution.

In summary, there are three key areas where we consider that model management workflow engines could be enhanced:

Conservative executions. Engines supporting the execution of model management workflows should have some of the cutting-edge features seen on build systems like minimal executions (as defined by Mokhov et al. [125]). To support these in model management workflows, the engine must trigger executions that are consistent with the impact that model and other resource changes have on the different tasks. For example, if the user changes a model-to-text transformation program that follows a model-to-model transformation, it would be desirable that the workflow only executes the model-to-text transformation task as nothing else would be affected. Additionally, the interpreter needs to take into consideration the fact that models may be used as input and output by some tasks and handle this appropriately to ensure correct executions.

In this work we use the word *conservative* to describe workflow executions that only execute tasks transitively affected by external changes to workflow resources and where these resources are seen as a black box. The word *conservative* is related to the concept of incrementality described in Sec. 2.1.7 as they both aim to reduce redundant computations if changes in resources do not affect the outcome of an execution. However, in MDE incrementality is usually employed in the context of an activity such as a model transformation or a model querying program. As such, as opposed to any type of MDE incrementality that is observed on a single model management activity, a conservative execution refers to workflow unit incrementality i.e., across tasks in a workflow. Similarly,

a conservative execution is also related to the minimality definition presented in Sec. 2.3.3. However, the word *minimality* may suggest that a minimal workflow has effectively no further optimisations available. As such, a conservative execution does not make such a suggestion as further optimisation are indeed possible if workflow resources were to be monitored at a finer level of detail (e.g., changes to model elements or lines in a file) not as a black box.

Context-aware model loading and disposal. Model loading and disposal is an important activity that is often required in model management workflows. However, current model management workflow tools either load and dispose the models on each task that executes them or load the models at the start of the workflow execution and dispose them at the end. The first strategy may incur loading and disposal overheads particularly when the same model is reused in multiple tasks, while the latter approach may incur memory issues when multiple models are involved in the workflow as these consume memory resources during the whole execution. Similarly, in build systems it is expected that the user will handle the loading and disposal strategy which may not be efficiently handled by their conservative execution mechanisms as dependencies between models and tasks may not be clearly specified. Because large models can be slow to load and memory-intensive [101], we argue that they should be loaded only when needed by the tasks in the workflow. Likewise, to free up memory, it is also important to dispose them as soon as they are no longer useful. For example, if a model is to be reused by several tasks, it should not have to be reloaded in between them. Similarly, if a model is only used once, it does not need to remain loaded during the execution of other tasks in the workflow. A context-aware model loading and disposal strategy should know if a loaded model needs to be retained in memory to be reused by another task or if it can be safely disposed of.

Model management traceability. Traceability is an important feature that enables impact analysis, regression testing and system understanding while also being sometimes demanded by certifying authorities in safety critical systems. While traceability is often a by-product of model management languages and tasks, it is rarely offered as a side product of model management workflows. To our knowledge, only ChainTracker [69] offers end-to-end traceability for its workflows, while none can recover traces. We consider traceability to be a missing feature of model management workflow engines. These should provide trace collection and recovery mechanisms that collate traces in a structured and analysable format that is maintained across executions. This would allow workflow users to determine model and program coverage across the workflow, debug programs in the context of the workflow, and to assess the impact of

model, program and template changes in the rest of the workflow artefacts.

3.2 Research overview

This section introduces the research hypothesis in Sec. 3.2.1 and its objectives in Sec. 3.2.2. To narrow down the breadth of the research Sec. 3.2.3 clarifies its scope.

3.2.1 Hypothesis

> *The performance of repetitive executions of **model management workflows** can be significantly improved with the help of a **conservative** interpreter that consumes declarative workflow specifications capturing dependencies among **models** and **model management tasks**. At the same time, these inter-dependencies can be used to establish and maintain **traces** at model element level of granularity.*

The highlighted terms are defined as follows:

Model management workflow: A set of model management tasks (e.g., model validation and model-to-model/text transformations) to be executed in such an order that respects their dependencies.

Conservative: Executions where only tasks transitively affected by a set of external changes to workflow resources are executed, and where affected resources are considered as black boxes.

Models: Heterogeneous artefacts that represent a domain in a structured way and which may need to be loaded to memory and disposed from it.

Model management tasks: Heterogeneous activities that manipulate models such as transformations and validations.

Traces: Semantically-rich relationships among resources such as models, model elements, model management programs, and files.

3.2.2 Objectives

To assess the validity of the research hypothesis, the following research objectives were defined:

(i) Development of a prototype of a MDE workflow system, herein referred to as ModelFlow, able to be conservatively executed and produce model management traces as a side product of its execution.

(ii) ModelFlow must be able to accommodate diverse MDE workflows. As such, an evaluation of the ability of the prototype to capture and execute multiple MDE workflows will be performed.

(iii) To determine the correctness of the workflow dependencies and execution, an evaluation of these based on different workflows and resource modification scenarios will be provided.

(iv) To determine if ModelFlow performance is adequate, this will be compared against executions with a sample of general-purpose build tools.

(v) Assess the overhead of the framework features, including the production of model management traces and automated model management.

Derived objectives to support objective (i)

(i.i) ModelFlow must be able to handle heterogeneous model formats. As such, an extensibility mechanism to support multiple modelling formats will be offered.

(i.ii) ModelFlow must be able to support the execution of heterogeneous model management tasks. As such, an extensibility mechanism to support multiple model management tasks will be offered.

3.2.3 Scope

The research objectives are bound to the following research scope:

1) Regarding objective *(iii)*, to determine the correctness of a workflow execution we will carefully select relevant change scenarios for a workflow rather than an exhaustive list of scenarios covering all possible change combinations.

2) Regarding objective *(iv)*, it may not be possible to compare the solution against all available build tools therefore the comparison will be carried out with a selection of those most used for model management (e.g., Ant) and general purpose programming (e.g., Gradle).

Areas that are out of the thesis scope are described below.

Task parallelisation. While it is the next recommended optimisation, it is not our goal to support parallel executions in this work. However, taking into consideration this optimisation, we shall implement ModelFlow in such a way that adding parallelisation does not require a complete redesign. In practice, this feature motivates the decision to adopt a topological scheduler as it facilitates parallelisation.

Incrementality. This work attempts to support conservative MDE workflow executions, that is, workflows that only execute the tasks transitively affected by a set of workflow resource changes. While the term conservative is related to incrementality because both aim to reduce redundant computations, in MDE the term incrementality is generally used within tasks and not within workflows. As discussed in Sec. 2.1.7, there are many types of incrementality for model management activities like model transformations and model querying, but

in general they are used in the context of a single model management task. In this work tasks are considered a black box that may or may not execute incrementally.

Collaboration and Continuous Integration. In this work it is not our goal to explore how to support collaboration across fragmented and distributed teams that use an MDE workflow. In practice, build tools are commonly executed locally and independently from other users and that is the convention we shall follow. Similarly, while it is our goal to integrate ModelFlow with other build tools, it is not our goal to directly integrate it with any Continuous Integration (CI) tool. Build tools can usually be executed within CI tools, as such, by integrating ModelFlow with a popular build tool its indirect integration with CI tools is possible.

Business process. We argue that much like other build tools, an MDE workflow can be orchestrated and executed based on task interdependencies. That is, the workflow can be scheduled by resolving these dependencies. While build processes provide an alternative that prescribes the control flow of a workflow execution it also provides a much richer set of control flow patterns (e.g., choice, merge, synchronise, etc.) and resource patterns (e.g., role based distribution, authorisation, etc.). While building a workflow using a business process notation could provide a precise and detailed prescription of the workflow, it would also require users to develop a deep understanding of the process constructs (of control and resources) which may be more than the subset that a user usually needs (e.g., a "dependency" relationship between tasks). It is not our intention to create a prescriptive tool that captures workflows at this level of detail but rather to support *generic* MDE workflows and integrate them to regular software build process.

Authentication and authorisation. As in the previous point, we assume that a user should be able to execute the workflow completely, that is, tasks are not split across users. As such user authentication and authorisation to execute tasks or modify resources is out of this work's scope.

Replacing current build tools. Model management tasks are only a subset of all potential tasks to be used in a build script. Attempting to develop a complete replacement for a build tool such as Gradle, Maven, Make or Ant would be an unrealistically ambitious task. As such, in this work we attempt to provide a complement to current build tools that is tailored for model management workflows.

Scalability. This work attempts to provide a solution that can execute model management workflows but also do so conservatively. Evidently some of the features that enable the conservative execution may incur a slight overhead compared to a batch execution. However, our goal is to keep this overhead from being significantly time consuming compared to a batch execution while providing significantly better performance in scenarios where only partial re-executions are required.

4 ModelFlow: A model management workflow framework

ModelFlow is a prototype for specifying and executing multi-step workflows involving model management tasks. It consists of a textual language for specifying model management tasks and their dependencies, and an interpreter that can conservatively execute such workflows based on changes made to relevant artefacts (e.g., models, model management programs, generated files). ModelFlow also supports context-aware model loading and disposal, and offers end-to-end traceability. The prototype is in active development under the EpsilonLabs GitHub organisation[1].

4.1 ModelFlow's features

A workflow can be described as a series of tasks that are executed in a predefined order. A model management workflow involves model management tasks that interact with models that need to be loaded and disposed of at some point during the execution (e.g., MTC-Flow). ModelFlow is a model management workflow language and interpreter with the following characteristics.

4.1.1 Declarative workflow

Many workflow languages and model management languages are declarative to allow users to focus on *what* they need to do rather than on *how*. As such, we decided to use a declarative language to capture model management workflow specifications as described in Sec 4.3. In the case of model management workflows, the user will specify which models are used by different tasks and the manner in which they are used (e.g., input, output, or both), along with the tasks that must be executed beforehand. By placing the focus on how these elements interact with each other, the engine can propose an appropriate execution plan in an order that satisfies these relationships using a scheduler. This enables the user to focus on capturing dependencies between workflow elements rather than on the execution order.

[1] https://github.com/epsilonlabs/modelflow

4.1.2 Conservative executions

Conservative workflow executions only execute tasks transitively impacted by an external change to resources in the workflow (e.g., source programs, consumed or generated models). The more complex and time-consuming model management workflows become, the more benefit a conservative execution can bring. To support conservative executions ModelFlow records a *stamp* of the inputs and outputs of each task (including models) in an execution trace (Sec. 4.4.3). Conservative executions have the benefit of potentially executing a smaller subset of tasks than the original execution, which may reduce the time of the overall execution. However, processing inputs and outputs to determine if a re-execution is necessary produces an additional overhead compared to a batch execution. In practice, this overhead may be less noticeable when individual task executions are time-consuming. Similarly, if all tasks are equally time consuming, then the benefit of conservative executions grows as the number of affected tasks reduces.

4.1.3 Automated model management

In ModelFlow models are loaded when first needed by tasks in the workflow and disposed when no longer needed. In practice this model remains loaded as long as there is use for it through different tasks in the workflow. Model loading has been highlighted as a performance bottleneck when large models are involved [101]. As such, ModelFlow has been designed to minimise the invocation of model loading and disposing procedures. By disposing models as soon as they are no longer needed, the memory is freed of a resource that is no longer necessary. Similarly, by loading models only when first required the process avoids loading all resources before they are actually needed. This strategy is useful when different models are needed at different stages of the workflow. When models are used in throughout the workflow execution or at the start and end of the workflow, ModelFlow can reduce the overhead of the loading and reloading procedures by keeping models in-memory throughout the execution.

4.1.4 Model management traces

It is important to support the collection, creation and maintenance of model management traces in a workflow as it allows users to determine model and program coverage across the workflow, debug programs in the context of the workflow, and to assess the impact of model, program and template changes in the rest of the workflow artefacts. In addition, several standards, certifying authorities and government guidelines demand traceability (Sec. 2.2.2). As many model management tasks already produce traceability as a side-product

of their execution, the new feature to be incorporated to the architecture is the recovery of side-product traces and the ability to create new traces in those tasks that do not produce any by default. As such, overarching traces are recovered during the execution in a standardised format (model) that can be persisted and reused. The declarative specification of the workflows allows ModelFlow to map traceable artefacts (e.g., models and tasks) in the produced traces to the declared artefacts in the workflow. By building and maintaining this traceability information, users can answer traceability queries. The standardised format in which these are produced, allows users to process this information in arbitrary tools. Evidently, keeping traces incurs an additional performance overhead and requires models to be kept loaded for longer while the workflow management trace is updated. Additionally, not all workflows require the collection of traces for analysis or any other purpose. As such, this feature has been made optional and is only enabled on demand.

4.2 Architecture

Figure 4.1 provides an overview of the architecture of ModelFlow. ModelFlow consumes a workflow specification captured in a declarative domain specific language. The workflow specification declares models and tasks, while tasks explicitly state which models are required or produced by them. Both models and tasks can be configured within their declaration (e.g., `src` parameter) according to the parameters accepted by their definition type (e.g., `emf` or `etl`).

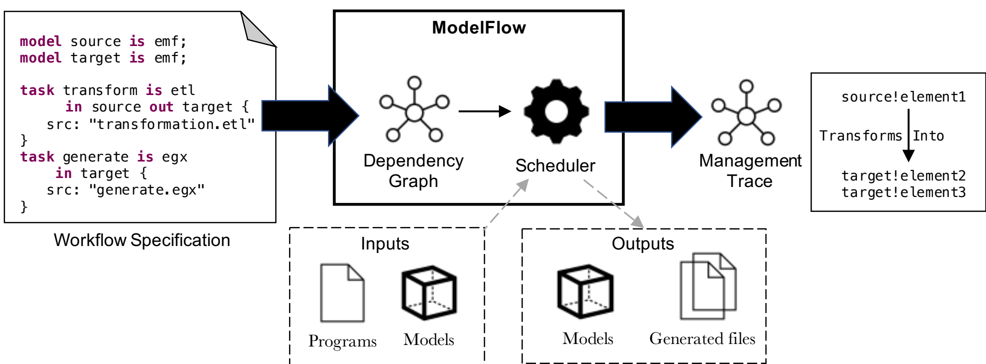


Figure 4.1: ModelFlow architecture

When the workflow specification is consumed by the interpreter, it is translated into a dependency graph that captures model and task interdependencies. This graph is used by the scheduler to determine how to execute the tasks in the workflow. Task executions may implicitly require model management programs and generate other files that are resolved as inputs or outputs by the task at runtime. This information is used by the ModelFlow to determine if future executions of the tasks will be required. Additionally, ModelFlow collects

and aggregates model management traces that tasks produce and share during their execution (e.g., traces of a transformation linking model elements created in a target model from model elements in a source model). In addition to any generated models and files, ModelFlow may produce as output the aggregate of the management traces generated during the workflow execution.

4.2.1 Components

ModelFlow is composed of the components shown in Figure 4.2 which are described below.

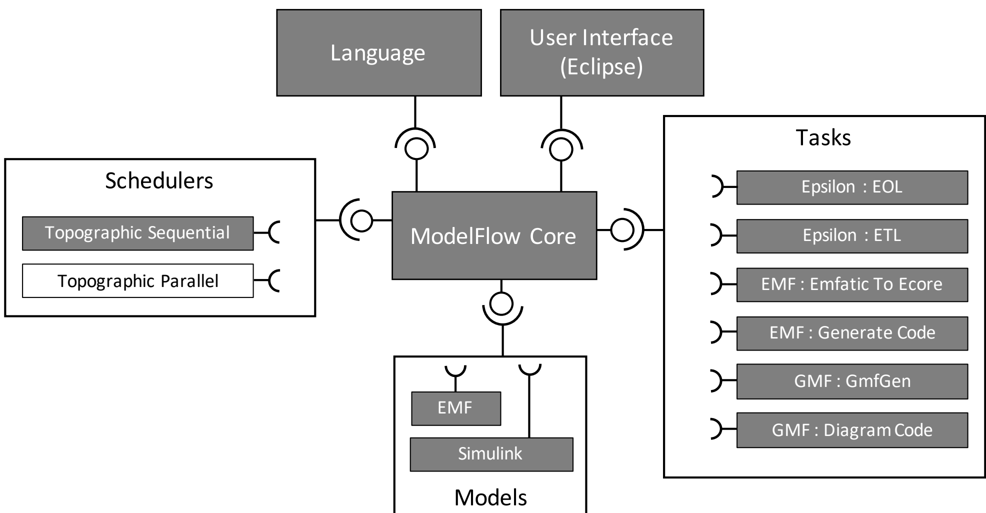


Figure 4.2: ModelFlow component diagram

Model Components. These components provide an interface to manage model technologies. Examples of such components include:

epsilon:emf: Supports Epsilon EMF models.
epsilon:simulink: Supports Epsilon Simulink models as described in Sec. 5.6.

Task Components. These components provide an interface to manage model management tasks. Examples of such components are listed below. Note that ModelFlow does not provide model loading/disposal tasks (as Epsilon tasks in Ant) since these operations are performed automatically by the engine.

epsilon:eol: Executes an Epsilon imperative program.
epsilon:etl: Executes an Epsilon model-to-model transformation program.
epsilon:egl: Executes an Epsilon model-to-text transformation program.

epsilon:egx:	Executes an orchestrator of EGL programs.
epsilon:eml:	Executes an Epsilon merge program.
epsilon:ecl:	Executes an Epsilon comparison program.
epsilon:flock:	Executes an Epsilon migration program.
epsilon:eunit:	Executes an Epsilon model testing program.

Scheduler Components. These provide alternative execution mechanisms to execute the workflow (see Sec. 4.4.1). Currently only the topological sequential scheduler is fully supported although there is an experimental version of a topological parallel scheduler.

Language Component. This component can parse a workflow definition and resolve the dependency graph and execution plan required to execute the workflow. Currently ModelFlow uses an ANTLR based concrete syntax that extends EOL's.

Core Component. This component is responsible for the execution of the workflows and is the one that organises all the other components. In particular, this component is responsible for resolving the dependency graph and proposing an execution plan for the workflows. Additionally, it orchestrates the model loading, saving and disposal while also maintaining the execution traces that support the conservative executions and the model management traces that support end-to-end traceability.

UI Component. This component provides tools to support the creation and execution of model management workflows. It provides views for the dependency graph, a runtime configuration to execute the workflows and some compile-time validations.

4.3 Language

As discussed in the background a domain specific language is described in terms of its abstract syntax i.e., the metamodel of the language; its concrete syntax i.e., a graphical or textual representation that is used to create an model instance with the constructs and relationships defined in the abstract syntax; and its semantics i.e., the context and reasoning behind the elements defined in the abstract syntax along with the rationale for the allowed and forbidden links among them. Note that in the case of executable DSLs like ModelFlow, the semantics of the language must include its behavioural specification. This section describes all these aspects.

To distinguish between the kinds of elements involved in the construction and execution of tasks and models alike we will use the terms:

- *Rule* or *Declaration*: an item defined by the user using a concrete syntax which indicates how tasks or models are to be configured and related. A rule or declaration shall be later compiled and resolved as an executable instance.

- *Definition* or *Interpreter*: an implementation that defines how to execute task or model instance of a given *type* e.g., an `EOL` task.

- *Instance*: a runtime item that has been configured based on the information from its declaration and workflow context which can be executed according to the implementation provided by its definition type.

To clarify the use of these terms we provide the following example. Listing 4.1 shows a task *rule* that *declares* two task *instances* (task $A\{x = 1\}$ and task $A\{x = 2\}$). Each of these instances is of *type* `epsilon:eol`, which is a task *definition* that can execute an EOL program from the configuration of the task *instances*.

```
1  task A is epsilon:eol forEach x in Sequence{1..2}{
2    src: x.asString() + ".eol"
3  }
```

Listing 4.1: Example of a task declaration.

Sec. 4.3.1 presents the abstract syntax of the language and Sec. 4.3.2 its concrete syntax. Sec. 4.3.3 introduces the metamodel used to represent the resolved workflow. Finally, Sec. 4.3.4 describes the semantics of different language elements in terms of their execution.

4.3.1 Abstract syntax

In ModelFlow, workflow specifications are organised in `ModelFlowModule` modules (see Figure 4.3). As `ModelFlowModule` extends `EolModule`, it can contain user-defined operations and import other EOL library modules and ModelFlow modules. A ModelFlow module also contains a set of task, model, and parameter declarations along with `pre` and `post` blocks that are inherited from the `ErlModule`.

ConfigurableRule: An abstract type that is used to configure task and model declarations. Each configurable rule has a `name`, a definition `type` and may contain a set of `parameters` to configure itself. The parameters are a list of key-value pairs. The definition type must know how to process each of the parameter keys. Just as well, the parameter's value must be in a format that can be processed by the definition `type`. Parameter values can be in the form of statements or expression that can be evaluated at runtime.

ModelDeclaration: A model declaration specifies a single model that may be consumed, modified, or produced by an arbitrary number of tasks in the

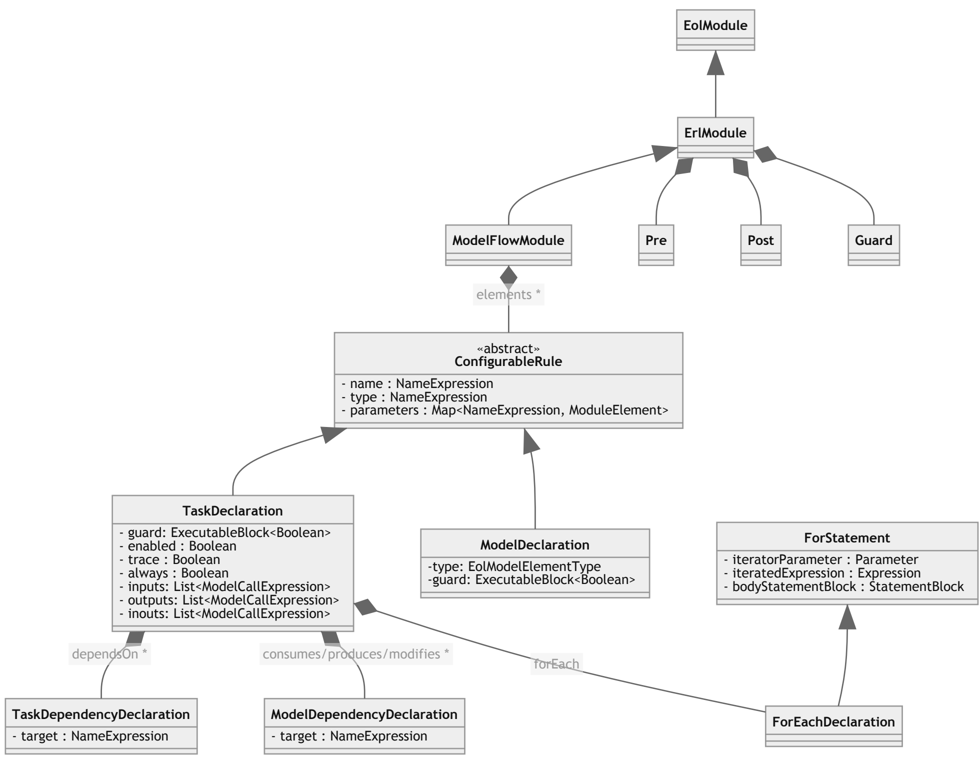


Figure 4.3: Class diagram of ModelFlow's abstract syntax

workflow. This element inherits all its configuration from a `ConfigurableRule`, that is, name, definition type and parameter list. In the case of its type, this must correspond to a model definition type.

TaskDeclaration: A task declaration specifies an atomic or multi-instance task that shall be executed. This element inherits all its configuration from a `ConfigurableRule`, that is, name, definition type and parameter list. In the case of its type, this must correspond to a task definition type. A task definition may declare a list of `ModelCallExpressions` to indicate which models it consumes, produces, or modifies. Task dependencies can also be declared through a list of `TaskDependencyDeclarations` Each task can optionally define a `guard` which determines whether the task should execute. Tasks that shall build multi-instance tasks can use the `ForEachDeclaration` to provide a collection to use to configure these tasks.

ModelDependencyDeclaration: A model call expression indicates a dependency between the declaring task and a model resource. Each model call expression can declare an `alias` for the model being referenced.

TaskDependencyDeclaration: A task dependency declaration holds the name of a list of tasks that the declaring task depends on. In other words, the list of dependency tasks must be up to date for the declaring task to execute.

ForEachDeclaration: This construct represents a for-loop iterator that creates task instances for each of the iteration parameter values.

Guard: Guards determine whether a task should be allowed to execute. In the case of multi-instance rules, the guard is evaluated on each of its atomic tasks.

Pre and Post: A ModelFlow module can define *pre* and *post* blocks that execute EOL statements before and after the tasks' execution, respectively.

4.3.2 Concrete syntax

We now present the concrete syntax of ModelFlow that is constructed using Epsilon/ANTLR facilities. Just as other Epsilon languages, the concrete syntax of ModelFlow is hybrid as it contains a mix of declarative constructs and imperative code (in the form of EOL expressions and statements). ModelFlow accepts model and tasks declarations but allows imperative code to configure their properties and to initialize and terminate the workflow setup.

Model resource declaration

Listing 4.2 provides the concrete syntax for a model declaration. This declaration starts with the `model` reserved word followed by the name of the model (`<Name>`). Then the model type definition follows (`<Type>`) which shall be used to match the model with a model definition. Inside the curly brackets multiple parameters can be declared to configure the model. Alternatively, if no parameters are required, the model declaration can omit the curly brackets and close with a semicolon.

```
1 model <Name> is <Type> ({
2   (<Parameter.Key> (:expression|{statementBlock}))*
3 }|;)
```

Listing 4.2: Concrete syntax of a model resource declaration.

Listing 4.3 illustrates an example of a declaration of a model of type *epsilon:emf* named *GenModel*. The model definition type *epsilon:emf* requires the model file (`src`) and the metamodel (`metamodelUri`) and expects their values as Strings. In the example, the `src` parameter is provided as a String statement while the `metamodelUri` is provided as an executable block that after evaluation returns a String.

```
1 model GenModel is epsilon:emf {
2   src : "workflow.genmodel"
3   metamodelUri {
4     return "http://.../emf/2002/GenModel";
5   }
6 }
```

Listing 4.3: Model resource declaration example.

Task declaration

The concrete syntax for a task declaration is specified in Listing 4.4. The declaration starts with the `task` keyword followed by the name of the task (`<Name>`). Then follows the task definition type (`<Type>`) preceded by the `is` word. In addition to the list of parameter declarations (as in Model Declaration), inside the curly brackets a task may also declare a guard. The task declaration may also list task dependencies (*TaskDependencyDeclaration* in the abstract syntax) using the `dependsOn` construct followed by the names of the tasks separated with the `and` keyword.

```
1  (@disabled|@noTrace|@always)
2  task <Name> is <Type>
3    (in(?)? <ModelCall> (and <ModelCall>)*)?
4    (inout(?)? <ModelCall> (and <ModelCall>)*)?
5    (out(?)? <ModelCall>(and <ModelCall>)*)?
6    (dependsOn <TaskName> (and <TaskName>)*)?
7    (forEach <iterationParam> in (expression|{
        statementBlock}))?
8  ({
9    (guard (:expression)|({statementBlock}))?
10   (<Parameter.Key>(?)?(:expression|{statementBlock}))*
11 }|;)
```

Listing 4.4: Concrete syntax of a task declaration.

Additionally, the task declaration can specify which models shall be used as `input`, `output` or `inout` by providing `ModelCall` elements accordingly (lines 3-5). For any of these model calls (*ModelDependencyDeclaration* in the abstract syntax) it is possible to specify a list of aliases to be used. The concrete syntax of each of these elements (`<ModelCall>`) is defined (Listing 4.5).

```
1  <Model.Name> (as <Model.Alias>)?
```

Listing 4.5: Concrete syntax of `<ModelCall>`.

The generation of multiple task instances from a single task declaration is possible using the `forEach` construct (line 7 in Listing 4.4). This mechanism uses values from a collection to configure the individual task instances. The collection is declared as a statement or executable block and the iteration variable can be used to configure the task parameters of the different task instances. An example of such a multiple task declaration is provided in Listing 4.1. In this example, the iteration variable `x` is used to configure the task's `src` parameter which takes the values of `1.eol` and `2.eol`.

Additionally, a task declaration may be annotated to configure its runtime behaviour. The `@disabled` annotation ensures this task is not executed. In contrast the `@always` annotation ensures the task is always executed regardless

of whether its inputs and outputs are up-to-date or not. Furthermore, a task may be annotated with `@noTrace` to indicate that its model management traces are not to be recorded in the workflow's model management trace. For example, tasks that serve as utilities rather than as process steps can omit being traced. This annotation only works when the workflow being executed has been flagged to record traces.

An example of two tasks is presented in Listing 4.6. The first task named `gencode` uses the task definition type `emf:generateCode` which does not require any parameters to be configured but does require an input model such as GenModel. The second task named `transformGenModel` of type `epsilon:etl` uses two models as inputs and one as output. Note that the input `ECore` model also indicates an alias to be used in this task only. The task also indicates a dependency with another task with name `validation`. The task configures its `src` parameter with a specific value but also uses a guard to evaluates that the source file exists to be allowed to execute.

```
1  task gencode is emf:generateCode in GenModel;
2
3  task transformGenModel is epsilon:etl
4      in ECore as Ecore and GenModel
5      out Gmf
6      dependsOn validation
7  {
8    guard : self.src.exists()
9    src : "mmop/validate.etl"
10 }
```

Listing 4.6: Task declaration example.

4.3.3 Workflow metamodel

This section describes the workflow metamodel. The outcome of the language compilation is a model that conforms to this metamodel. The abstract syntax of the language is translated into the metamodel presented in Figure 4.4.

Named: This abstract class is used to capture the name of configurable items.

Configurable: An abstract class that represents an item that needs to be instantiated based on a `definition`. It also points to a Java object that represents the Epsilon module element that declared it. Elements that inherit from this class contain a series of property elements that are used to configure them.

Property: This element represents a `key-value` pair used to configure tasks and resources. The key of this element is a string, while the value is represented

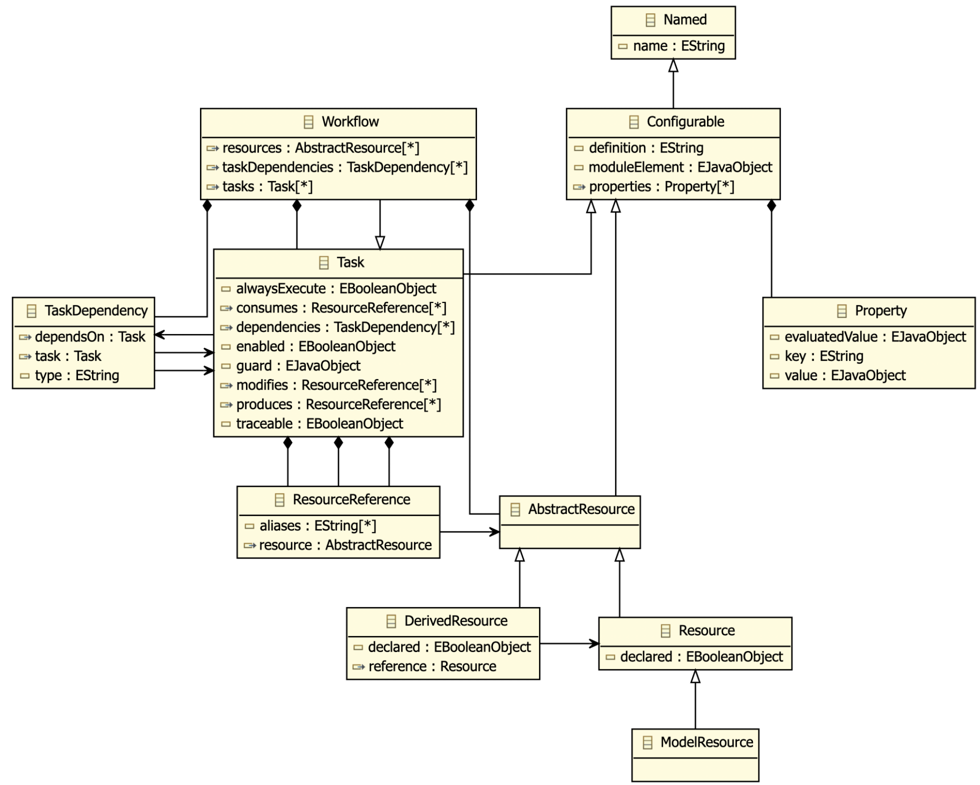


Figure 4.4: Workflow specification metamodel

by an executable block or expression that returns a value. This resolved value is captured in the `evaluatedValue` attribute once it is computed.

Workflow: A workflow element is the root of the model, and it represents an executable subprocess that is composed of tasks. A workflow may contain multiple resources, tasks, and task dependencies. Note that this class inherits from `Task`, which means that the workflow can be contained in another workflow.

AbstractResource: This abstract class is used to represent arbitrary types of resources, that is, a software artefact that can be consumed, modified, or produced by tasks. An abstract resource inherits from `Configurable`, therefore it can contain properties to configure itself and requires a `definition`.

Resource: This class is used to represent concrete resource instances that do not need to be loaded or disposed, for example, a file.

ModelResource: This class is used to represent a model resource, that is, an object that needs to be loaded and disposed.

DerivedResource: A derived resource is used to refer to an in-memory resource that can be shared across tasks but is not declared in the workflow. Such a derived resource can be a product of a task execution and can be reused by other tasks.

Task: This element represents an atomic executable work unit. This element extends from `Configurable` and as such it requires a name, a definition and

may contain configuration properties. A task may reference resources that it `consumes`, `modifies`, or `produces` as a result of its execution. Similarly, a task may specify any number of tasks as dependencies, that is, as pre-requirements for its execution. Its `alwaysExecute` attribute is used by the execution engine to skip the computations required to know if inputs and outputs have changed, which is used to determine if the task execution is required. When the `alwaysExecute` flag is enabled, the task is executed regardless of the state of its inputs or outputs. The `enabled` attribute is a boolean used to mark the task execution as enabled or disabled. Its `traceable` attribute is used to indicate whether the traces are to be recorded. This attribute is only relevant when the workflow execution has indicated that traces should be captured. The `guard` of the task holds an executable Java object that upon evaluation will determine is the task is allowed to continue with its execution.

ResourceReference: This element is used to map to a task a resource that can be consumed, modified, or produced by it. Within this reference it is possible to assign aliases to the resource that shall be valid only for the task that contains the resource reference.

TaskDependency: This element is used to capture dependencies between tasks. This element declares how a `task` depends on another through the `dependsOn` attribute.

4.3.4 Semantics

These are the execution phases of the language:

1) **Parsing:** This stage starts by parsing the workflow concrete syntax and resolving all elements in their corresponding abstract syntax. For example, the multi-task declaration from Listing 4.1 is resolved as a single instance of the class `TaskDeclaration` shown in Figure 4.3.
2) **Resolution:** In ModelFlow, the resolution phase is used to resolve dependencies and configure tasks and models. In practice this means that elements from the abstract syntax, which act as configuration placeholders, are resolved into runnable tasks and model instances. It is at this phase that the multi-task is resolved as two task instances, each with a different file source value.
   To achieve the task and model instantiation, their parameter values (which are statements or executable blocks) must be evaluated. Similarly, the for-loop collections of multi-task declarations must be evaluated to create the appropriate number of instances and configure each with their corresponding parameters.
3) **Pre and parameters:** In this stage the global parameters are evaluated from runtime information and then the pre blocks are executed.

4) **Dependency and execution graph construction:** At this stage the dependency graph with task and resource instances is resolved. Then the execution graph, which contains the order in which task shall be executed, is resolved from the dependency graph.
5) **Task iteration:** The engine iterates over all the tasks in the execution graph.
6) **Post:** At this stage, the block is executed and some of the internal caches are cleaned.

4.4 System design

In this section we describe the interpreter that can (a) conservatively execute such workflows based on changes made to relevant artefacts (e.g., models, model management programs, generated files), (b) provide unified model management traces and (c) load and dispose models as required. Sec. 4.4.1 describes how the interpreter knows when to execute a task in the workflow. Sec. 4.4.2 describes the process through which task and model definition types are retrieved and instantiated. Sec. 4.4.3 presents the process followed to support conservative task executions. Finally, Sec. 4.4.4 describes the process to capture model management traces and the structure of the model in which they are captured.

4.4.1 Knowing when to execute

ModelFlow uses a topological scheduler to dispatch and execute the tasks in the workflow. In Sec. 2.3.3 we discussed the advantages and disadvantages of such a scheduler based on Mokhov et al. [125]. Overall, this scheduler facilitates execution parallelisation but is unable to adapt the execution order based on dependencies discovered at runtime. In practice, ModelFlow uses declared model and task dependencies to determine the execution order, dynamic dependencies of tasks (such as programs or generated files) and models (e.g., reference model files) are resolved to determine if a task needs to be re-executed, not to alter the execution order.

Dependency graph

The first step in the build execution process involves deriving a dependency graph from the compiled workflow specification. In build tools, a dependency graph usually captures the order in which tasks are to be executed. In ModelFlow, the dependency graph contains not only tasks but also models, and is intended to capture dependencies between tasks and dependencies between models and tasks. Parameter inputs and outputs such as a task's source program or a task result are not captured in the graph. Cycles are allowed as

the graph serves to capture dependencies of the workflow, not to indicate an execution order to the scheduler.

The dependency graph is a *directed graph* composed of interconnected task (`TaskNode`) and model resource (`ModelResourceNode`) nodes. Multiple nodes for the same task or model are not allowed and models used by a task must be declared as one of: consumed, produced or modified. Similarly, tasks cannot `dependOn` themselves or on a model. The types of edges that are allowed in the graph include task interdependencies which indicate that one task should execute before another (`dependsOn`) and model-task interdependencies which indicate whether a task consumes (`in`), produces (`out`) or modifies (`inout`) a model. All these edges are created from explicit elements in the workflow declaration i.e. `dependsOn`, `in`, `out`, `inout`. However, in some cases, task definitions such as `epsilon:ecl`, produce in-memory models as a side-product of its execution e.g. a comparison model, that are accessible using a specific alias. These side-product models are not explicitly declared in the workflow program. In the cases where another task declaration requires such a model, the resolved dependency graph will capture this task-model relationship. For example, `TaskZ` in Listing 4.7 has both types of model-task dependencies, as `ModelA` and `TaskY.comparison` are both models consumed by `TaskZ` but one is declared in the workflow while the other is dynamically provided by the `epsilon:ecl` task.

```
1  model ModelA is epsilon:emf;
2  model ModelB is epsilon:emf;
3  model ModelC is epsilon:emf;
4
5  task TaskZ is epsilon:eml
6    in ModelA as A and TaskY.comparison as Comparison
7    inout ModelB as B
8    out ModelC as C;
9
10 task TaskY is epsilon:ecl
11   dependsOn TaskX;
12 task TaskX is epsilon:evl;
```

Listing 4.7: Sample workflow declaration

Visual representation. ModelFlow provides a visual representation for the dependency graph that is built using the Epsilon-Picto tool. This thesis uses this representation to facilitate the comprehension of the workflows to be used as examples or case studies. As such, this section is intended to describe its graphical elements and their meaning.

Consider the workflow declaration in Listing 4.7. Orange edges are used to

denote *modification* relationships between tasks and models. In it, `TaskZ` modifies `ModelB`. Yellow edges are used to denote explicit *production* relationships between tasks and models. For example, `TaskZ` explicitly produces `ModelC` because the task declaration indicates so. However, there are tasks that produce derived outputs because of their task definition type. For example, `TaskY` is an `epsilon:ecl` comparison task and the task definition (`epsilon:ecl`) provides a comparison model as an output of its execution. Purple arrows are used to denote this type of model resource production and are only shown when another task in the workflow consumes or modifies the resource. Green edges are used to denote *consumption* relationships between tasks and models. In the example, `TaskZ` consumes `ModelA` and the derived `comparison` model resource that is produced by `TaskY` for being a comparison task. Note that for all the above cases, model aliases are provided as edge labels. Additionally, all arrows described above are dotted to indicate that they relate a task and a model resource. In contrast, blue filled arrows are used to denote *requirement* relationships between tasks. A `dependsOn` construct in the workflow declaration is transformed into a *is required by* relationship in the view. For example, since `TaskY` depends on `TaskX` in the workflow declaration, `TaskX` is required by `TaskY` in the view. The resulting graph is a flowchart but not the actual execution plan.

Execution graph

The previous section presented the dependency graph which is used to capture how models and tasks depend on each other. However, to actually execute a workflow ModelFlow needs to translate this information into an executable graph that respects these dependencies. As such a wellformed dependency graph is an input of the algorithm that generates the execution graph. In contrast to the dependency graph, the execution graph is composed exclusively of task nodes (`TaskNode`) and uses a single type of edge that is later used by a scheduler to navigate the graph. Like the dependency graph, the execution graph is also directed but additionally it is *acyclic* to ensure that the execution terminates[2].

The algorithm that builds this graph, presented in algorithm 1. All declared tasks are captured in the execution graph, even if they are marked as disabled. Disabled tasks are skipped by the scheduler but are kept in the execution graph to respect task and model inter-dependencies. The first step in the algorithm to build the execution graph consists in creating all task nodes (lines 2-4). Task interdependencies are explicitly created by users to suggest an execution order and have a higher priority than task-model dependencies. As such. the

[2]The cycle detection mechanism in the ModelFlow prototype is provided by the JGraphT library.

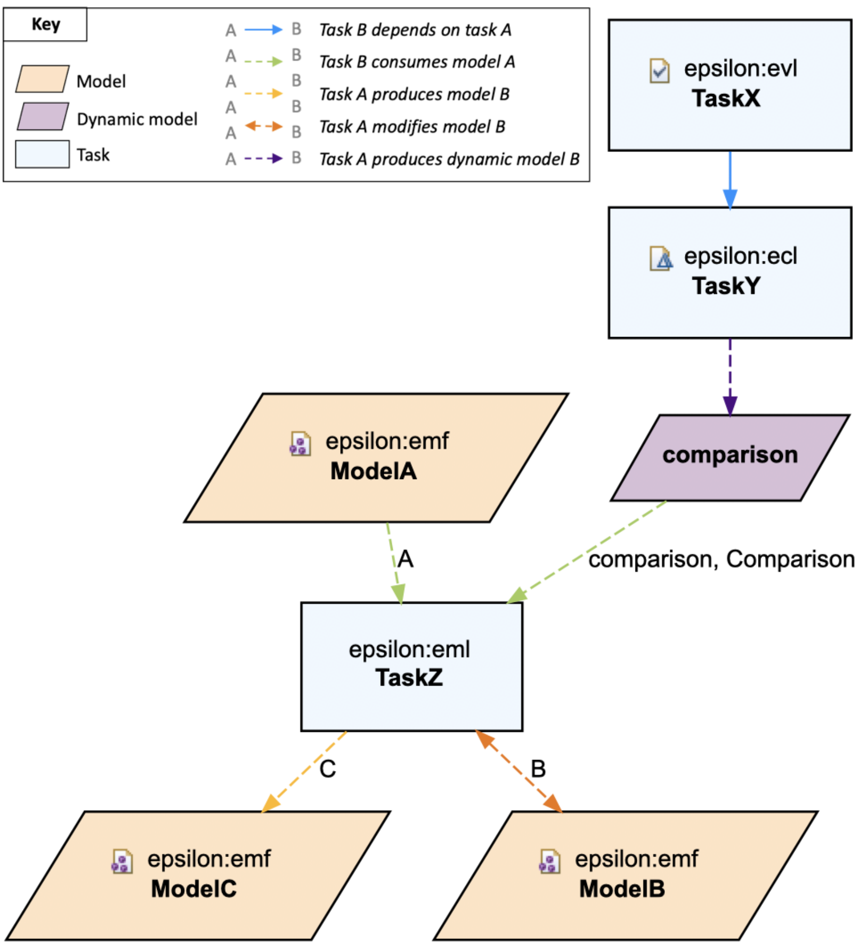


Figure 4.5: ModelFlow dependency graph of Listing 4.7.

next step in the algorithm is to insert the task interdependencies (`dependsOn` declarations) as edges (lines 5-15). The algorithm checks that for every edge insertion such an edge does not already exist and that no cycle is induced. If any edge induces a cycle, the algorithm will throw an exception and exit. Once the task interdependencies are captured, the algorithm goes on to capture the task-model interdependencies. Because there is only one type of edge in the execution graph, the algorithm analyses model-task edges in the dependency graph and for each model that connects two tasks by being used as input (`in` or `inout`) by one and as output (`out` or `inout`) by another then an edge is inserted (lines 16-28). As with the task interdependencies, edges are inserted only if such an edge does not already exist and if it will not induce a cycle, otherwise the execution is halted by throwing an exception. If all edges were successfully added, the graph may contain multiple paths that can be traversed to arrive from a source node to a target node. As such, the following step (lines 29-39) consists in removing all paths between a source and a target node except for the shortest path.

As an example, Figure 4.6 is the resulting execution graph built from the dependency graph in Figure 4.5. Once more, the blue arrows denote a *is required by* relationship.

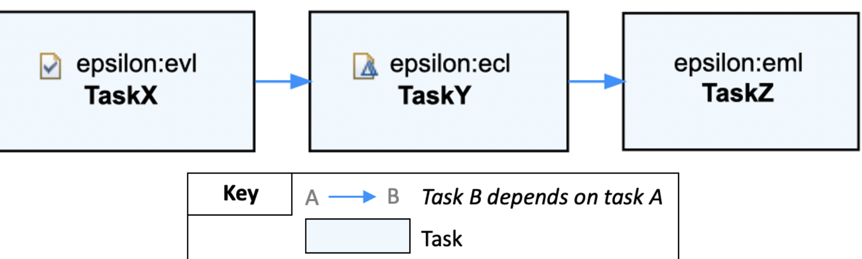


Figure 4.6: ModelFlow execution graph of Listing 4.7.

Schedulers

The next step in the build is to execute the plan. This is orchestrated by the *Executor* component by iterating over the graph in *topological* order. This ensures all required tasks are executed before the task at hand. In the present implementation of ModelFlow, tasks are executed sequentially. In future work we will add support for concurrent executions.

4.4.2 From declarations to runnable entities

After the dependency graph is built, task (*ITaskDeclaration*) and model (*IModelDeclaration*) declarations have been transformed into nodes in the graph. When the scheduler dispatches for execution a given task node (*ITaskNode*) a task instance (*ITaskInstance*) is created and configured with the parameter values from the declaration. Any model nodes (*IModelNode*) used by the tasks are also transformed into model instances (*IModelResourceInstance*). Additionally, a serialisable task element (*ITask*) is created with the resolved configuration of the task. In summary, *task nodes* are used to determine *when* to execute a given task declaration, *task* elements represent a *snapshot* of the resolved configuration for a given task node, and task instances represent the *executable unit* that performs actions prescribed by a task definition e.g., how to execute an EOL program. The Java classes involved in the process of transforming a *task* declaration into a runnable instance are illustrated in Figure 4.7.

Making definitions available

ModelFlow provides extension mechanisms that allow users to provide their own task and model definitions. The same mechanism is used by ModelFlow to provide readily available tasks and models to execute in ModelFlow workflows such as those of the Epsilon family. This mechanism relies on providing a

Algorithm 1: Execution plan construction algorithm.

```
   in Dependency graph (dg)
   Result: Execution graph (eg)
1  eg ⟵ new DirectedAcyclicGraph();
2  for t in dg.taskNodes do
3  |  eg.addNode(t);
4  end
5  for e in dg.edges do
6  |  if e is Task-to-Task dependency then
7  |  |  if e_{t1-t2} ∉ eg.edges & e_{t2-t1} ∉ eg.edges then
8  |  |  |  try:
9  |  |  |  |  eg.addEdge(e);
10 |  |  |  catch CycleInducedException:
11 |  |  |  |  haltExecution();
12 |  |  |  end
13 |  |  end
14 |  end
15 end
16 for t1 in dg.taskNodes do
17 |  for t2 in dg.taskNodes ≠ t1 do
18 |  |  if any (t2.inputs or t2.inouts) match any (t1.inouts or
   |  |   t1.outputs) then
19 |  |  |  if e_{t1-t2} ∉ eg.edges & e_{t2-t1} ∉ eg.edges then
20 |  |  |  |  try:
21 |  |  |  |  |  eg.addEdge(e_{t1-t2});
22 |  |  |  |  catch CycleInducedException:
23 |  |  |  |  |  haltExecution();
24 |  |  |  |  end
25 |  |  |  end
26 |  |  end
27 |  end
28 end
29 paths ← AllPaths(eg);
30 it ← TopologicalOrderIterator(eg);
31 while it.hasNext() do
32 |  node ← it.next();
33 |  for edge in node.outgoingEdges do
34 |  |  redundant ← paths.select(p ⟹ p.source ≡ node &
   |  |   p.target ≡ edge.target));
35 |  |  len ← min(redundant.length);
36 |  |  long ← redundant.filter(p ⟹ p.length ≠ len);
37 |  |  eg.removeAll(long.edges);
38 |  end
39 end
```

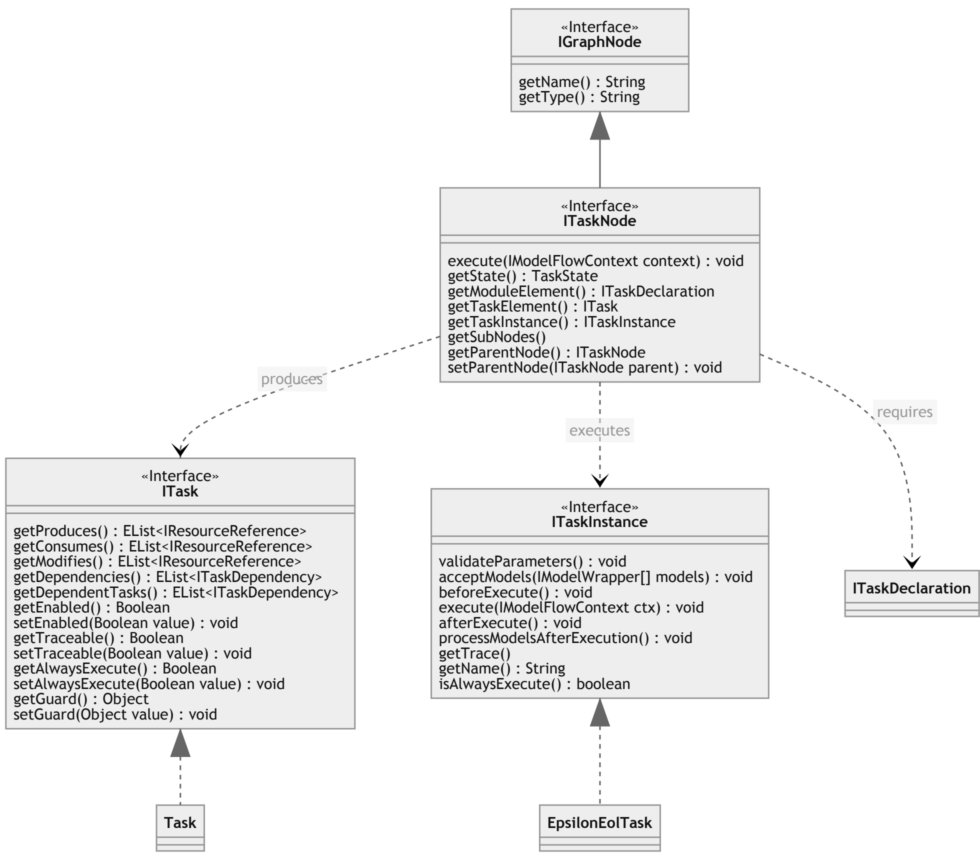


Figure 4.7: Class diagram of classes involved in the task instantiation process

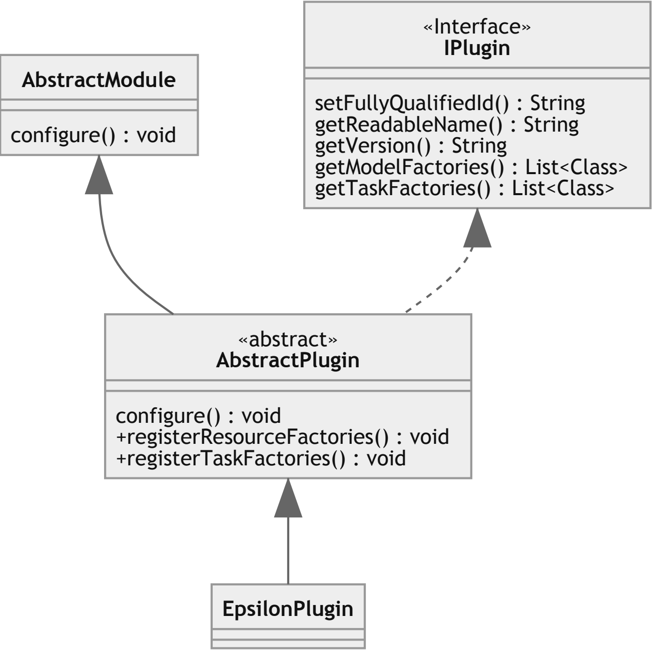


Figure 4.8: Class diagram of classes needed to contribute plugins

plugin object (instance of *IPlugin*) which references a group of model and task definitions. This plugin provides the definitions through the `getModel-Factories()` and `getTaskFactories()` methods which return a list of classes that extend the *IModelResourceInstance* or *ITaskInstance* interfaces, respectively. Additionally, plugins are made available by contributing to the extension point `org.epsilonlabs.modelflow.engine.pluginExtension`. For example, Figure 4.8) shows how the set of Epsilon models and tasks are contributed under the *EpsilonPlugin* class.

To match task and model declarations with their definitions, the definitions must be uniquely identifiable so that declarations can reference them by name. ModelFlow provides the annotation `@Definition` to provide an identifier for definitions for tasks (lines 1-5) and models (lines 6-10) as shown in Listing 4.8.

```
1   @Definition(name = "epsilon:eol")
2   public class EpsilonEolTask
3       implements ITaskInstance {
4     ...
5   }
6   @Definition(name = "epsilon:emf")
7   public class EpsilonEmfModel
8       implements IModelResourceInstance {
9     ...
10  }
```

Listing 4.8: Example of task and model definition classes.

Making task definitions runnable

When a task node is about to execute, the class that matches the definition type is instantiated. For example, based on Listing 4.8, a task declaration with the definition type `epsilon:eol` will produce a task instance of class *EpsilonEolTask*. After this, ModelFlow goes on to configure the task instance with information from the task declaration such as the source program (`src`). Task definitions must provide setter methods that indicate the name of the parameters to be accepted through the `@Param` annotation and the type of the parameter based on the method argument class. Listing 4.9 shows how the *EpsilonEolTask* definition class can receive a configuration parameter with the key `src` and `profile` expecting a file or a Boolean, respectively.

```
1  @Param(key="src")
2  public void setSrc(File src) {
3    this.src = src;
4  }
5  @Param(key="profile")
6  public void setProfile(Boolean profile) {
7    this.profile = profile;
8  }
```

Listing 4.9: Example of parameter configuration in task definition.

Task instances are runnable because they provide custom implementations of the *ITaskInstance* interface (see Figure 4.7) which is invoked when the task node is executing (see Sec. 4.4.3). The method `validateParameters()` is used to ensure that all the configuration parameters that were received are valid, potentially setting other internal values once all parameters have been assigned and resolved. The method `acceptModels()` which receives an array of model wrappers[3] (*IModelWrapper*) is used to determine if the models that are provided by ModelFlow are useful for the task (e.g., expected type or configuration) and to extract any information from them. Then, the method `beforeExecute()` is used to invoke any preparation activities that should be carried out before the execution. The `execute()` method is the one that defines the main logic of the execution after all checks have been successful. The `getTrace()` method is used to extract management traces from the execution and to translate them in the format that ModelFlow is expecting. This process is discussed in more detail in Sec. 4.4.4. Finally, the method `afterExecute()` is used for any clean-up activities required.

For multitasks like the one defined in Listing 4.1 a *TaskNode* and a corresponding *ITask* is created for each resolved task from the *forEach* collection.

[3]These elements have a reference to the model node in the dependency graph, its aliases and the resolved model instance (*IModelResourceInstance*).

Making model definitions runnable

As with tasks, model definitions can also use the `@Param` annotation to declare accepted configuration parameters. However, model definitions must implement the *IModelResourceInstance* interface which has a different set of methods that the definition must override. In particular, the `configure()` method is intended to perform further internal model setup after all parameters have been resolved. As in the `IModel` Epsilon interface, model definitions also need to indicate how to `load()`, `dispose()` and `save()` their contents, and to indicate if they are loaded (`isLoaded()`). Once configured and loaded, tasks can access the actual model instance by invoking the `get()` method. For example, Epsilon models must implement the *IModel* interface and Epsilon tasks consume elements of this type, as such, calling `get()` on a model instance representing an Epsilon model would return the required *IModel* instance.

The methods `asInput()`, `asOutput()`, `asInOut()`, `asTransient()` are used to adapt the model to the different tasks in the workflow that use them and are discussed in the next section. The methods `loadedHash()` and `unloadedHash()` are used to determine whether a model has changed and are described in Sec. 4.4.3.

Automated model management

ModelFlow minimises the number of model loading invocations by loading models when first used in the workflow and disposing them as soon as they are no longer needed. In particular, the decision to dispose a model is made when no task that remains to be executed needs the model.

While the model is being used by workflow tasks, these may use it in different ways. For example, one task could produce the model and a subsequent task could consume the model. Model definitions implement the `asInput()`, `asOutput()`, `anInOut()` and `asTransient()` methods to re-configure models as required by the next task to be executed taking into consideration its previous state. While switching from an output model to an input model may not require the model to be reloaded, switching from an input model to an output model may require a reload. As such, model definitions determine how to react to usage changes.

After each task execution, models used as outputs are saved but only disposed when no other task in the workflow will use them.

4.4.3 Conservative task executions

To determine whether a task needs re-executing ModelFlow examines its input and output parameters along with the models involved in the task. To do so, ModelFlow uses an execution trace model as store containing computed *stamps*

«Interface»
IModelResourceInstance<M>

getName() : String
setName(String name) : void
setAlias(String alias) : void
getAliases() : Collection<String>
configure() : void
asInput() : IModelResourceInstance<M>
asOutput() : IModelResourceInstance<M>
asInOut() : IModelResourceInstance<M>
asTransient() : IModelResourceInstance<M>
load() : void
isLoaded() : boolean
save() : void
dispose() : void
loadedHash() : Optional<Object>
unloadedHash(Object trace) : Optional<Object>
getKind() : ResourceKind
get() : M

Figure 4.9: Class diagram of the IModelResourceInstance interface

for parameters and models used by a task. Stamps were first introduced by Erdweg et al. [53] as values that could precisely indicate whether the file was up-to-date and that could be computed using a convenient strategy such as a timestamp, a hash, etc. We describe the execution trace model below.

Execution trace metamodel

To support conservative executions, the workflow execution engine requires a store in which the different inputs and outputs of a task are stored. The purpose of this store is the analysis of inputs and outputs used to compute whether the resources are up to date compared with previous executions. ModelFlow uses an execution trace model as this store. This execution trace conforms to the metamodel shown in Figure 4.10 and has been designed to trace the execution of models conforming to the metamodel from Figure 4.4. The type of data captured with this model includes end states along with property and resource stamps. Data that is not relevant for the identification of the up-to-date status of a task, is not stored in the trace. The execution trace is built from runtime information provided by the interpreter. Since the execution trace is captured in an EMF model, it is stored by default as a binary XMI file. The different metamodel elements are described below.

ExecutionTrace: This element is the root of the model and contains a list of `executions` (as `WorkflowExecution` elements) along with the `latest` known version of the model resources involved in the last workflow execution in the form of `ResourceSnapshot` elements.

Snapshot: This abstract element is used to capture a `stamp` or *hash* of an object along with the `timestamp` of when it was captured.

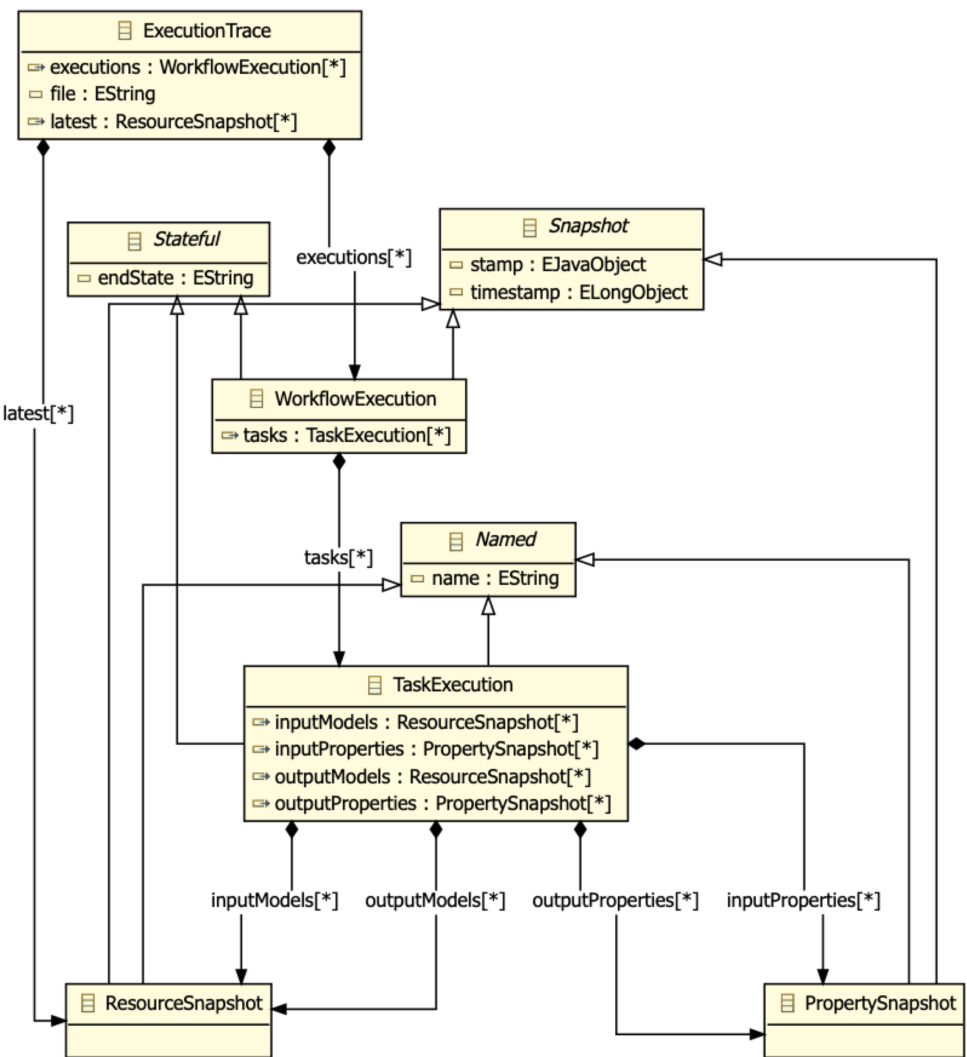


Figure 4.10: Execution trace metamodel

Stateful: This abstract class is used to capture the end state of a task or workflow execution.

Named: This abstract class is used to capture the name of task, model, and property references.

ResourceSnapshot: This element inherits from the class `Snapshot` the `stamp` and `timestamp` properties. Similarly, it inherits from class `Named` the `name` of the model resource it that the snapshot information belongs to.

PropetySnapshot: Like the ResourceSnapshot, this element inherits from `Snapshot` and `Named`. However, in this case the `name` of this element refers to the key of a task property that the snapshot information belongs to.

WorkflowExecution: A `WorkflowExecution` contains all `TaskExecution` instances. Additionally, it contains the stamp and timestamp of the original workflow that dictated the execution.

TaskExecution: A `TaskExecution` inherits from `Named` so that the name corresponds to the name of the task being captured. Additionally, it contains a list of input and output models as `ResourceSnapshots` and the snapshots of input and output properties as `PropertySnapshots`. Notice that for models that are declared as `inout`, that is, as to be modified by the task, the task execution captures a snapshot at the beginning of the execution that is stored within the `inputModels` collection, while it also captures another snapshot at the end of the execution that is recorded in the `outputModels` collection.

Detecting task changes

To determine if a task needs to re-execute, ModelFlow records a stamp of input and output parameters. In previous sections we have described how task definitions must use the `@Param` annotation to configure their properties. In this section we describe how some of these parameters can also be used as inputs and how non-configuration inputs and output parameters can be associated to the task. For example, Listing 4.10 shows two examples of input declarations for an *epsilon:egx* task definition. The input declaration consists in annotating getter methods with the `@Input` annotation and indicating a *key* to use as identification. In particular, the `getSrc()` method in line 2 is provided to use the source program as input while the `getImports()` method in line 6 is used to use additional imported files as inputs. Any Epsilon based program may import additional programs which may contain additional functionality or operations. By declaring these imported files as inputs of the task we ensure that a task execution is triggered if its source or any of its import dependencies change. Notice that while the `src` parameter is provided by the task declaration, the `imports` are resolved at runtime.

```
1  @Input(key="src")
```

```
2 public File getSrc() {
3   return src; // Provided by a task declaration
4 }
5 @Input(key="imports")
6 public List<File> getImports() { // Resolved at runtime
7   return getModule().getImports().stream().map(i->i.
       getFile()).collect(Collectors.toList());
8 }
```

Listing 4.10: Annotated input methods of an `epsilon:egx` task definition.

Annotated input methods provide objects that shall be *stamped* and stored in the worflow's execution trace model. Task declarations do not need to compute the stamps themselves; this is delegated to ModelFlow's `ParameterManager` class. This class looks for all `@Input` annotated methods of a task and computes a standard *stamp* based on the return value class, such as the *File* and *List*<*File*> returned by methods in Listing 4.10. By default, ModelFlow computes a message digest with the MD5 algorithm as stamps for objects that can be translated to a byte arrays, such as Files. To determine if a re-execution is required based on task inputs, ModelFlow compares the newly calculated stamps of all input keys with those available in the execution trace, if any.

In ModelFlow outputs are also processed before an execution to determine if they have been externally changed and therefore trigger a re-execution to discard the changes or skip it to protect them. For example, in case generated files were manually modified when they should not, ModelFlow could trigger a re-execution which restores the generated files to their original state. At the same time, manual modifications in generated files could have valuable information that is temporarily useful e.g., if changes are to be propagated to templates used as input or if evaluating the behaviour of tasks that depend on these changes. In these cases, we can prevent the producer task from executing to observe the response in other parts of the workflow.

Task outputs can be declared using the `@Output` annotation which also requires a *key* as identifier. ModelFlow offers two execution modes to deal with outputs: predetermined or interactive. In a predetermined execution ModelFlow will either discard any externally modified outputs by triggering a re-execution or skip the task execution altogether to prevent the changes from being overwritten. Alternatively, in an interactive execution, whenever ModelFlow detects an output to be externally modified, it will prompt the user which action to take. This is to prevent unintended modification of output resources (models, files, etc.).

In contrast to input stamp calculation, stamps cannot be recomputed from the outputs before the task is executed. As a work-around, ModelFlow stores in the trace sufficient information so that the stamp can be re-computed without

a re-execution. For example, recorded traces of generated files consist of a map with absolute file paths as keys and their calculated stamp as value. In contrast, recorded traces for input files only store the value of the computed stamp. This enables ModelFlow to re-compute the value and compare it with the value stored in the trace.

While ModelFlow handles the stamping process for basic types like Files and Strings, in some cases task definition may need to provide their own implementation. For example, the *epsilon:egx* task definition requires a dedicated stamper for generated files that takes into account protected regions. Protected regions in EGL are designated areas in generated files where manual modifications are allowed. To use this special stamper for its output files, the *epsilon:egx* task definition annotates its `generateOutputFiles()` method as an output with a dedicated *hasher* as shown in line 1 of Listing 4.11. This *hasher* implements the *IHasher* interface (Figure 4.11) for which two methods must be implemented: `fromExecutionTrace()` and `fromEvaluatedParameter()`. The first method is used to compute output stamps from information stored in the trace of a past execution while the latter is used to compute the output stamps once outputs have been produced after a task execution. The method in Listing 4.11 returns a dedicated object (`ProtectedFiles`) which can be serialised in the trace with sufficient information so that the stamps can be recomputed before execution in future invocations.

```
1  @Output(key="outputFiles", hasher=EglHasher.class)
2  public ProtectedFiles getOutputFiles() {
3    Collection<String> files;
4    if (outputFiles.isEmpty() && outputRoot.isPresent()
         && target.isPresent()) {
5      files = Arrays.asList(outputRoot.get() +File.
           separator+ target.get());
6    } else {
7      files = outputFiles.stream().map(OutputFile::
           getName).collect(Collectors.toList());
8    }
9    CompositePartitioner partitioner = getModule().
         getContext().getPartitioner();
10   return new ProtectedFiles(files, partitioner);
11 }
```

Listing 4.11: EGX input and output declaration in task definition.

If the execution of a task is to go ahead, the computed stamps for the inputs are stored in the execution trace before the execution by the `ParameterManager` while stamps for outputs are stored after the task execution.

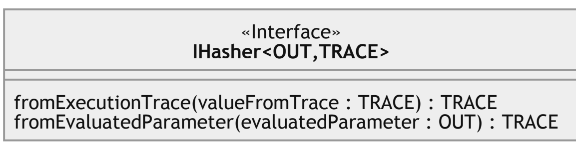


Figure 4.11: Class diagram of the IHasher interface

Detecting model changes

Just like inputs and outputs of the tasks, changes on models used by the task can also influence the decision to re-execute. In ModelFlow, model definitions must indicate how to compute their stamp because their contents may be split across multiple files or backed by a database. Moreover, with the intention of avoiding unnecessary loading, model definitions must also indicate how to compute this stamp when the model is loaded and when it is not. To compute these stamps, model definitions implement the methods `loadedHash()` and `unloadedHash()` from the `IModelResourceInstance` interface.

Take for example the stamp computation of *epsilon:emf* model definitions. Regular EMF models are backed by an XMI file but may reference others. If the model does not depend on these references to be processed, then the definition only needs to compute the stamp of the main model file. Otherwise, the definition needs to resolve all the dependent files and compute their stamps. In practice, these dependencies can only be resolved when the model is loaded, as such these are determined in the `loadedHash()` method before a task execution, if the model is an input, or after its execution, if the models is an output. In the cases where a model is both, the hash is computed before and after the task execution. The `unloadedHash()` is only used when the task is evaluating whether to re-execute and if the model has not been loaded by the execution of another task.

While this loaded/unloaded stamp mechanism is useful to prevent model loading, it requires two alternative computations (when loaded/when not) that must result in the same stamp if the model has not changed. This may pose a problem for model definitions that can only compute a stamp when loaded or when unloaded. Other strategies such as using a model indexing framework could also be used to avoid loading the models unless necessary. A model index such as Hawk [8] keeps a graph store of model elements of models in a repository and can be updated periodically or on demand. The index should point to the containing resource, in case these are multiple local or remote files. ModelFlow needs to know if a model has changed compared to the last time it was executed (using a given stamp or timestamp) however the current version of Hawk cannot provide this information. While adding support for this feature is an implementation matter that requires providing a dedicated index updater

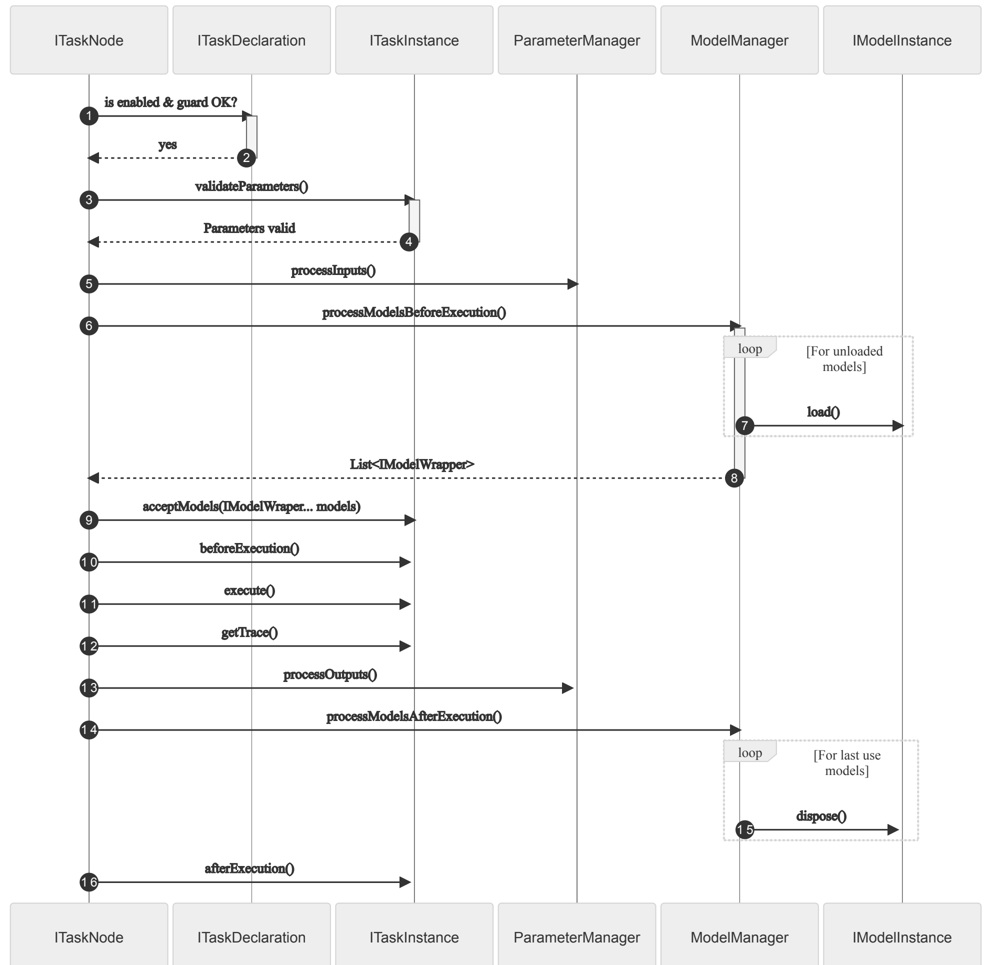


Figure 4.12: Sequence diagram of the process of a task's first time execution

and expanding the information stored in the graph nodes (e.g., version), this has been left as future work. Adding support for these features would be an important enhancement of model indexing frameworks.

Task execution process

Figure 4.12 shows a sequence diagram that provides an overview of execution process of a task node that has been instantiated and configured. There are five actors in this sequence diagram the task node, which is executing, the task declaration and task instance along with the task manager which locates and loads or disposes required model instances.

The successful execution of a task node starts by checking if the task declaration is enabled and if the guard provided evaluates to true ①. If this is successful then the node asks the task instance to validate its parameters ③. Then if this is successful, the parameter manager is asked to process inputs ⑤ while the model manager is asked to process models before the execution ⑥ which may include loading them if they are used for the first time ⑦. Then the

task instance method `acceptModels()` ⑨ is invoked using the list of model wrappers returned by the model manager ⑧. Any illegal model configuration for the given task may throw an exception, otherwise the task node goes on to invoke the `beforeExecute()` ⑩ followed by the `execute()` ⑪ methods of the task instance. Following the execution and if the declaration was configured to record model management traces, then the next method to be invoked from the task instance is `getTrace()` ⑫. After the traces have been processed, the parameter manager is asked to process any outputs ⑬ while the model manager is asked to process models after execution ⑭ which may involve disposing any model instance ⑮ that is last used by the current task node. Once the execution is finished and traces have been recorded the last method to be executed is `afterExecute()` ⑯ in which clean-up activities are performed.

4.4.4 Model management traces

ModelFlow provides facilities that enable the recovery and/or creation of traces from the execution of model management tasks. The traces are aggregated in a model that conforms to ModelFlow's model management trace metamodel, described later in the section. In subsequent workflow executions, ModelFlow maintains the trace model up to date by updating the traces for tasks that were executed. This model is captured as an EMF model that can be serialised. ModelFlow also provides a custom visualisation for recovered traces to facilitate the understanding and analysis of the activities in the workflow.

Model management tasks often produce traces as a side product of their execution. For these tasks, ModelFlow can collect their side-product traces and make them conformant to ModelFlow's metamodel. Since not all model management tasks produce traces (e.g., EOL tasks), ModelFlow also provides facilities that can be used by the tasks to create traces at runtime. We illustrate ModelFlow's trace aggregation process for tasks that produce traces as side product in the following example.

Consider a model management workflow that transforms a tree model into a graph model which is later used to generate a graph representation in Graphviz/DOT language. The first part of the workflow consists in the validation of the tree model to ensure that all tree elements have labels defined and that these are unique. The metamodel of the tree model is shown in Listing 4.12 and its EVL validation in Listing 4.13. The execution of the EVL produces a set of `ConstraintTraceItems` trace elements (see Figure 4.13) indicating the constraint executed, the model element instance it was evaluated for and whether it was successful.

```
1 class Tree {
2   id attr String label;
```

```
3   val Tree[*]#parent children;
4   ref Tree#children parent;
5 }
```

Listing 4.12: Tree metamodel

```
1 context Tree {
2   constraint HasLabel {
3     check : self.label.isDefined()
4     message : " Found tree with label undefined "
5   }
6   constraint HasUniqueLabel {
7     guard : self.satisfies("HasLabel")
8     check : Tree.all().label.select(l|l== self.label).
          size() == 1
9     message : self.label + " is not unique "
10  }
11 }
```

Listing 4.13: Tree validation in EVL

After the validation, the next step in the workflow consists in transforming the tree model into a graph model. The graph metamodel is presented in Listing 4.14 and the ETL transformation in Listing 4.15. The execution of the ETL produces a set of `Transformation` trace elements (see Figure 4.13) indicating the transformation rule executed, the source model element that triggered the transformation and the collection of model elements that were generated as a result.

```
1 class Graph {
2   val Node[*] nodes;
3 }
4 class Node {
5   id attr String name;
6   val Edge[*]#source outgoing;
7   ref Edge[*]#target incoming;
8 }
9 class Edge {
10  ref Node#outgoing source;
11  ref Node#incoming target;
12 }
```

Listing 4.14: Graph metamodel

```
1 rule Tree2Node
2     transform t : Tree!Tree
3     to n : Graph!Node {
4   n.name = t.label;
```

```
 5     if (t.parent.isDefined()) {
 6       var e : new Graph!Edge;
 7       e.source ::= t.parent;
 8       e.target = n;
 9     }
10   }
```

Listing 4.15: ETL transformation from Tree to Graph

The last step in the workflow consists in the translation of the generated graph model into a Graphviz/Dot (text-based) representation. This is done by running an EGL transformation that uses the template shown in Listing 4.16. This execution generates a `TraceLink` elements (see Figure 4.13) which link a model element and a particular property (`ModelLocation`) to a region in a file (`TextLocation`).

```
1  diagraph ConnectionsView {
2    node [color=lightblue2, style=filled];
3  [%for (node in Node.all){%]
4    [%=node.name%]
5  [%}%]
6  [%for (edge in Edge.all){%]
7    [%=edge.source.name%] -> [%=edge.target.name%]
8  [%}%]
9  }
```

Listing 4.16: EGL template that generates a Graphviz/Dot graph from a graph model

The structure of the traces produced by these tasks varies from task to task but they share some common elements. For example, the artefacts that are traced include model elements (e.g., `ConstraintTraceItem` and `Transformation`), model element properties (e.g., `ModelLocation`) and regions in files (e.g., `TextLocation`). Additionally, they may hold information on type of link they represent as indicated by the reference to the constraint or transformation rule that connects source and target artefacts. ModelFlow uses this commonality to propose the structure of its trace metamodel (described below) and requires the different tasks to translate their traces (e.g., `Transformation`, `CosntratintTraceItem`) to conform with this metamodel. In addition to the standardised format in which traces are captured, the ModelFlow model management trace metamodel requires model elements to be uniquely identifiable as it is this identifier that is kept in the trace rather than the model element object itself.

Workflow users can then use the model management trace model produced as a side product of a workflow execution to perform analysis. For example, users can identify if there are any unused elements in a given model across the different workflow tasks. In the previous workflow example, this could translate

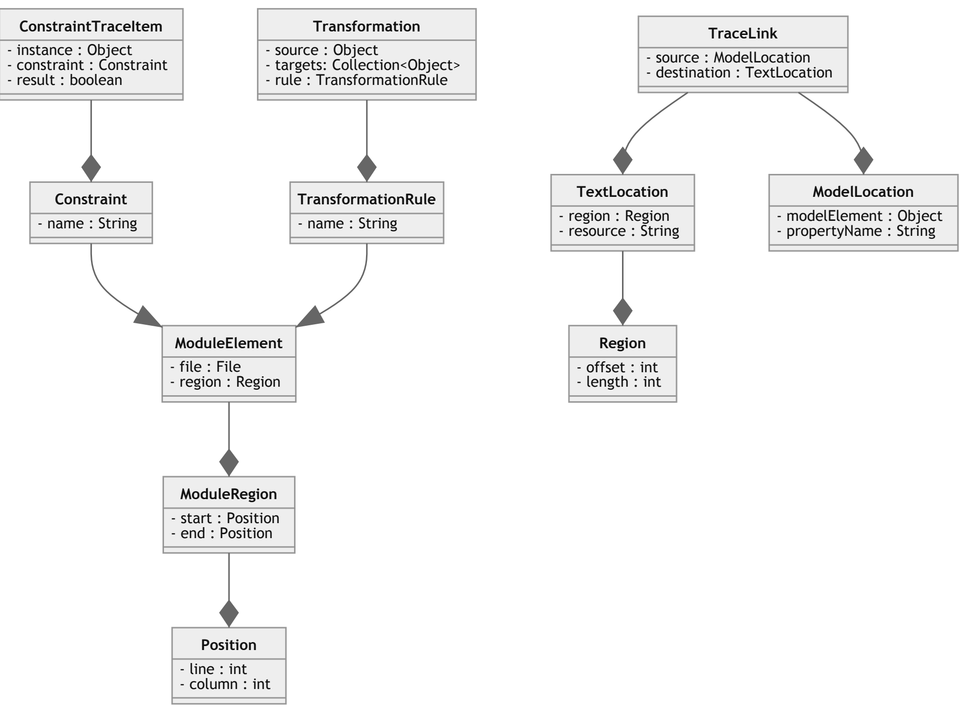


Figure 4.13: Class diagram of traces returned by EVL (List<ConstraintTraceItem>), ETL (List<Transformation>) and EGL (List<TraceLink>)

to identifying if all tree elements were translated into nodes in the graph model and if all of these nodes were in turn translated into a node declaration line in the output Graphviz description. Similarly, users can use the management trace to identify which model elements or properties template lines or generated output lines depend on. To illustrate this with the previous workflow example, a user could examine the model elements from the graph model related to a given template line and then navigate to identify which element they came from in the tree model. Similarly, the user could even check if the validation threw any warnings in model elements from the tree model that were indirectly used to produce a given line in the generated code. By analysing and navigating the model management trace model, users can determine model and program coverage across the workflow, debug programs in the context of the workflow, and assess the impact of model, program, and template changes in the rest of the workflow artefacts.

Model management trace metamodel

The result of ModelFlow's execution is an up-to-date version of the model management trace aggregating all traces that were collected from the execution of tasks in the workflow. The model management traces are captured in an EMF model conforming to the metamodel is presented in Figure 4.14 and

described below. Overall, this metamodel can capture traces that link arbitrary number of source and target elements which can represent model elements, model element properties, complete files or regions in files.

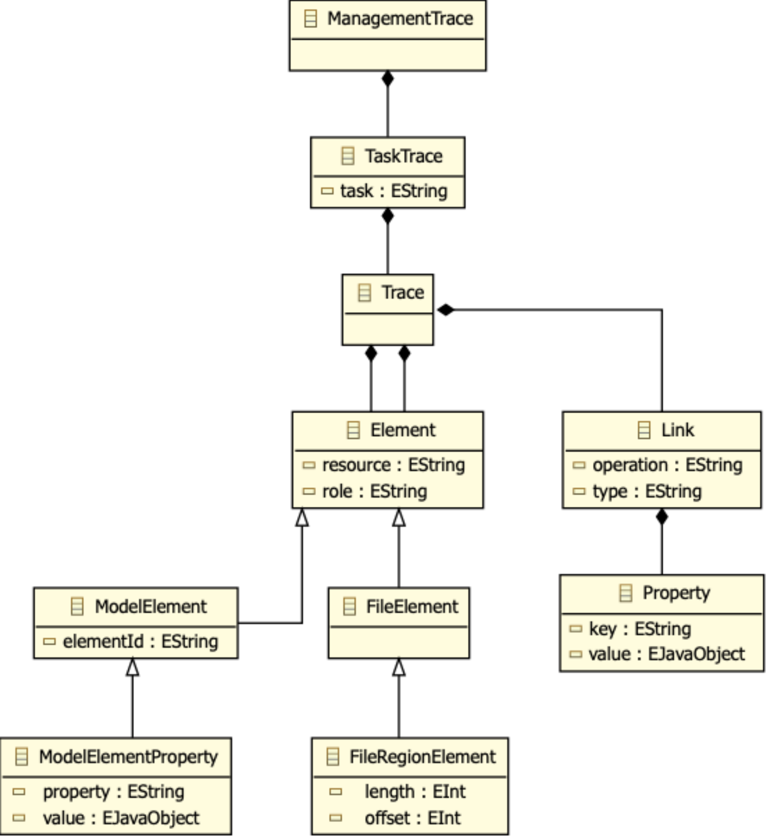


Figure 4.14: Model management trace metamodel

ManagementTrace: This is the root element which contains `TaskTrace` elements for each task that produces traces in the workflow.

TaskTrace: This entity is used to bind a collection of `Trace` elements to a task in the workflow identified through its name.

Trace: This entity represents a trace link that is composed of source and target elements connected through a link. The link may contain additional properties to characterise it. The source and target `Elements` of the trace link can have an arbitrary multiplicity and level of granularity.

Element: This is an abstract class that encompasses all possible types of source and target elements used in a `Trace`. An element must be associated to a `resource`, which is identified through its name. This resource may be contained in the workflow specification or produced during the execution of the workflow, e.g., a generated file. Additionally, an element may optionally specify its `role` in the trace.

ModelElement: This type of element represents a model element in a model resource. This entity requires an element identifier which is expected to

be unique within its containing model.

ModelElementProperty: This type inherits from `ModelElement` and represents a property of a model element. In addition to the inherited attributes, this element captures the name of the property.

FileElement: This type represents a file element. This is a commonly used type when dealing with model-to-text transformations. In this case, its `resource` attribute represents the name of the file. Optionally, a `region` in the file can be specified.

FileRegionElement: This element extends FileElement and represents a region in a file that is characterised by an `offset` and a `length`.

Link: This class is used to represent the link that connects source and target elements of a trace. The type of the link should specify the type of activity that produced it e.g., a model-to-model transformation. Additionally, trace links that are created as a result of a model management rule or operation within a task, e.g., a transformation rule in ETL, can specify which through the `operation` property. A link may contain additional properties to further characterise it.

Property: The metamodel also support attaching metadata to the traces through `Property` elements. Each property represents a key-value pair. For the trace to be serializable, the property value must be so.

Contributing traces

All task definitions in ModelFlow must implement the `ITaskInstance` interface which has a `getTraces()` method that returns a collection of `Trace` elements. To contribute a set of traces, tasks can override this method and translate their traces into the format expected by ModelFlow's Trace model.

For example, consider the definition of an ETL task definition, which overrides the `getTraces()` method as illustrated in Listing 4.17. Line 3 retrieves the collection of `Transformation` elements which represent the traces that resulted from the execution. Then, line 4 transforms each of these elements into ModelFlow `Trace` elements and then these are returned within an `Optional`.

```
1 @Override
2 public Optional<Collection<Trace>> getTrace() {
3   Collection<Transformation> etlTraces = module.
        getContext().getTransformationTrace().
        getTransformations();
4   Collection<Trace> mfTraces = etlTraces.stream().map(
        transf -> transform(transf)).collect(Collectors.
        toList());
5   return Optional.of(mfTraces);
6 }
```

```
7  private Trace transform(Transformation t) {
8    ManagementTraceBuilder builder = new
        ManagementTraceBuilder();
9    // link
10   String rule = t.getRule().getName();
11   builder.traceLink("Transformation",rule);
12   // source
13   String sId = getId(t.getSource());
14   String sModel = getContainer(t.getSource());
15   builder.addSourceModelElement(sId,sModel,null);
16   // targets
17   t.getTargets().forEach(target -> {
18     String tId = getId(target);
19     String tModel = getContainer(target);
20     builder.addTargetModelElement(tId,tModel,null);
21   });
22   return builder.build();
23 }
24 private String getId(Object element) {
25   try {
26     ModelRepository repo = module.getContext().
          getModelRepository();
27     IModel model = repo.getOwningModel(element);
28     return model.getElementId(element);
29   } catch (Exception e) {
30     return "unknown";
31   }
32 }
33 private static String getContainer(Object element) {
34   ModelRepository repo = module.getContext().
        getModelRepository();
35   IModel model = repo.getOwningModel(element);
36   for (IModelWrapper r : this.getResources()) {
37     if (model.equals(r.getModel())) {
38       return r.getResource().getName();
39     }
40   }
41   return "unknown";
42 }
```

Listing 4.17: Retrieving trace from `epsilon:etl` task definition

The utility method `transform(t:Transformation):Trace` (line 7) uses a builder utility that is provided by ModelFlow to assist in the creation of traces. The methods of this utility are shown in the class diagram of Figure 4.15.

There are two additional methods in Listing 4.17, `getId(Object):String`

ManagementTraceBuilder

+ManagementTraceBuilder()
+build() : Trace
+link(String type, String operation) : ManagementTraceBuilder
+link(String type) : ManagementTraceBuilder
+addSourceModelElement(String elementId, String container, String role) : ManagementTraceBuilder
+addTargetModelElement(String elementId, String container, String role) : ManagementTraceBuilder
+addSourceModelElementProperty(String elementId, String container, String name, String role) : ManagementTraceBuilder
+addTargetModelElementProperty(String elementId, String container, String name, String role) : ManagementTraceBuilder
+addSourceElement(String file, Integer offset, Integer length) : ManagementTraceBuilder
+addTargetElement(String file, Integer offset, Integer length) : ManagementTraceBuilder
+addProperty(String key, Object value) : ManagementTraceBuilder

Figure 4.15: Class diagram of the model management trace builder

(line 24) and `getContainer(Object):String` (line 33), which are used by multiple Epsilon task definitions in ModelFlow. In the case of the `Transformation` object that an ETL transformation trace returns, source and target elements represent model elements, not their identifiers. As such, the `getId` method is used to retrieve the id of the element within its containing model. Similarly, the `getContainer` method is used to identify the name of the model as used in the workflow specification.

But what about the task definitions that do not contribute traces? The model management task definitions can use ModelFlow's model management creation utilities to adapt their execution. In the case of EOL programs, their ModelFlow task definition has been adapted to accept a tracing utility accessed as a variable (`mfTrace`) in the programs at runtime (see Listing 4.18 —lines 4-5).

```
1 for (tree in T!Tree.all) {
2   var node = new G!Node;
3   node.name = tree.label;
4   if (mfTrace.isDefined()) {
5     mfTrace.trace(tree,node,"tree2node");
6   }
7 }
```

Listing 4.18: An EOL program creating traces at runtime

The `mfTrace` variable is created before the task's execution with a reference to the EOL task being executed. As such it contains an empty list of traces that is incremented every time the program calls the `trace` method. This method receives the source and target objects along with the link type. Then it resolves the model element ids and resources and translates them into a ModelFlow `Trace` instance. At the end of the execution, the collection of traces, if any, are aggregated in the management trace model.

```
1 public void trace(IModelElement source, IModelElement
     target, String link) {
2   ...
3   // once resolved the source and target ids and
```

```
      resources the trace is built with the information
4   Trace trace = new ManagementTraceBuilder().build();
5   traces.add(trace);
6 }
```

Listing 4.19: Trace utility to capture traces through EOL programs.

4.5 Implementation

ModelFlow is currently implemented as a series of Eclipse Java plugins that extend the language and processing facilities of the Epsilon project. As a model-based project, it uses several metamodels to capture the workflow specification, the execution trace, and the end-to-end traceability. Currently, workflow specifications can be prescribed using a Java API or a concrete Epsilon-based syntax. Sec 4.5.1 describes the rationale behind some of the implementation decisions while Sec. 4.5.2 describes the structure of the project in terms of the plugins implemented.

4.5.1 Decisions

We provide rationale for key implementation decisions such as the task and model definitions and the execution engine.

Execution engine. One of the first decisions we had to make was to decide a framework upon which to build and support the execution of ModelFlow. Because of its support for conservative executions, Gradle was our first option. However, at early development stages some of the Gradle internal features proved to be inaccessible or rigid which challenged our ability to experiment with task execution order, task generation and output management. To avoid being constrained we decided to implement our solution from first principle atop Epsilon.

The decision to build ModelFlow on Epsilon had multiple benefits. In particular, there were several languages built on top of EOL (e.g., ETL, EVL) that could be used as a guide for the addition of another language. In practice, we were more familiar and had hands-on experience in the usage and development of Epsilon which facilitated experimentation and modification of the code.

Task and model definitions. For similar reasons we decided to use Epsilon as the main task and model provider. In contrast to other model management frameworks or languages such as ATL and QVTo, Epsilon is a family of languages that support multiple model management activities. By adding support to the range of Epsilon languages ModelFlow would be immediately benefiting

from a range of operations that could be executed within the workflows. Additionally, the Epsilon architecture which decouples modeling languages from modelling technologies, provides similar benefits that make a range of modeling technologies such as spreadsheets, CSV, JDBC databases models available within ModelFlow workflows. Supporting multiple Epsilon languages and model drivers was key to demonstrate the extensibility of ModelFlow. However, case studies in Chapter 6 required the integration additional non-Epsilon tasks and models that also contributed to the demonstration its extensibility.

4.5.2 Plugins

We describe the Eclipse Java plugins that conform the implementation of ModelFlow. Overall, these can be classified as engine, contributors, setup, example and integration plugins.

Engine plugins. These plugins provide the basic functionality of ModelFlow along with the supporting user interface for Eclipse.

org.epsilonlabs.modelflow.engine: This plugin provides the core functionality of ModelFlow including the definition and parsing of workflow specifications, the resolution into a dependency graph, the task scheduling, the processing of task inputs and outputs and the maintenance of execution and management traces. Additionally, it provides extension points which are used by contributors to define additional model management tasks or model resources.

org.epsilonlabs.modelflow.engine.dt: This plugin contains the tools that are provided for users in the developer environment, i.e., Eclipse. In particular, this plugin contains the editor for the ModelFlow concrete syntax, the run configuration (Sec. 6.5.1) which allows users to run the workflows and Picto views that render a graphical display of the dependency graph of workflow specifications and of the resulting model management traces.

Contributor plugins. These plugins demonstrate the extensibility of the Modelflow framework. Each of these plugins is a model manager contributor (`mmc`) which contributes tasks and/or model definitions.

org.epsilonlabs.modelflow.mmc.core: Provides a set of basic tasks definitions including: reading file contents into memory, timed sleep, and console printing.

org.epsilonlabs.modelflow.mmc.epsilon: Provides definitions for Epsilon tasks including EOL, EVL, ETL, EPL, EML, Flock, ECL, EGL, EGX and EMG. Similarly, it provides definitions for Epsilon model resources such as EMF and Simulink. The Epsilon Simulink model implementation

is presented in Ch. 5 and its integration with ModelFlow was used to support the industrial case study (Sec. 6.3).

org.epsilonlabs.modelflow.mmc.emf: Provides the EMF task definitions for generating an Ecore from an Emfatic file and to generate code from an Ecore metamodel. These tasks were introduced to support the EuGENia case study (Sec. 6.2).

org.epsilonlabs.modelflow.mmc.gmf: Provides task definitions for GMF tasks which include the generation of GmfGen models from GmfTree, GmfGraph and GmfMap models and the generation of diagram code from GmfGen models. These tasks were introduced to support the EuGENia case study (Sec. 6.2).

Example plugins. These plugins provide functional examples of ModelFlow workflow specifications along with the model and task artefacts used in the workflow.

org.epsilonlabs.modelflow.examples.component: Provides a fully functional Workflow for the component case study (Sec. 6.1).

org.epsilonlabs.modelflow.examples.eugenia: Provides a Workflow for the EuGENia case study (Sec. 6.2) that is fully functional.

Setup plugins. These plugins are used to configure the development environment of ModelFlow and resolve required dependencies.

org.epsilonlabs.modelflow.target: This plugin specifies a collection of dependency plugins from Eclipse approved update sites.

org.epsilonlabs.modelflow.dependencies: This plugin is a utility that downloads and re-exports a series of dependency plugins from maven when these are not available as Eclipse update sites.

Integration plugins. These plugins provide integration with external build tools that allow the invocation of ModelFlow from them.

org.epsilonlabs.modelflow.maven: This plugin provides the ModelFlow integration with Maven in the form of an executable maven plugin. More details are provided in Sec. 6.5.2.

4.6 Summary

This chapter described the main features of ModelFlow: declarative specification, conservative executions, automated model management and aggregation of model management traces. Then, it introduced ModelFlow's architecture as language and interpreter that monitors task and model inputs and outputs while

producing management traces as output of the workflow execution. The chapter described its language, built atop Epsilon, in terms of its syntax and semantics. Then, the system design was described describing the process of contributing task and model definitions to ensuring they can configure themselves, declare inputs and outputs and can recover their traces, while also describing the execution process more in detail. Finally, the chapter described some of the key implementation decisions and plugins that were produced.

5 Supporting heterogeneous models: MATLAB/Simulink

In the previous chapter we introduced ModelFlow, a model management workflow tool that supports arbitrary types of models. In practice, most research on MDE and most of the open-source model management frameworks such as OCL and ATL tend to focus on manipulating models built atop the Eclipse Modelling Framework (EMF), a de facto standard for domain specific modelling. However, EMF is not the only type of modelling framework that there is. For example, MATLAB Simulink is a widely used proprietary modelling framework for dynamic systems that is built atop an entirely different technical stack to EMF. And yet, industrial software development processes rely on a variety of modelling tools such as Rhapsody, MagicDraw and PTC Integrity Modeller (PTC-IM) that are specialised in development tasks and have their own technical infrastructure.

One of the goals of this work is to support models beyond the EMF realm, targeting tools used in industry. This goal is set out to support organisations in their attempt to adopt open-source modelling and model management tools and use them along with their proprietary tools. One of such tool bridges was proposed between PTC-IM and the Epsilon model management framework in [198] in the context of the SECT-AIR project. Another one is the result of this thesis and bridges the Epsilon framework with MATLAB/Simulink models.

MATLAB/Simulink is a modelling framework for dynamic systems that is widely used across many industries including aerospace and automotive [11, 150, 151]. While this framework has its own set of model management capabilities to operate on its own models, such as code generation and validation, it does not offer facilities to export these models in XMI, the default exchange format for EMF models. As such, involving Simulink models in model management activities outside of MATLAB – particularly those involving other heterogeneous models – can be challenging.

The Massif [191] project offers facilities that make Simulink models available to model management frameworks with EMF support; this is achieved by transforming Simulink models into an EMF-compatible representation and vice-versa. With this approach the full Simulink model must be translated into EMF. This upfront transformation can be crippling for large models (as

demonstrated later in the chapter) and unnecessary when the model management programs do not work on the entire model. Additionally, Simulink models that continuously evolve may require the co-evolution of the EMF-counterpart which involves the re-execution of a non-incremental transformation which can be time consuming for large models. Furthermore, model management programs might be limited by the set of model element types supported by the Simulink-to-EMF transformation [123] which currently does not support Stateflow blocks.

Since MATLAB/Simulink is a tool that allows the creation of large and complex designs [119], we anticipated that the upfront transformation required with Massif would be expensive in time for these models. As such, we set out to implement an alternative approach that would shift the cost away from their EMF transformation and into the complexity of the manipulating program. Our approach consists in translating model management operations into MATLAB programs at runtime (on-the-fly). This ensures a constant synchronisation between the modelling tools and the MATLAB models. Since no upfront transformation is required, the round-trip engineering and co-evolution costs are eliminated. Sec. 5.1 introduces the modelling technologies used in our approach and Sec. 5.2 presents the architecture of our "live" approach to bridge MATLAB Simulink models with Epsilon.

Sec. 5.3.1 compares the performance of our approach against Massif's upfront model transformation by measuring the execution time of different stages of a representative model validation process. This process involves the execution of OCL-like invariants that validate structural properties on a sample of the largest available Simulink models on GitHub. Our evaluation indicates that our approach is more appropriate for continuously changing models as it can reduce the overall time of the validation process by up-to 80%. In contrast, the transformation-based approach (Massif) is better suited for signed-off models that need to be extensively queried as the cost of the transformation is a one-off and the validation two orders of magnitude faster.

Although the experimental results above show that it can reduce the overall execution time for a set of validation tasks on large models [162], the execution time was still high for certain classes of queries. Queries on collections of model elements were identified as an area for optimisation. To improve the performance of our solution we rewrite and delegate the execution of bulk queries to the MATLAB engine to take advantage of its inner indexes that are inaccessible by external clients. Sec 5.2 presents a query optimisation approach which works on collections of Simulink and Stateflow model elements. The experiment described in Sec. 5.3.2 with models that grow exponentially in number of elements demonstrates that off-loading to MATLAB these queries can improve their performance by up to 99%.

Our approach offers an additional option to manage Simulink models from model management frameworks that is convenient for large and/or continuously evolving Simulink models. Observations and lessons learned are discussed in Sec. 5.4. Our implementation atop Epsilon, which offers a set of model management languages, makes this approach accessible to a range of model management activities such as model validation, model-to-model and model-to-text transformations, model comparison, etc, that can involve multiple heterogeneous models (e.g., EMF, UML) in the same program. Sec. 5.5 discusses related work.

ModelFlow can manage heterogeneous models, including Simulink models which are managed through the proposed approach. The ModelFlow implementation enables tasks that use these models to load and dispose them when needed and to determine if they have changed from previous executions to support conservative executions. Through ModelFlow, it is also possible to generate unified traces from workflows that involve such models. Sec. 5.6 describes the integration of these Simulink models with ModelFlow and how are their model elements captured in the traces. Later, chapter 6 demonstrates the use of this integration in a case study.

5.1 Background

In this section we introduce the modelling technologies at the core of this work: MATLAB/Simulink, Epsilon, EMF and Massif.

MATLAB is a commercial tool developed by MathWorks that provides a variety of numerical computing environments. Under its Simulink [116] environment, it provides a graphical block-based modelling framework that supports the design, simulation, and analysis of dynamic systems as well as model management activities like code generation and continuous model verification for such systems.

Simulink models. These are file-based models that represent dynamic systems based on interconnected blocks. A sample Simulink model representing the behaviour of a car in motion after the accelerator pedal [117] is presented

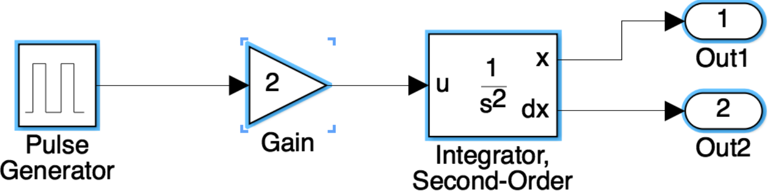


Figure 5.1: Example MATLAB/Simulink model.

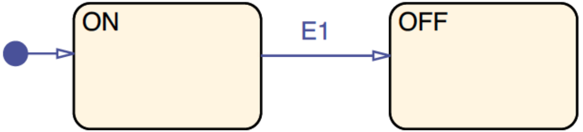


Figure 5.2: Example of MATLAB/Stateflow model elements

in Figure 5.1. The model contains five blocks from the Simulink library: a pulse generator, a gain, a second-order integrator and two outports. The pulse generator produces an input signal which simulates the accelerator pedal. The gain simulates the multiplied effect in the car acceleration. The second-order integrator enables the acquisition of the position and speed of the car from the acceleration through its outports. These blocks are interconnected by their ports through directed lines called signals.

Simulink model elements have both a type and a subtype. Example model element types include `Block`, `Line` and `Port`. Elements of *type* `Port` may have an `inport` or `outport` *subtype*. The list of subtypes is much longer for `Block` elements. All elements in Figure 5.1 are blocks and their subtypes, from left to right, are: `DiscretePulseGenerator`, `Gain`, `SecondOrderIntegrator` and `Outport`.

Stateflow. MATLAB offers an additional toolbox of decision logic, called Stateflow [118], used to describe how blocks react to events, input signals and time-based conditions. This toolbox is based on state machines and flow charts that can be attached to Simulink model elements. Figure 5.2 shows a sample Stateflow diagram containing two states named ON and OFF representing the operating modes of a system, and one transition[1], named E1, that connects one state to the other.

Stateflow model elements are persisted within a Simulink model. On a Simulink model there is a corresponding Stateflow machine which contains all Stateflow charts of the model. Each chart defines decision logic by combining logical elements such as states, boxes, functions, data, events, messages, transitions, junctions, and annotations. Only states, boxes and functions may contain any other logical elements in arbitrary levels of nesting. Stateflow charts may be included as blocks in the Simulink model.

All model elements in Stateflow are `Stateflow.Object` instances and their specific type names are always preceded by the `Stateflow` prefix and a period. For example, states are of type `Stateflow.State`.

Simulink functions. Simulink models can be manipulated manually using MATLAB's graphical interface or programmatically by invoking Simulink

[1] The arrow on the left is not a transition.

functions via MATLAB's command line interface. Listing 5.1 illustrates some of the main Simulink functions that enable model navigation and modification.

```
1 load_system m
2 find_system('m','Type','Block')
3 find_system('m','BlockType','Gain')
4 gain=add_block('simulink/Math Operations/Gain','m/Gain'
    )
5 get_param(gain,'BlockType')
6 set_param(gain,'Name','newName')
```

Listing 5.1: MATLAB Simulink functions

Line 1 shows how to load a model named `m` (same as its filename without extension) before we can interact with it. Line 2 shows how to retrieve all model elements of a given type, in this case, of elements of type `Block` from model `m`. For the model in Figure 5.1, this evaluation would return five blocks. By changing the value of the type parameter to `Line` or `Port` (instead of `Block`) the same evaluation would return the 4 signals or 8 ports from the figure, respectively. To find block model elements by their subtype it suffices to change the `type` keyword for `BlockType` in the `find_system` function. Line 3 illustrates query at subtype level which looks for block elements of subtype `Gain`. A similar approach applies for line and port elements which must replace the `BlockType` keyword for the corresponding `LineType` or `PortType`.

Line 4 illustrates the creation of a block of type `Gain`. The first function argument is the path of the library block to be used while the second argument represents the location in the destination model where the block will be created. This path starts with the name of the Simulink model, ends with the new element's intended name, and may contain in-between the name of intermediary nested blocks that will contain the new element. Regarding the management of model element properties, line 5 gives an example of how to retrieve the subtype property of a gain block while line 6 shows how to set the block's name.

MATLAB Java API. MATLAB provides several Application Programming Interfaces (APIs) that allow the invocation of MATLAB functions from languages like Python, C, C++, Fortran, and Java. In the case of its Java API, MATLAB provides the `MatlabEngine` class which can start or connect to a MATLAB engine and to evaluate MATLAB functions. The Java API also provides wrappers for specific MATLAB types such as structural arrays, cell arrays, etc.

Listing 5.2 illustrates a sample program that starts a MATLAB engine (line 1), evaluates MATLAB functions (lines 2-3) and then closes the connection with the engine (line 5). The evaluation of MATLAB functions through the engine is done using the `eval` method which receives the functions as strings.

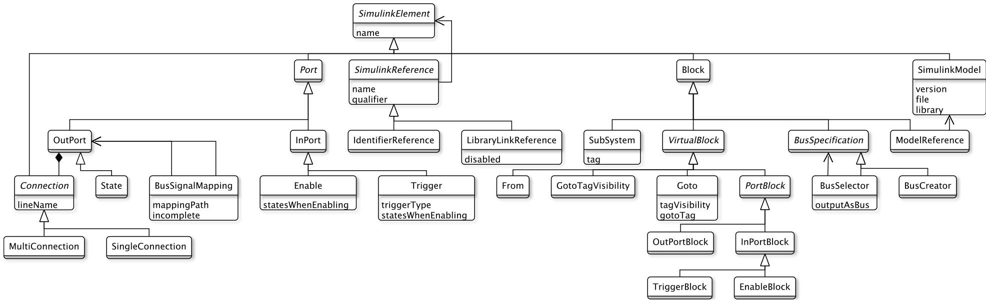


Figure 5.3: Simplified view of the Simulink element types provided by Massif's Simulink metamodel

Line 4 shows how the `getVariable` method can then be used on the engine to retrieve variables from MATLAB's workspace.

```
1  MatlabEngine e= MatlabEngine.startMatlab();
2  e.eval("load_system sl;");
3  e.eval("m=getSimulinkBlockHandle('sl')");
4  Object m = e.getVariable("m");
5  e.close();
```

Listing 5.2: MATLAB Java API

Massif. The Massif [191] project enables the transformation of MATLAB/Simulink models into an EMF-compatible representation and vice-versa. Massif connects to MATLAB's engine to parse and update Simulink models. The resulting EMF models conform to an Ecore Simulink meta-model defined by Massif which is limited to Simulink elements i.e., leaving out Stateflow elements.

Massif's Simulink Ecore meta-model. The Massif meta-model considers any Simulink model element that can be identified and named as a subtype of the `SimulinkElement` class. All subclasses of `SimulinkElement` are presented in Figure 5.3. Its direct descendants are `Connection`, `Port`, `Block` and `SimulinkModel`.

The `SimulinkModel` class is the root model element which keeps a reference to the file and version of the Simulink model. This class contains all the `Block` elements along with their `Port` and `Connection` elements.

In Massif, the ports (`Port`) of a block are either of type `InPort` or `OutPort` and they can be represented by a virtual block of class `PortBlock`. Similarly, the lines that connect the block ports are instances of the `Connection` class which can be either `SingleConnection` or `MultiConnection`. Any block whose MATLAB subtype cannot be found as a class in Massif is considered as a generic `Block`. Some blocks have predefined properties as attributes e.g., the

`tag` property in the `SubSystem` class, but most of their properties are dynamically added to their `parameters` attribute which contains array of `Property` elements, each with a name, value and type.

Some of the Massif meta-model constructs differ from the way MATLAB manages Simulink models. The most notable difference is that Simulink's block library offers 140 different `Block` subtypes (e.g., Gain, Sum, UnitDelay, etc.) while Massif only provides 11 concrete ones. The Simulink subtype of blocks that do not fall under the previous 11 subtypes can be retrieved from the block's `parameters` attribute, looking for the one with the `BlockType` identifier. Similarly, there are 5 `Port` subclasses in Massif's meta-model out of the 6 subtypes found in the Simulink library and, it is unclear how the `State` class in Massif maps to one or both of the `Reset` and `ifaction` port types in Simulink. A related inconsistency can occur when, after a transformation into EMF, the attributes of some block subclasses can have redundant or unpopulated values as they can also be found within the block's `parameters` attribute e.g., the `tag` attribute in the `SubSystem` class which can also be found in the `parameters`. Another difference is that the `Connection` class in Massif refers to Simulink model elements of type `Line` and subtype `signal` and that the `MultiConnection` and `SingleConnection` subclasses in the meta-model are used to refer to the `SegmentType` property of lines in MATLAB which can take the value of `trunk` or `branch`, correspondingly. In addition, subtype capitalization is important for Simulink functions e.g., `input` is used to refer to a port subtype as opposed to `Input` which identifies a block subtype. By contrast, in Massif the `InPort` and `InPortBlock` classes are used to refer to the port and block elements, respectively. Finally, MATLAB also handles special data types such as Cell Arrays[2] and Structure Arrays[3] which Massif stores as plain Strings.

From Simulink to EMF and vice-versa. Massif provides four different ways to transform Simulink models into an EMF-compatible representation. This process is known as the *import* process. The import modes can affect performance of the process as they differ in the way the MATLAB/Simulink `ModelReference` blocks[4] are resolved: The *shallow* mode does not process the referenced model; the *deep* mode creates new `SimulinkModel` elements for each `ModelReference` block; the *flattening* model processes these blocks as `SubSystem` blocks; and the *referencing* mode processes `ModelReference` blocks as new EMF resources (once) and references them in the model.

The Massif *export* process transforms the Simulink EMF-compatible representation into a Simulink file. This process can produce files with either `.slx` or

[2]Indexed data containers that can store any type of data.
[3]Groups of data in containers that store any type of data
[4]Blocks that represent a reference to another model

`.mdl` extension.

5.2 Integration with Epsilon

In this section we introduce the architecture and implementation of an approach that bridges models of the MATLAB Simulink environment with the Epsilon model management framework through on-the-fly translations of model management constructs into MATLAB functions. We chose the Epsilon [173] model management framework to implement and evaluate our approach based on the connectivity facilities that it offers and for the variety of model management languages in which the implementation becomes available. A similar approach can be implemented by other model management frameworks with similar connectivity facilities, such as ATL [87].

We present a concrete implementation (known as driver or EMC) that bridges with Epsilon a Simulink models and their Stateflow model elements. We have implemented other bridges for Simulink Dictionaries and Requirements whose architecture is described, and their use demonstrated in [162]. All drivers are publicly available as plugins of the Epsilon project [173].

Implementation. The Epsilon Model Connectivity (EMC) layer enables the uniform navigation and manipulation of models in any Epsilon model management language regardless of the model's underlying technology. Each driver implementation can access and interact with "live" Simulink models as they generate on-demand MATLAB commands that are executed on the Simulink model. To achieve this, these drivers connect to MATLAB's engine via the MATLAB Java API.

To illustrate the on-the-fly translation from EOL to MATLAB functions, consider the EOL program in Listing 5.3. At runtime, this program receives a model managed by the Simulink EMC driver, which can handle elements of type Block and knows how to manipulate their properties. The EOL `Block.all()` statement is used to retrieve all the Simulink Block model elements from the model. To collect these elements the Simulink driver replaces the `?` placeholder in the MATLAB function from line 1 in Listing 5.4 with the appropriate values, in this case the name of the model and the kind of element looked for i.e., `Block`. The resulting function (line 2) is then submitted for evaluation to the MATLAB engine through its Java API. The function returns a collection of block identifiers which is wrapped by the Simulink EMC into a lazy collection of `SimulinkBlock` instances to be used in subsequent processing. The `first()` statement from our EOL program in Listing 5.3 is then called on this lazy collection of `SimulinkBlock` elements. This statement is an Epsilon operation that works on collections of any type to return their first element. The following

statement `Name` is acting on the first `SimulinkBlock` returned. Since this model element belongs to the Simulink model handled by the Simulink EMC driver, it is the driver which handles the requested property access. To do so, the driver replaces the `?` placeholder in line 3 of Listing 5.4 and submits its populated version (line 4) to the MATLAB engine over the API. The `get_param` MATLAB function in this place is expecting as first argument the block's identifier (or handle) which is a number of type double. The last step consists in parsing the function result and assigning its value to the EOL variable `name`.

```
var name = Block.all().first().Name;
```

Listing 5.3: Collection of block names in EOL

```
1 find_system('?','type','?')
2 find_system('modelName','type','Block')
3 get_param(?,'Name')
4 get_param(34.394856839,'Name')
```

Listing 5.4: MATLAB functions to collect Simulink blocks and their names.

Architecture. Figure 5.4 shows the architecture of Simulink-based drivers and how they relate to the core facilities of the Epsilon Model Connectivity layer (Group 1). The concrete drivers such as the Simulink EMC (Group 3) uses common classes and helpers that are provided by the abstract Common Simulink EMC (Group 2) which extends the core EMC. The common facilities include the configuration of the Simulink project and the establishment of a connection with the MATLAB engine. In addition, a set of abstract classes to handle lazy collections of Simulink-based model elements are also provided. Each concrete driver extends the `AbstractSimulinkModel` class and implements its own approach to create, delete, and collect elements of specific types. This is done by overriding the respective methods from the `Model` superclass.

5.2.1 Simulink models

Model. The Simulink EMC driver considers a Simulink file (*.slx or *.mdl) as a model. This model is managed as an instance of the `SimulinkModel` class (see Figure 5.4). A model defines the behaviour of inherited methods from the class `AbstractSimulinkModel` in the Common Simulink EMC layer which in turn extends functionality from the `CachedModel` class defined in the EMC layer. Together, these classes describe how a model will perform CRUD operations on its *owned* model elements and the model itself, while they also determine how to *load* and *dispose* the model instance before and after the execution of a model management program e.g., validation, transformation.

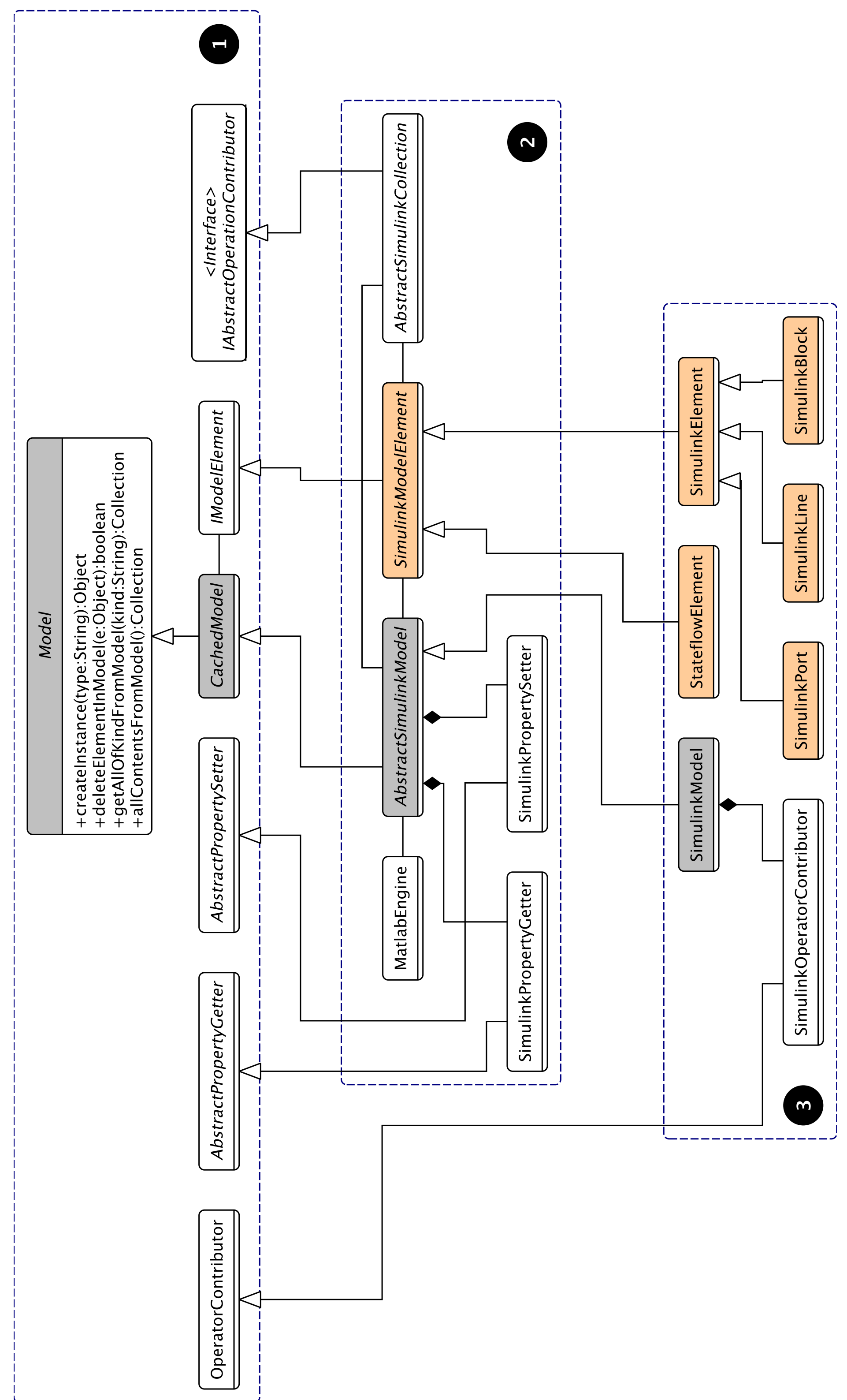


Figure 5.4: Class diagram with architecture of the Simulink driver. Group 1 represents the Epsilon Model Connectivity (EMC) Layer. Group 2 contains the Common Simulink EMC facilities. Groups 3 show the main contents of the Simulink Model EMC.

Model elements. The `SimulinkModel` manages elements that inherit from the `SimulinkModelElement` class. As such, elements can be of type `SimulinkElement` or `StateflowBlock`. For each MATLAB Simulink type e.g., `Block`, `Port` and `Line`, there is a corresponding class e.g., `SimulinkBlock`, `SimulinkPort` and `SimulinkLine` that extends `SimulinkElement`. These classes provide additional methods e.g., to link blocks or to change their parents; and may override the behaviour of CRUD operations for the type of element they work on.

As discussed in Sec. 5.1, Simulink elements in MATLAB have subtypes e.g., an element of type `Block` may be of subtype `Gain` or `SubSystem`. In Epsilon, the union of an element's super types and of its concrete type is referred to as the element's *kinds*. The Simulink EMC driver considers the Simulink subtype (e.g., `Gain`) as the model element concrete type, while still considering their Simulink type (e.g., `Block`) as one of their kinds. Stateflow element types (e.g., `Stateflow.State`) are used as their concrete type in Epsilon. At the same time, all Stateflow elements belong to the `Stateflow` kind in Epsilon.

MATLAB Simulink model elements provide different ways to be identified (e.g., *path*, *id*, *handle*). However, MATLAB Simulink functions return either *handles* or *paths*. As such, for Simulink elements, the driver uses as identifier their *handle* property which is a non-persistent session-based immutable identifier of type Double. In contrast, the driver uses the *id* property (Integer) to manipulate Stateflow elements which is easy to retrieve from the Stateflow objects returned by most Stateflow functions and queries. In the rest of the chapter, we use interchangeably the words *identifier* and *handle* of a model element to refer to the mechanism by which specific element instances are retrieved across MATLAB toolboxes.

Create element. The `SimulinkModel` instance manages the creation of Simulink and Stateflow model elements. When the reserved word `new` precedes a type name in an Epsilon program, the interpreter invokes the method `createInstance (type:String)` of the EMC model.

To instantiate Simulink blocks, MATLAB requires the path of the block in the Simulink library. The user is responsible for providing this path in order to instantiate a block in Epsilon. Once provided, the model populates the MATLAB function `add_block` with the path of the library block then asks the MATLAB engine to evaluate it. Listing 5.5 shows the creation of `Sum` and `SubSystem` blocks in EOL using their library block path[5]. The Simulink driver creates these blocks at the top level of the Simulink Model but they can later be placed elsewhere by changing their parent.

```
1 var sum = new `simulink/Math Operations/Sum`;
```

[5]The use of the back-tick (`) is required when a type identifier contains spaces.

```
2 var subsystem = new `simulink/Ports & Subsystems/
      Subsystem`;
```

Listing 5.5: Model element creation

There is no equivalent `add_port` function in MATLAB to create port model elements. In contrast, the `add_line` MATLAB function which creates lines, requires the source and target ports to be connected. The Simulink EMC driver does not allow the direct creation of lines through EOL statement such as `new Line` or `new signal`. Instead, lines are created using *linkage* methods on block elements which may specify the source and/or target ports to be connected. For example, provided the model from Figure 5.1 with no lines, these can be created with the following EOL program:

```
1 pulse.link(gain);
2 gain.linkTo(integrator, 1);
3 integrator.linkFrom(outport1, 1);
4 integrator.linkFrom(outport2, 2);
```

Listing 5.6: Linking methods for block elements in EOL

In MATLAB, Stateflow elements use a different syntax for instantiation which consists of their type followed by a container. For example, a Stateflow state can be created by invoking the function in Listing 5.7 where `chart` is the container Stateflow element. This same statement can be used in EOL to instantiate this state by preceding it with the `new` reserved word (line 1). Additionally, the Simulink EMC can delay the instantiation of Stateflow elements until the parent is assigned. In other words, a placeholder is created when using a statement with no parent (line 2) which is only submitted to the MATLAB engine for instantiation when its `parent` property is assigned (line 3). Before then, other properties of the Stateflow element can be assigned in memory to its placeholder. These properties are submitted to MATLAB just after the element is instantiated.

```
1 Stateflow.State(chart)
```

Listing 5.7: Stateflow element creation in MATLAB

```
1 var off = new `Stateflow.State`(chart);
2 var on = new `Stateflow.State`;
3 on.parent = chart;
```

Listing 5.8: Stateflow element creation in EOL

Delete element. In Epsilon programs, deleting a model element involves the use of the `delete` reserved word before the element to delete as shown in Listing 5.9.

```
1 delete sum;
2 delete subsystem;
```

Listing 5.9: Model element deletion in EOL

Deleting a Simulink block or a line in MATLAB is performed with the functions from lines 1 and 2 in Listing 5.10, respectively. There is no equivalent `delete_port` MATLAB function.

```
1 delete_block(blockElement);
2 delete_line(lineElement);
```

Listing 5.10: Simulink element deletion in MATLAB

The `SimulinkModel` is responsible for the deletion of model elements and does this through its `deleteElementInModel(e:Object)` EMC method . For Simulink elements, the Simulink EMC chooses the appropriate MATLAB function for the element being deleted and provides its appropriate identifier. MATLAB has a different syntax to delete Stateflow elements which is the dot notation e.g., `elementId.delete`.

Read and update element properties. The `SimulinkModel` delegates to instances of the `SimulinkPropertyGetter` and `SimulinkPropertySetter` classes the responsibility of reading and updating properties of model elements. The former receives a model element and the property that is to be retrieved from it while the latter additionally requires the value to be assigned to the element's property.

Depending on the kind of model element that these act upon, they populate and evaluate different MATLAB functions. For example, when dealing with Simulink blocks, these property *managers* evaluate the MATLAB functions from Listing 5.11.

```
1 get_param(element,'PropertyName')
2 set_param(element,'PropertyName',value)
```

Listing 5.11: MATLAB Simulink element getter and setters

An example of an EOL program retrieving and populating Simulink element properties is shown in Listing 5.12.

```
1 subsystem.name = "Controller";
2 var subsystemName = subsystem.name;
3 sum.description = "Sum block";
4 var sumDescription = sum.description;
5 var inportHandles = subsystem.LineHandles.Inport;
```

Listing 5.12: Get and set Simulink element properties in EOL

Lines 1 and 3 set element properties while lines 2, 4 and 5 get property values

from them. In the case of line 5, the property `LineHandles` returns a Structured Array, which is a MATLAB-specific type that represents an array of key-value pairs. In MATLAB, their values are retrieved using the `getfield(element, property)` function. The Simulink EMC driver can identify these types and navigate them as any other property. In the example, the value of its `Inport` key is retrieved.

In MATLAB the dot notation is used once more to get and set properties from Stateflow elements. This is illustrated in Listing 5.13 where the name of a Stateflow State `element` is retrieved (line 1) and set (line 2). The syntax to do the same in an EOL program would be identical.

```
1  element.Name;
2  element.Name='NewName';
```

Listing 5.13: Get and set Stateflow element properties in MATLAB and EOL

Retrieve elements. To collect all instances of a given type, Epsilon programs use the `all()` operation on types. Alternatively, to collect all available elements on the model, Epsilon provides the `allContents()` operation at the EMC model level. Given a model `M`, Listing 5.14 illustrates different ways to retrieve Simulink model elements in EOL.

```
1  var blocks = M!Block.all();
2  var lines = M!Line.all();
3  var ports = M!Port.all();
4  var sums = M!Sum.all();
5  var subsystems = M!SubSystem.all();
6  M.allContents();
```

Listing 5.14: Retrieval of model elements in EOL

The `getAllOfKindFromModel(kind:String)` method from the `SimulinkModel` is triggered by the `all()` operation (lines 1-3). At first this method attempts to find elements of either `Block`, `Line`, `Port` or `Stateflow` kind. If the `kind` argument does not match any of those, as in lines 4-5, then the `SimulinkModel` will attempt to find the MATLAB subtype e.g., `SubSystem` blocks or `Stateflow.State` elements. In contrast, the use of the `allContents()` in line 6 triggers the result aggregation of collections by kind i.e., `Block`, `Port`, `Line`, and `Stateflow` elements.

Line 1 in Listing 5.15 reminds the reader how elements of type Port, Block or Line can be collected in MATLAB, while line 2 shows how this function is adapted to collect elements by their subtype. The `SimulinkModel` class populates and submits the appropriate MATLAB functions for the element kinds (e.g., Block) or types (e.g., Sum) to be collected and then stores the results in lazy collection objects which extend the `AbstractSimulinkCollection`

class.

```
1 find_system(model,'type','Port')
2 find_system(model,'blockType','Sum')
```

Listing 5.15: Retrieval of Simulink elements in MATLAB

Stateflow elements are collected using the MATLAB functions in Listing 5.16 which act on the model handle. All Stateflow objects can be retrieved by passing the `Stateflow.Object` as `isa` parameter but subtypes (e.g., `Stateflow.State`) can also be passed instead. The approach to collect these from Epsilon is shown in Listing 5.17.

```
1 model.find('-isa','Stateflow.Object');
2 model.find('-isa','Stateflow.State');
```

Listing 5.16: Retrieval of Stateflow elements in MATLAB

```
1 M!Stateflow.all();
2 M!`Stateflow.State`.all();
```

Listing 5.17: Retrieval of Stateflow elements in EOL

Element methods. The Simulink EMC adds convenience methods to its model and model elements, such as the one for linking blocks in Listing 5.6. Other methods are also available, such as `getType`, `getParent` and `getChildren`. Nevertheless, MATLAB provides many more functions for its Simulink and Stateflow model elements that would be challenging to individually replicate in the EMC driver. To deal with this, when an unknown method in EOL is called on the model or its elements the following strategy is applied.

Many MATLAB functions for Simulink model and model elements have a common structure (Listing 5.18) which takes the model element as first argument. At the same time, model element operations in EOL are executed as instance methods and have the form shown in Listing 5.19.

```
method_name(element,arg0,...,argN)
```

Listing 5.18: MATLAB function structure

```
element.methodName(arg0,...,argN);
```

Listing 5.19: EOL method structure

To execute non-hard-coded MATLAB functions, the Simulink driver dynamically translates the method as a MATLAB command and submits it to the MATLAB engine for evaluation. The `SimulinkOperatorContributor` class specifies this behaviour. As an example, consider the EOL statements in Listing 5.20 which would be translated to the corresponding MATLAB functions

in Listing 5.21.

```
1  subsys.find_mdlrefs();
2  subsys.find_mdlrefs('AllLevels',true);
```

Listing 5.20: Invocation of MATLAB functions as EOL methods

```
1  find_mdlrefs(subsys)
2  find_mdlrefs(subsys,'AllLevels',true)
```

Listing 5.21: Sample MATLAB functions that act on Simulink elements

MATLAB operations acting on Stateflow elements commonly[6] share the same syntax as EOL, except for operations with no arguments which do not require brackets in MATLAB. Through the `SimulinkOperatorContributor` class the Simulink EMC driver can change the translation of these functions depending on the model element kind they act upon.

5.2.2 Collection query optimisation

The Simulink driver returns lazy collections of model elements when retrieving elements by type or kind. This capability was already presented in [162]. However, performing collection and selection operations on these collections can become computationally expensive as these collections grow in size because they are performed sequentially by default. Taking advantage of some of the MATLAB functions which can perform bulk operations much more efficiently, in this work we use them on collections of Simulink or Stateflow model elements when *select* or *collect* operations involve property checks on their members.

A *collect* operation works on a collection and consists in evaluating an expression on each member of the collection to return a new collection with the evaluation results. For example, the EOL statement from Listing 5.22 starts on a collection of all Block elements in the model and returns a new collection with all the names of these elements.

```
Block.all().collect(b|b.Name);
```

Listing 5.22: EOL collection of Simulink block names

A *select* operation also works on collections and filters the collection leaving only the elements that satisfy a given condition. For example, the EOL statement from Listing 5.23 starts from a collection of elements of Inport type and returns a copy of the collection with only the elements named `Temperature`.

```
Inport.all().select(i|i.Name=="Temperature");
```

Listing 5.23: EOL selection of Simulink inport blocks

[6]The only method that does not follow this structure is provided by the driver.

Lazy collections of Simulink or Stateflow model elements work by storing the array of model element identifiers (handles) and only constructing the appropriate wrapper (e.g., `SimulinkBlock`, `StateflowBlock`) when acting on the elements of the collection. For example, when `Block.all()` is invoked in Epsilon, the collection of blocks returned by the appropriate MATLAB function is an array of Simulink handles (doubles). There are operations we can compute on this array without having to resolve them into their corresponding `SimulinkBlock` wrapper instance. For example, we can get the number of blocks on the collection by getting the size from the array of Simulink handles. However, when select or collect operations are invoked on a lazy collection, their argument expressions are likely to involve interactions with properties from elements in the collection. As such the lazy collections must iterate over their elements, instantiate them in their appropriate wrapper class and evaluate their expressions.

We have extended the implementation of the lazy collections to support the invocation of select and collect operations without having to instantiate wrapper classes for all its elements. To achieve this, the lazy collection first checks whether the operator's expression can be optimised (i.e., has a specific form) and if so then the collection builds a MATLAB function that can use the array of Simulink handles. Currently, we optimise only those operations whose expressions can be translated into a valid bulk MATLAB statement.

For *collect* operators we currently support simple property navigation expressions such as Listing 5.22. The MATLAB functions in Listing 5.24 are used to collect properties from collections of Simulink (line 1) or Stateflow (line 2) model elements. These functions take as first argument the array of element handles and replace the `'?'` placeholder with the property name to be retrieved.

```
simulink=get_param(handles, '?')
stateflow=get(handles,'?')
```

Listing 5.24: MATLAB Simulink and Stateflow collection operations

The Epsilon operator *sortBy* reuses this implementation to sort the elements on the collection after they have been collected in bulk.

The *select* operator optimisation for collections of Simulink model elements uses the MATLAB function in Listing 5.25 to perform the bulk queries. This function replaces the question mark placeholders with property-value pairs that all elements in the collection must match. When more than one property-value pair is used the function performs the logical AND operation. As such, optimised select operations in Epsilon only support expressions which involve logical AND expressions that, as in the collect case, involve simple property checks. Select operations that do not match this criterion fall back to the default non-optimised evaluation.

```
simulink=find_system(handles,'?','?')
```

Listing 5.25: Simulink element selection MATLAB function

An example of a supported EOL query on a collection of Simulink model elements is shown in Listing 5.26. The corresponding MATLAB function submitted to the engine is shown in Listing 5.27, where `all` refers to a collection of Simulink element handles.

```
1  Gain.all().select(g|(g.Gain==2) and (g.Name=='Gain'))
```

Listing 5.26: EOL selection of Simulink gain blocks

```
1  find_system(all,'Gain',2,'Name','Gain')
```

Listing 5.27: MATLAB selection of Simulink gain blocks

The *select* operator for collections of Stateflow elements delegates to the MATLAB function in Listing 5.28. The question mark placeholders in this function can be replaced with property-value pairs to be matched from the elements in the collection. This MATLAB function supports more fine-grained queries than the `find_system` MATLAB function. For example, it supports multiple logical operators (i.e., AND, OR, XOR and NOT) to join property-value pairs and supports regular expressions for property values.

```
stateflow=handles.find('?','?')
```

Listing 5.28: Stateflow element selection MATLAB function

Listing 5.29 is an example of an EOL select operation that can be performed on collections of Stateflow elements. Listing 5.30 shows the MATLAB function that is constructed and submitted to the MATLAB engine, where `all` represents a collection of Stateflow handles.

```
1  `Stateflow.State`.all().select(s|
       (s.Name.startsWith('S')) and
       (s.IsExplicitlyCommented==0) or not
       (s.IsImplicitlyCommented==0)))
```

Listing 5.29: EOL selection of Stateflow states

```
1  all.find('-regexp','Name','^S','-and',
       'IsExplicitlyCommented',0,'-or',
       '-not','IsImplicitlyCommented',0)
```

Listing 5.30: MATLAB selection of Stateflow states

The *select* operator is reused by other Epsilon operators such as: *selectOne*, *find*, *findOne*, *reject*, *rejectOne*, *exists* and *forAll*.

5.3 Evaluation

This section presents a two-part evaluation of the Simulink-Epsilon drivers. The first part (Sec. 5.3.1) consists of an experiment that compares the performance of managing Simulink models directly via MATLAB functions or building an intermediate EMF representation with an upfront model-to-model transformation. This experiment was first published in previous work [162]. The second part of the evaluation is presented in Sec. 5.3.2 and compares the performance of collection operators executed on collection of Simulink and Stateflow model elements using the query optimisations described in Sec 5.2.2.

5.3.1 Experiment on Simulink models

This section evaluates the execution-time performance of two approaches to bridge MATLAB/Simulink models in a model management framework. The first approach consists in the use of the Simulink Model driver to manage models in the Epsilon model management framework. The second approach uses Massif facilities to transform Simulink models into an EMF-compatible representation. Since Epsilon provides an EMF driver able to read and write arbitrary EMF-based models, we use it to manage those produced by Massif in the second approach. In the following, we refer to the first approach as *live* —since it directly manipulates the actual Simulink model, and to the second one as *Massif/EMF* —as it uses the Massif's import facilities to produce their EMF-compatible representation.

Epsilon supports model element caching through an abstraction that both the Simulink Model driver and the EMF driver reuse. We evaluate both approaches with these facilities enabled and disabled. Note that at the time of this experiment, the query optimisations on Simulink and Stateflow elements had not been implemented.

Experiment setup

To evaluate the model management of Simulink models through both approaches, we compare the performance of their model validation process applied on large Simulink models. We have selected a model validation process as a representative model management operation, but other operations such as model-to-model or model-to-text transformations could have been used instead.

Validation process. This process is based on the execution of EVL invariants that validate structural properties of the models. EVL has a dedicated engine that consumes an EVL validation script, and any number of models provided by Epsilon drivers of arbitrary modelling technology at runtime. An example

of an EVL script is shown in Listing 5.31. This script starts by specifying the context in which the invariants are to be executed, in this case all elements of kind Block. Invariants may be of type *constraint* or *critique* depending on the severity level of a failed compliance. Line 2 of the script shows the declaration of an invariant of type *critique* with name `BlockNameIsLowerCase`. Invariants declare their validation *check* as an EOL statement, which in this case (line 3) verifies that the name of the element is lowercase. The `self` reserved word is a reference to the current model element the invariant is acting on. If a given block fails the `check` statement, then *fix* elements become available if present in the invariant declaration. In the script, the fix in line 4 updates the element name to lowercase as specified in the *do* environment (line 7). The fix title (line 5) is just informative.

```
1  context Block {
2    critique BlockNameIsLowerCase {
3      check : self.Name == self.Name.toLowerCase()
4      fix {
5        title : "Name to lower case"
6        do {
7          self.Name = self.Name.toLowerCase();
8        }
9      }
10   }
11 }
```

Listing 5.31: Sample EVL script with invariant 9 from Table 5.1

Before the EVL engine can execute the model validations, the models must be loaded. When the EMF driver is used to process an EMF model, the model loading stage consists in the registration of meta-model packages and creating an in-memory representation of the model. When the Simulink Model driver is used to process a Simulink model file, the model loading stage consists of establishing the connection with the MATLAB engine and requesting the model to be loaded there.

In the following we consider the model loading and validation execution as two different stages of the validation process. The overall validation process for each approach is captured in Figure 5.5 where loading and validation are represented by stages 1 and 2, respectively. In the Massif/EMF approach we consider the transformation of the model (from Simulink to EMF) as an additional stage of the validation process (Stage 0 in Figure 5.5). We refer to it as the import stage after the Massif facilities that enable this transformation.

The implementation of the Epsilon drivers and the structure of the meta-model used in the EMF driver affect the way the model is navigated in EOL-based programs. Consequently, the EVL validation script cannot be reused

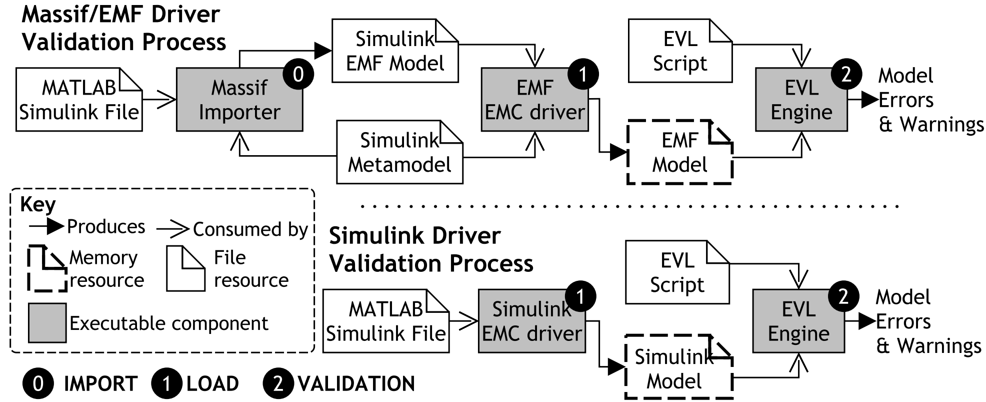


Figure 5.5: Model management execution process for both approaches, in this case, running a model validation with EVL.

as-is across approaches. To illustrate this, consider an EOL program that retrieves the `PortDimension` property of a block model element. When executed on a model managed with the Simulink Model driver, the EOL statement from Listing 5.32 can retrieve this property from an element of type block.

```
block.PortDimension;
```

Listing 5.32: Port dimension block property in EOL with Simulink Model Driver

In contrast, when using the EMF driver with the Massif meta-model, the statement needs to be adapted (as in Listing 5.33) because the `Block` class in the meta-model does not have a `PortDimension` attribute but instead has a `parameters` attribute containing a set of `Property` elements, one of them with the `PortDimension` identifier.

```
block.parameters.selectOne(p|p.name==
    "PortDimension").value;
```

Listing 5.33: Port dimension block property in EOL with EMF/Massif

In this experiment we measure the execution-time performance of the different stages of the validation process i.e., (0) Simulink-to-EMF transformation, (1) model loading, and (2) model validation. Notice that: Stage 0 is only applicable to the *Massif/EMF* approach; Stage 1 is applicable to both approaches; and Stage 2 is applicable to each approach with both the Epsilon caching facilities enabled and disabled.

Each stage of the validation process was executed 20 times with 5 warm-up iterations for each model. We used the Java Microbenchmark Harness (JMH) [145] tool to run these experiments on a quad core Intel Core i5-7200U CPU @ 2.5 GHz with 16GB of RAM. The Java Virtual Machine (64-Bit) was provided with up to 10GB of memory and ran Java 8 on JDK 1.8.0_152. All EMF-compatible models were generated using the *shallow* mode of the

Table 5.1: Evaluated invariants

#	**Kind**	**Context**	**Description**
1	PropertyCheck	Goto	TagVisibility property is local
2	NavigationAndFilter	From	There is a Goto block in scope with the name of the GotoTag property
3	PropertyCheck	Inport InPortBlock	PortDimensions property should not be inherited (-1)
4	PropertyCheck	Outport OutPortBlock	Description property is not null or empty
5	NavigationAndFilter	SubSystem	ForegroundColor property is green for all connected Inport blocks
6	TransitiveClosure	SubSystem	Subsystem is no more than three levels deep
7	VertexConnectivity	SubSystem	All outports are connected
8	LoopAbsence	SubSystem	No feedback. Outports do not connect to the same subsystem
9	PropertyCheck	Block	Block's name is in lower case

Massif import facilities which does not process external model references. The validation scripts and the Simulink models that were used in our experiments can be found in the examples of the Epsilon project[7].

Validation scripts. Equivalent EVL scripts are used to evaluate each approach. Each script consists of 9 invariants (see Table 5.1) intended to exercise the model (e.g., using different operations or navigation strategies) through typical query language features [170] performed on signature model element types [11]. The scripts are equivalent to the best of our knowledge as they are using (a) equivalent EVL *contexts* which may vary in naming across approaches (e.g., `Inport` vs. `InPortBlock`), (b) equivalent model element navigations (such as the `PortDimension` property discussed above), and (c) equivalent way in which the constraint checks and guards are prescribed. In Table 5.1 the *Kind* column refers to type of query check inspired by well-formedness constraint categories used by the Train Benchmark [170], and the *Context* column refers to the EVL context, that is, the model element types on which the invariant is executed. Stateflow blocks were not included in the validation scripts as Massif does not support them.

The validation scripts for the *live* approaches used 96 lines of code (LOC) and that for the *Massif/EMF* approach used 110 LOC. The body of the invariants was written in the same number of lines for both approaches (89 LOC) and the extra lines were related to helper operations.

[7] https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/examples/org.eclipse.epsilon.examples.emc.simulink.emf

Table 5.2: Number of elements per type by MATLAB model file size (MB).

Size (MB)	Block	Inport	Outport	Goto	From	SubSystem
1.112	8785	1373	1177	69	103	717
1.131	8628	1372	1167	62	93	740
1.133	8645	1372	1167	62	93	740
1.134	9536	1489	1269	38	57	861
1.135	8645	1372	1167	62	93	740
1.138	8651	1376	1177	62	93	745
1.141	8634	1374	1156	67	99	714

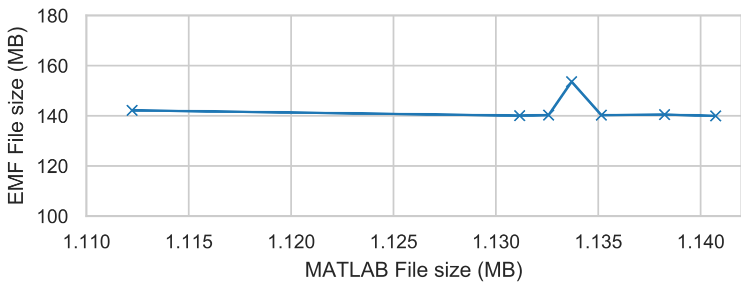


Figure 5.6: Imported EMF model size vs. original MATLAB files.

Model selection. We used BigQuery [63] to find in GitHub publicly available Simulink files (*.slx) larger than 1MB[8]. Out of the 70 models found, we selected the first 7 models that could be translated into EMF in under 2 hours using Massif's import facilities. Table 5.2 shows the number of model elements of each type used in the validation. The number of block elements on the models ranged from 8628 to 9536. Due to their inaccessibility, we did not process any libraries in any approach.

Results

All invariants were executed in the same number of model elements for all approaches. Similarly, the results of the validation reported the same number of unsatisfied constraints on all approaches. The file size of the EMF models produced by the import stage are displayed in Figure 5.6, plotted against the size of the original MATLAB file.

Figure 5.7 shows the execution time of each stage of the model validation process (in seconds and logarithmic scale) against the size of the MATLAB Simulink model files (in MB). Sub-figure (a) displays the distribution of Massif's *import* task (Stage 0) which transforms Simulink models into an EMF-compatible

[8]We had access to one industry model that was 1.4MB in size but for the experiment we had to find others in public repositories. To increase our chances to find complex models and to facilitate the collection procedure, we looked for models persisted in a file larger than 1MB in size.

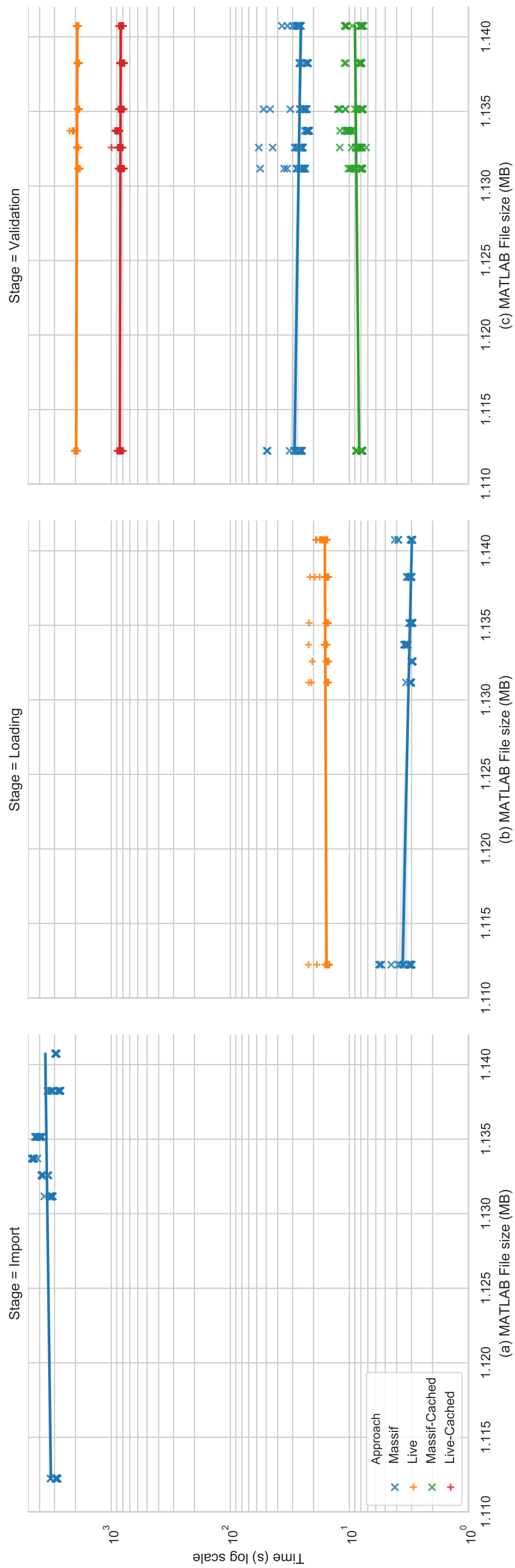


Figure 5.7: Execution time (log-scale) against MATLAB file size per stage of the validation process.

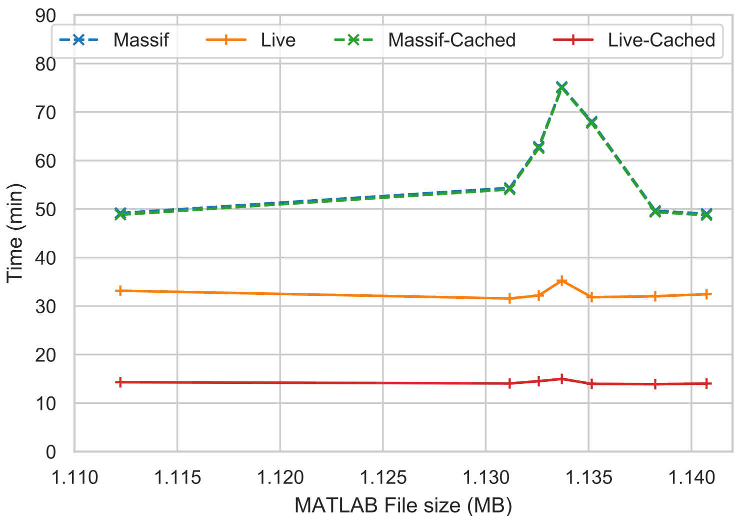


Figure 5.8: Total execution time (import + loading + validation) vs. MATLAB file size. *Note that Massif and Massif-Cached overlap.*

model. Similarly, Sub-figure (b) displays the time distribution of the model loading task (Stage 1), required by both the EMF and Simulink Model drivers. Sub-figure (c) displays the time distribution of the model validation task (Stage 2) for both approaches with and without caching.

Figure 5.7 shows that most of the performance overhead of the Massif/EMF approach happens at the import stage while most of the Simulink Model driver overhead happens at the validation stage. The import stage of the Massif/EMF approach took between 2,911 and 4,486s to finish. The Massif/EMF approach achieved the loading stage in 2.95-3.63s while the Simulink Model driver achieved it in 15.5-16.5s. The live approach was approximately one order of magnitude slower at the loading stage. In the validation stage, the Massif/EMF approach took between 22.4-28.9s while it took the Simulink Model driver 1,877-2,098s to complete. With caching facilities enabled in both drivers, the Massif/EMF approach took 8.10-10.2s while the Simulink Model driver took 816-882s to finish. With and without caching, the live approach was approximately two orders of magnitude slower at the validation stage. The caching facilities improved the performance in the validation stage by 54.4-72.0% in the Massif/EMF approach and 55.3-58.0% in the live approach.

Figure 5.8 shows the whole validation process execution-time (in minutes) calculated using the sum of averages of each stage for each approach with and without caching. By comparing this overall process, we observe that the live approach improves the performance of the Massif/EMF approach by taking 70.7-80.0% less time when caching is enabled and by 32.6-53.2% with no caching.

In Figure 5.6 we observe that the size of the EMF model produced by Massif is much larger than the original MATLAB/Simulink (.slx) files. This is partly

due to *.slx being a compressed file format. As Table 5.2 shows, the size of the MATLAB/Simulink file is not directly proportional to the number of Block[9] elements in the model. In contrast, the size of the EMF model file seems to be related to the number of block elements, which would explain the peak on the EMF file size with the MATLAB/Simulink model with the largest number of block elements.

Discussion

In this experiment we focused on a program that only reads large Simulink models. We intended to investigate the performance of using of both approaches with large models. In this subset of models, our experiment shows that the overhead of the Massif/EMF approach lies on the upfront model transformation whereas for the Simulink EMC it lies in the complexity of the model management program. In contrast, the actual execution of the program with the EMF driver works much faster than with the Simulink EMC driver. This is partly due to the full model being loaded in memory and potential internal optimisations of the mature EMF driver.

Intense querying is a scenario for which the EMF approach is more suitable, as the communication with MATLAB is expensive in time, and our experiment shows the clear advantage that the EMF driver has over our Simulink implementation. However, our experiment also shows the non-negligible impact that the importing stage has over the overall execution. Choosing one approach over the other is a matter of determining the size of the model, understanding the purpose of the model management program, and being aware of constraints such as performance or model coverage. For example, it is likely that large models will incur in computationally expensive import procedures with Massif. Whether this is a sensible cost depends on the number of times the import is to be executed, the available time, the model management framework to be used i.e., if it only supports EMF and the range of operations to be performed (e.g do they require Stateflow blocks?). To avoid the cost of the import process on continuously evolving models, a practitioner may choose to manually replicate modifications in the Simulink model in the already imported EMF copy, however this would be an error-prone activity.

With the same large models, our implementation avoids the import/export procedures when the models are evolving e.g., changing property values, adding new blocks or removing blocks. Indeed, intense querying is not the best use scenario for our driver as demonstrated by the experiment. With the knowledge of the new query optimisations, the validation scripts used in the experiment could be rewritten to take advantage of these optimisations to reduce cost of

[9] Inport, Outport, Goto, From and SubSystem are all subtypes of Block

the validation stage.

In Section 5.3.2 we show how the driver can be used to generate Simulink models. Further investigation would be needed to show how the Massif approach copes with continuously evolving models and programs that modify or create the Simulink model. Validation scripts in EVL can also feature `fix` constructs that invoke EOL expressions on the elements that do not pass the constraints. While we have not evaluated this, we can anticipate that the validation step with fixes would require little additional time for both the Simulink EMC driver and the EMF driver. The difference would be that the overall validation process with Massif/EMF would require an additional step to generate the modified Simulink model from the modified EMF which could potentially be just as expensive in time as its import procedure.

Threats to validity

We selected a validation program as a representative model management operation to compare both approaches. As indicated in the *Validation scripts* paragraph, the invariants used in the experiments were intended to exercise the models in similar ways in both approaches by means, for example, of interacting with the same types of elements and navigating properties in similar ways. As such, the invariants were not intended to be representative of validations performed in industry, although some were inspired by industrial cosmetic checks. Validations performed in EVL can be seen as complementary validations as Simulink models can go through custom validation checks within MATLAB using its Model Advisor tool.

Our evaluation only tested the performance of a single model management language (EVL). Performance results may vary across other types of model management programs and for different EVL programs. Moreover, the validation scripts were limited to read-only operations.

The sample of models may not be significant but was limited by the 2-hour cap imposed to the import stage. Our experiments would benefit from more diverse models with a broader range of sizes and more varied constraints.

There may be hidden differences in the implementation of each driver (EMF vs Simulink) such as internal optimisations which do not make them entirely comparable. However, for the purpose of this experiment, both driver implementations were considered black boxes.

Large and complex models can be built by referencing multiple models persisted in small files. Our decision to use large models allowed us to skip the model reference processing by ensuring that a single model contained the most model elements. Additional metadata other than model elements, such as images, can contribute to the model file size without affecting its complexity.

We have not measured the impact of the meta-information in the file size, but this is mitigated by indicating the number of model elements that were present in each file.

5.3.2 Experiment on collection queries

We have designed an experiment that evaluates the performance of the collection operator optimisations presented in Sec 5.2.2. The research question is whether these modifications improve the performance of select- and collect-based operators when executed on collections of Simulink or Stateflow elements of different sizes. All resources required to reproduce the experiment are available under the Epsilon project[10].

Experiment setup

This experiment includes the evaluation of EOL queries on collections of Simulink and Stateflow model elements. We execute each query on four models with a similar structure but with different number of model elements that grow exponentially. For each query and model, we observe how the use of the query optimisations on collections affects the execution performance.

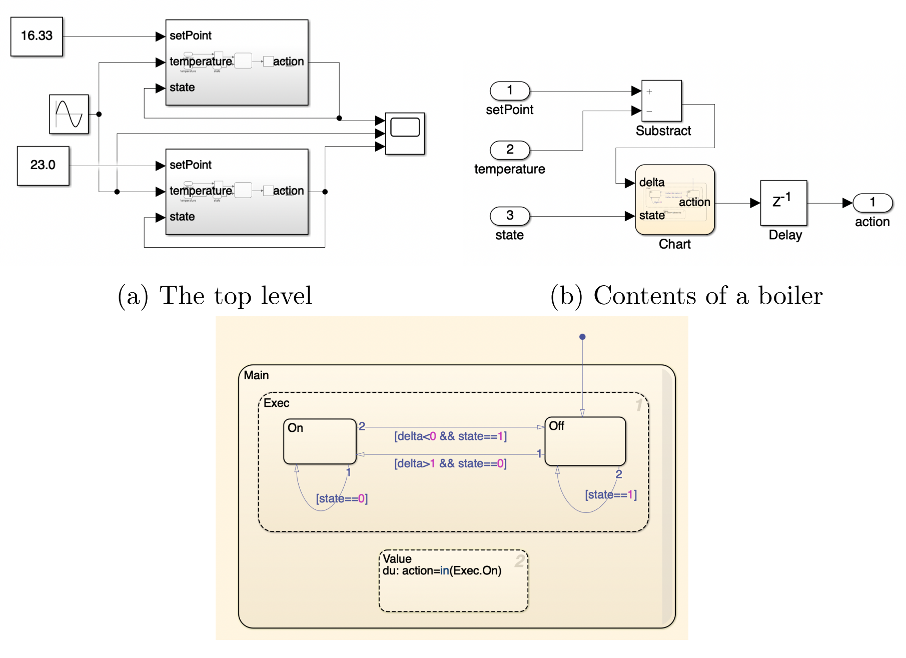
As we need to have control over the number of elements of a given type on each model, we decided to generate the test models. As such, the models share a similar structure but have some variability which is described later in the section. While the generation script is not part of this evaluation, it serves to demonstrate the write capabilities of the Simulink Model driver.

Model generation process A boiler control system can be designed using an on/off closed loop control. Closed loop control systems are very common, and they can be designed and simulated using the Simulink environment. Furthermore, on/off controllers are easy to model as state machines which can be designed using MATLAB's Stateflow environment. Since boiler systems can contain both Simulink and Stateflow model elements, we use them at the core of our model generation process.

The model generation process consists in producing several contrived components with different set points[11] all receiving the ambient temperature from a pulse generator and displaying their status in a scope. To scale our experiment, each model has a different number of boilers which grow exponentially (base 3) and the value of their set point is spread out so that each has a different value within their operational range. At the same time, each boiler has only one pulse generator and scope. Four models were generated in total.

[10] https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/tests/org.eclipse.epsilon.emc.simulink.test/experiments/query-optimisation

[11] The temperature at which they start to heat



(a) The top level

(b) Contents of a boiler

(c) Contents of a boiler's Stateflow chart

Figure 5.9: Structure of generated Simulink models

Figure 5.9a illustrates the root level of the model where all boilers receive as input the ambient temperature from a pulse generator and display their operational state in a scope. The set point of each boiler is represented by a block of type constant with the temperature value. The internal structure of a boiler is illustrated in Figure 5.9b. Each of them has three input ports and an output port. The inport that receives the set point is compared with the current ambient temperature using a block of type substract, whose output goes into a Stateflow chart. The contents of a chart are illustrated in Figure 5.9c. The chart computes the logic to go from state ON to state OFF and produces a signal that decides whether it is required to turn on or off the boiler. The action which results from the chart logic goes into a delay which represents the time taken for the real boiler to respond to the signal. The delayed signal is displayed in the topmost scope and the one which is used as feedback on the boiler subsystem and chart.

Queries The list of EOL statements to be evaluated are presented in Listing 5.34 where line numbers are used as query identifiers. These queries were designed to demonstrate both the usefulness of retrieved information from the boiler model, and the complexity supported by the query optimisations on collections. Four of these statements are executed on collections of Simulink elements while the other four are executed on collections of Stateflow elements. The queries use EOL select- and collect-based operators both in plain form e.g.,

select; and derived form e.g., *exists*, *sortBy*, *reject*, *forAll*. While most of the queries use single operators that evaluate one-argument predicates, Query 6 uses two operators (*select* and *forAll*) and Query 8 evaluates a three-argument predicate.

```
1  Block.all().collect(b|b.Name);
2  Block.all().sortBy(b|b.BlockType);
3  Inport.all().select(i|i.OutDataTypeStr=="boolean");
4  SubSystem.all().selectOne(s|s.Name=="Chart");
5  `Stateflow.State`.all().reject
       (s|s.Decomposition=="PARALLEL_AND");
6  `Stateflow.Transition`.all().select(t|not
       (t.SourceOClock==0)).forAll(t|t.LabelString<>"?");
7  `Stateflow.Transition`.all().collect(t|t.LabelString);
8  `Stateflow.State`.all().exists
       (s|s.IsImplicitlyCommented==1 or
       s.BadIntersection==1 or s.IsExplicitlyCommented==1);
```

Listing 5.34: List of EOL queries

Query 1 is used to retrieve the names of all Simulink blocks in the model, including those contained in the boiler subsystems. Query 2 sorts all these blocks by their block type. Query 3 acts on blocks of Inport type i.e., input ports 1 to 3 in each boiler subsystem (Figure 5.9b), and filters those of Boolean type i.e., port no. 3 which handles the boiler state. Query 4 acts on subsystem blocks which include the boilers and the chart blocks and selects the first element with the name “Chart”. Moving on to Stateflow elements, the list of non-parallel states is retrieved with Query 5 using the reject operator. Query 6 starts by filtering out default transitions i.e., those with no source state, and then checks if they have all been assigned a non-default name using the exists operator. In a similar fashion to Query 1, Query 7 retrieves the labels attached to all transitions in the model. Finally, Query 8 checks for malformedness across Stateflow states by checking whether they are explicitly or implicitly commented or if they have bad intersections.

Model population Our experiments evaluate the 8 EOL statements on four different models. Each evaluated EOL statement starts from a collection of model elements of a given type. These model element collections may contain Simulink elements of type Block, Inport or SubSystem; or Stateflow elements of type Stateflow.State or Stateflow.Transition. The number of elements of each type in the different models is presented in Table 5.3.

Infrastructure In the experiment each EOL statement was executed 20 times with 5 warm-up iterations on each model. The Simulink Model driver

Table 5.3: Number of elements per type on each model.

	Model 1	Model 2	Model 3	Model 4
Block	47	137	407	1217
Inport	15	45	135	405
Stateflow.State	15	45	135	405
Stateflow.Transition	15	45	135	405
SubSystem	6	18	54	162

caching facilities were not used. The experiments were executed on an 8-Core Intel Core i9 CPU @ 2.3 GHz with 16 GB of RAM. The Java Virtual Machine (64-Bit) was provided with up to 2GB of memory and ran Java 8 on JDK 1.8.0_231.

Results

In both optimised and non-optimised executions, all queries were executed on the same number of elements and yielded the same results.

The mean execution time of each query is presented in Table 5.4 under the Duration section. This section compares the time (in seconds) taken by each of the models with and without the collection operator optimisations. The iteration distribution on the four models is presented in the box plot of Figure 5.10. This figure compares the distribution with optimisations enabled (right/orange) and disabled (left/blue) for each model. Note that subplots do not share the y-axis to have a closer look at the distribution per query.

Regardless of the collection size, all queries with optimisations enabled outperformed those which did not use them, between 50% to 99%. Table 5.5 summarizes the performance improvement percentage that optimisations achieved on the different models and queries.

Another view of the results is presented in Figure 5.11 where the mean execution time per query is plotted against the number of model elements that the query acted on. The y-axis in this view has been capped at 40 seconds and only Query 8 went above this limit.

Additionally, Table 5.4 shows (under the MATLAB Communication section) the percentage of execution time that was spent sending or receiving information to/from MATLAB. Overall, this section shows that without operator optimisations the impact of the communications with MATLAB lies above 93% whereas with optimisations the impact can be reduced to 59% in some queries although remaining high (e.g., 99%) in others.

Q	Opt	Duration (s)				MATLAB Communication (%)			
		Model 1	Model 2	Model 3	Model 4	Model 1	Model 2	Model 3	Model 4
1	Off	0.15	0.38	1.06	3.39	94.74	96.64	97.32	97.61
	On	0.00	0.00	0.01	0.01	76.07	75.24	76.47	73.99
2	Off	0.20	0.55	1.73	4.96	97.21	97.70	97.95	97.79
	On	0.09	0.24	0.86	2.23	96.92	98.10	98.18	98.50
3	Off	0.06	0.14	0.35	1.08	95.11	96.69	97.06	97.24
	On	0.00	0.00	0.00	0.01	75.44	73.23	68.67	59.88
4	Off	0.02	0.06	0.16	0.47	93.60	96.77	97.44	98.06
	On	0.01	0.01	0.01	0.01	86.22	84.45	81.85	75.24
5	Off	0.12	0.39	1.93	14.87	95.59	97.05	98.13	99.07
	On	0.01	0.02	0.12	0.91	89.03	95.15	98.88	99.78
6	Off	0.70	2.28	8.33	38.49	98.80	98.97	99.15	99.39
	On	0.03	0.04	0.16	1.25	95.00	97.09	99.06	99.83
7	Off	1.03	3.13	10.17	38.17	99.31	99.38	99.49	99.58
	On	0.02	0.04	0.14	1.02	95.63	97.42	99.09	99.81
8	Off	2.72	8.50	28.04	109.40	99.49	99.49	99.55	99.60
	On	0.03	0.05	0.15	0.98	96.76	97.79	99.17	99.82

Table 5.4: Mean query execution time in seconds and percentage of time spent sending commands to MATLAB and awaiting a response. Column Q indicates the query number, while column Opt indicates whether the optimisations were enabled.

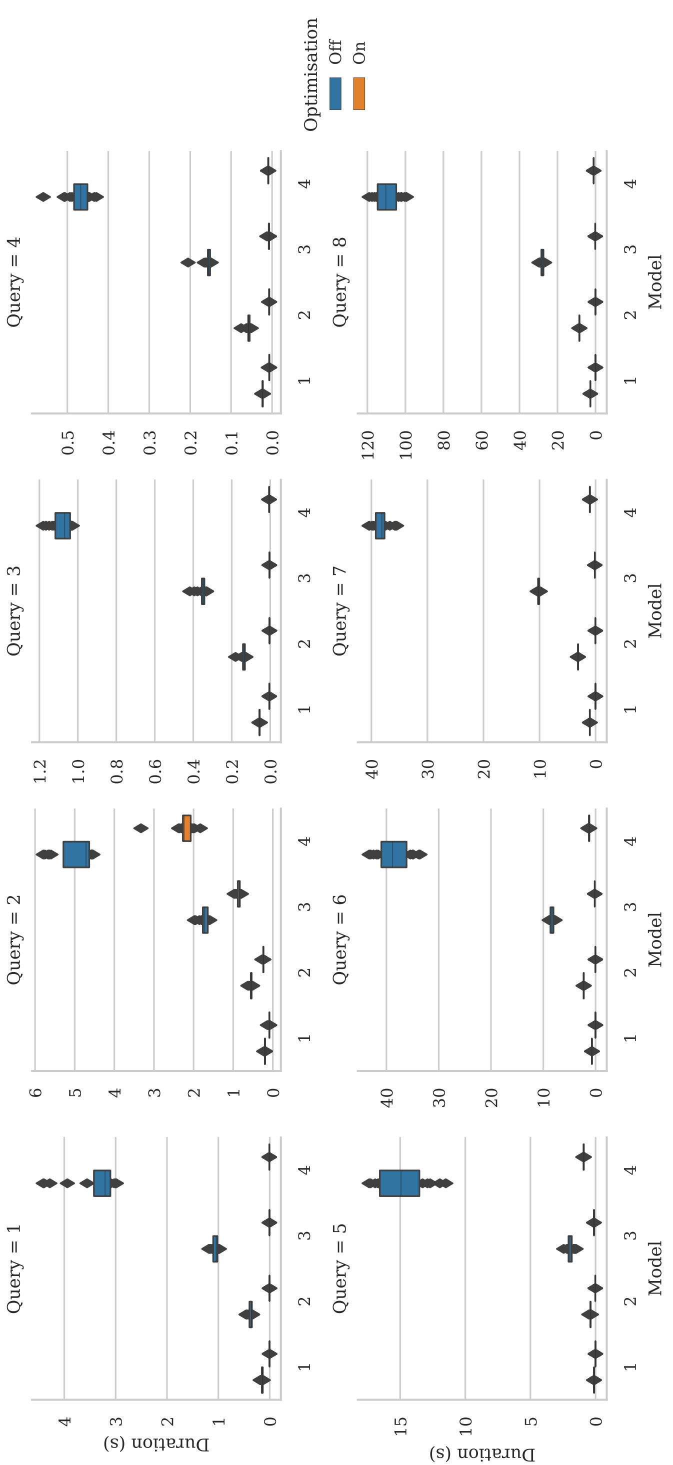


Figure 5.10: Distribution of the query performance on the models with and without optimisations.

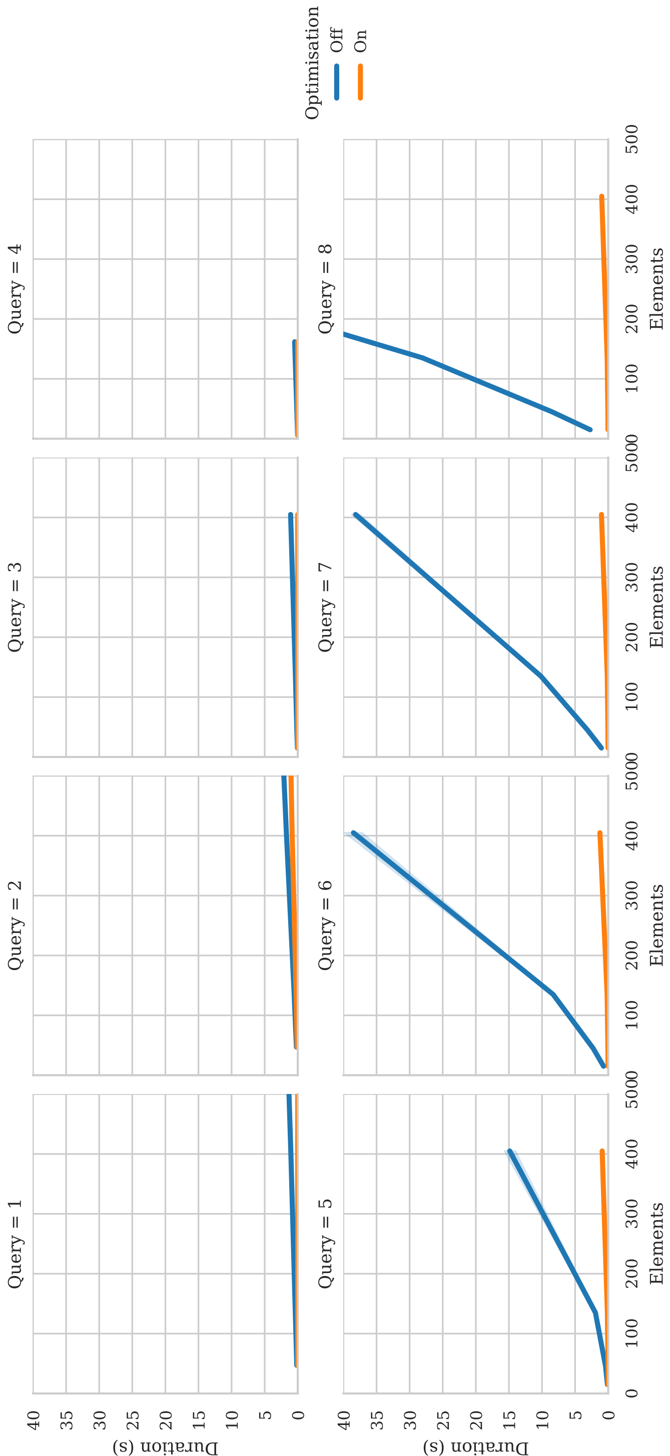


Figure 5.11: Performance of queries, with and without optimisation, against the number of elements in the models.

Table 5.5: Performance improvement (%) by query and model.

Query	Model 1	Model 2	Model 3	Model 4
1	97.52	98.95	99.43	99.76
2	55.03	56.15	50.20	55.09
3	91.80	97.20	98.82	99.48
4	69.92	87.68	94.95	98.01
5	91.73	93.95	93.71	93.88
6	96.36	98.11	98.07	96.75
7	97.58	98.72	98.64	97.33
8	98.78	99.45	99.46	99.10

Discussion

The first four queries acted on Simulink elements while the last four acted on Stateflow elements. Non-optimised queries were more time-consuming on Stateflow elements than on Simulink elements regardless of the complexity of the evaluated expression. Consider queries 1 and 7 which are comparable as they both invoke a collect operation that gathers a single property value but work on Simulink blocks and Stateflow transitions respectively. Even though in Model 4 query 1 acts on 1217 blocks while query 7 only on 405, query 7 is much more expensive in time than query 1 (without the optimisations). Since more than 98% of the execution time of Stateflow queries without optimisations is spent on the MATLAB exchange, a reasonable explanation for this difference is that MATLAB has more efficient indexes for Simulink blocks.

Based on preliminary observations, executing the functions that the driver generates in the MATLAB console is much faster than through its Java API for both the optimised and non-optimised implementations. Considering the impact that reducing the number of exchanges with the MATLAB Java API has, future work will involve investigating optimisations of more complex collect- and select-based arguments so they can be transformed into a single complex MATLAB function that only requires to be sent once.

To take advantage of these optimisations, the model management programmer should be aware of the operations that have been optimised to write the programs accordingly. A difference with the Massif/EMF approach is that in that approach there are no optimisations to be aware of.

Threats to validity

The models used in the experiment had a similar internal structure as it enabled us to focus on the impact of the number of model elements that the queries acted upon. From this experiment, it is unclear to what extent the structure of the models affects the performance.

We chose a range of collection queries that were sufficiently varied, and which could be optimised. We recognise that our evaluation could be complemented with more queries evaluating a broader range of expression forms.

5.4 Observations and lessons learned

This section summarises observations and lessons learned in the implementation of the Simulink-based drivers and our experiments.

Usability. Being able to manage these models in either the native tool or a model management framework requires metamodel understanding (model element types, their properties, and operations). Model management programs should provide uniformity and predictability in how model elements are managed as part of the conciseness and expressiveness they offer compared to general-purpose languages. For example, in Epsilon CRUD operations on model element types share the same syntax regardless of the model's underlying technology. This enables practitioners to focus on the model elements and the logic of their programs.

Uniformity can help to speed up the learning process and make these programs easier to write and maintain. Sec. 5.2 evidences the multiple styles that MATLAB uses to manage different model elements types, within the same model e.g., Simulink vs. Stateflow, and between different model formats e.g., Simulink vs. Simulink Requirements. It is not just the naming of the MATLAB functions that varies across operation types (e.g., `get` as property getter for Stateflow elements and `get_param` for Simulink elements), but also the arguments required by those functions. Similarly, different toolboxes use different notions of what constitutes an element id in their domain e.g., Simulink sometimes uses the element id but most functions only work with their path property (their location) or their handle (a session based, non-persisted identifier). Furthermore, in the case of Simulink different parameters sometimes yield different result types e.g., the `find_system` function can return handles or paths depending on whether the `FindAll` flag is active. A side-contribution of our approach is the unification of the syntax of several MATLAB toolboxes which can make it easier to focus on the core model management logic.

Completeness. MATLAB and its Java API provide facilities to support the execution of CRUD operations on its model elements and the model itself. This API also provides an interface for a few MATLAB-specific data types such as structured arrays. In contrast, Simulink models cannot be exported into any exchange format from MATLAB. It is common that vendor tools are reticent to export their models into exchangeable data formats e.g., to protect their

intellectual property. However, when they do export them, sometimes they do so partially —like PTC with partial exports [198], which can make the round-trip engineering prohibitive (e.g., [198]) or complex (e.g., [120]).

In the case of Massif, the Simulink to EMF transformation is done by an external party. Among the disadvantages of this transformation is the lack of support for Stateflow elements and slightly different naming conventions to the ones used in MATLAB, different places to find element properties depending on the element type and the management of Simulink data types as strings. In contrast, model element types used in the Simulink EMC driver are closer to those managed by the MATLAB command line interface and include Stateflow elements. In addition, by exploiting the MATLAB API facilities at runtime our Simulink EMC driver can also manipulate MATLAB specific data types.

Performance. Several criteria can impact the performance of model management processes that involve Simulink models e.g., the size of the model, the program complexity, and the rate of model evolution. Our first experiment on large Simulink models showed that the cost of the upfront Simulink-to-EMF transformation in time was particularly expensive in the Massif/EMF approach while the cost of the program execution time was much lower than that of the Simulink EMC driver (by 2 orders of magnitude). Considering the program execution performance, the Massif/EMF approach seems convenient for large signed-off models (transformation cost paid once) that need to be extensively queried. In contrast, this same experiment showed that the overall execution process was reduced by up to 80% with the Simulink EMC driver, which concentrated the time cost in the program execution. The overall execution performance makes the Simulink EMC driver better suited for continuously evolving models; otherwise recurrent transformations would be needed in Massif/EMF. We anticipated that the execution overhead in our approach was due to the time cost of the MATLAB exchanges. Our proposed optimisations on operations on collections of Simulink model elements (Sec. 5.2.2) were able to reduce the number of MATLAB exchanges by not making them proportional to the collection size.

For smaller models, the decision of one approach or the other is more related to the model coverage offered by the approach and the relevance of the EMF model i.e., its support in the model management tool and associated maintainability costs.

Other Model validation processes generally involve several iterations of checking constraints and fixing errors unless the model is correct to start with. Similarly, model-to-model transformation and other model management programs may also result in the generation or modification of Simulink models.

From our experiments the performance impact and completeness of the EMF-to-Simulink transformation is unclear although it is likely to have similar time costs as the import procedure and similar issues to those found in other tools such as those mentioned for the ReqIF requirements imported by MATLAB [120] or the XMI models exported by PTC [198]. Our on-the-fly approach does not need to incur in round-trip engineering costs as it directly acts on the models themselves.

Our piecewise translation of model management constructs to MATLAB is convenient to deal with multiple (heterogeneous) models in the same model management program and to process the model information within the managing program. A complete translation of these constructs to a MATLAB program that executes just once would be more complex to orchestrate and to interact with from the model management program e.g., to retrieve variable values that are assigned to elements from other models. The stark performance difference between the execution of MATLAB functions in Java or in its console suggests that further optimisations and strategies are required to reduce the number of exchanges with MATLAB and improve the performance of model management programs while still preserving their ability to interact with other models.

5.5 Related work

It is often desirable to have a common framework to manage models from heterogeneous modelling technologies. Traceability tools such as Capra [112] and Yakindu [1] are examples of those frameworks, which need to be able to read models used at different stages of the development process to create and manage traces among their model elements. Other examples include model management frameworks such as Epsilon [97] and ATL [87], which offer a subset of task-specific languages for model navigation, validation, model-to-model or model-to-text transformations, etc. and which are able to interact with a number of models of arbitrary underlying technologies.

When model management frameworks do not offer support for a specific modelling technology such as Simulink, import and export facilities can be used to translate the models into a supported format. Possibly for reasons of protecting intellectual property, proprietary modelling tools do not always offer exporting facilities into open modelling formats such as XMI. MATLAB does not offer any export or import facilities for Simulink Models with other open-source modelling formats. To address this feature gap, the open-source Massif project led the development of import and export facilities between EMF and Simulink models. Massif internally uses MATLAB's command line interface to parse the Simulink models and populate their EMF representation and vice-versa.

The OSLC [144] is an initiative that aims to simplify the software tool integration problem among proprietary tools. Built atop the W3C Resource Description Framework (RDF), Linked Data, and the REST architecture, OSLC provides a set of specifications targeted at different aspects of application and product life cycle management. OSLC is now being used by proprietary tool vendors (e.g., IBM Rational DOORS [80]) and some open-source tools (e.g., [50]) who expose a range of services following these specifications. Nevertheless, the comprehensiveness of the information exposed by these services is at the discretion of the service provider. MATLAB does not officially provide an OSLC interface for its Simulink models, although the Eclipse Lyo [178] project provides an OSLC adaptor for Simulink [48] for MATLAB version R2013b, and Massif provides an OSLC adaptor for their EMF-compatible representations [78]. Reqtify [33] is a proprietary tool which exposes internal traceability information from Simulink models in a similar fashion to OSLC.

Transformations from SysML to Simulink models (and vice-versa) have motivated several research works such as [164, 35, 27, 36]. [164, 35, 36] and [27] made use of model-to-text transformations with Acceleo [179] to produce MATLAB programs that on execution created the Simulink model. More specifically, [35, 36] generated several MATLAB scripts to populate different parts of the Simulink model, [27] proposed the use of a UML profile to annotate the SysML models before the MATLAB code generation, and [164, 36] suggested that to go back from Simulink to SysML the creation of a MATLAB script to parse Simulink models and produce an XML-based SysML model description file. In the domain of co-simulation, communicating between MATLAB Simulink and other frameworks is a common task. For example, [51] uses a software environment based on Ptolemy II [49] to run MATLAB scripts that get and set parameters of specific Simulink blocks and run simulations. As these works either use purposed SysML to Simulink transformations or focus on setting and getting parameter values of limited elements; they are not easily reusable for alternative model management scenarios such as querying the Simulink model or validating constraints. Examples of other works that used Simulink models external model management processes include [123] which performs independent translation of Simulink and Stateflow blocks into UPPAAL timed automata representations that are later combined and used in model checking and [54] which performs invariance checks on simplistic Simulink model representations written in JSON. In this regard, the Massif project and our approach facilitate the managing an EMF-compatible representation or the actual Simulink model (respectively) in a broader range of model management scenarios.

Our Simulink bridge built atop the Epsilon facilities is not the first one to bridge proprietary tools with the open-source model management languages of the Epsilon family. In [58] a spreadsheet driver was introduced to enable the

manipulation of spreadsheets as models where element types were resolved from spreadsheet names, elements from rows, and properties from columns while enabling flexible rules to resolve element references or change these conventions. Our approach is closer to that used by the PTC-IM driver presented in [198], where an interface with the PTC is used to manage the models. One difference with the PTC driver is that in MATLAB the API is not consistent and required commands to be built on demand. Additionally, MATLAB has a full-fledged language to manage its model elements that PTC does not, which allowed us to implement query optimisations. As in this work, one of the findings of [198] is that where performance is of essence, it is best to use the native tooling. In [198] the driver is evaluated against the native approach to manage the models by the tool i.e., Visual Basic. In contrast, in this work our first experiment compares two different approaches to bridge Simulink models with model management frameworks, while the second experiment evaluates an approach to reduce the overhead of queries while also measuring the cost in time of communicating with MATLAB. Another driver for relational databases was proposed in [105] which generated SQL queries at runtime. The main difference between this approach and ours is the domain of application and non-uniform MATLAB API used to manage different model types. [105] investigates the use services provided by the underlying technology to optimize those provided at the proxy level in a similar fashion to what we do in this work although no evaluation is provided.

5.6 Integration with ModelFlow

To use the Simulink model driver in ModelFlow a dedicated model definition is required. To configure a Simulink model, the model definition accepts as configuration properties the location of the Simulink model, of its Simulink project (optional) and of the engine and library path that ensure the connection to MATLAB. In addition, the model definition needs to support the computation of a custom hash of the model to determine if it is up to date and must ensure that model elements can be uniquely identified to make these traceable.

```
1 model Simulink is epsilon:simulink {
2   src:       <SimulinkModelLocation>
3   project:   <SimulinkProjectLocation>
4   engine:    <engineJarLocation>
5   library:   <libraryPathLocation>
6 }
```

Listing 5.35: Example Simulink model declaration in ModelFlow

Determining model up-to-date status. To compute the hash of the Simulink model we first need to identify any dependencies in the form of referenced Simulink models, Simulink Requirements, Simulink Dictionaries, MATLAB function scripts, etc. We compute this information using the MATLAB function shown in Listing 5.36.

```
1  dependencies.fileDependencyAnalysis('ModelName')
```

Listing 5.36: MATLAB function used to compute dependencies of a Simulink model.

The dependencies resolved by this function will change depending on whether the model is part of a Simulink project and if it is loaded within the project's context. As such, it is important to provide a project in Simulink model declarations when appropriate to ensure that the model is properly loaded and that dependencies can be resolved.

For each of the files returned by the MATLAB function, the Simulink model definition computes a message digest of the file contents with the MD5 algorithm. The hash of the Simulink model is a hash of hashes (of the model and its dependencies) to their corresponding message digests.

In subsequent executions where the model has not been loaded the Simulink model definition computes new message digests from the file paths of the map produced in the past execution.

Supporting model management traces. To ensure that model elements can be uniquely identified we have updated the Simulink driver to have the ability to identify elements by id. To compute this value, the driver delegates to the MATLAB engine the execution of the MATLAB function in Listing 5.37 takes as input an element's handle. This function works for both Simulink and Stateflow elements.

```
1  Simulink.ID.getSID(elementHandle)
```

Listing 5.37: MATLAB function used to compute the ID of Simulink and Stateflow model elements

6 Evaluation

This chapter presents the evaluation of ModelFlow through three case studies. The first case study is presented in Sec. 6.1 and it presents a contrived case study of a workflow with three models and three tasks. This case study evaluates the correctness and performance of ModelFlow executed under scenarios that change different resources used or produced by the workflow. The case study also compares the workflow language and execution in ModelFlow against those with the Gradle build tool. Sec. 6.2 presents the second case study which reproduces the graphical editor generation workflow used by the EuGENia tool, a graphical editor generator from annotated metamodels which predates this work. This case study further evaluates the correctness and performance of ModelFlow but this time compared against the original implementation and executed under realistic change scenarios for the specific workflow. Furthermore, this case study also provides measurements of the overhead of ModelFlow's features, such as model management tracing, and input and output model and parameter processing. The last case study is presented in Sec. 6.3, and it describes a sanitised industrial workflow that involves heterogeneous models including Simulink and HTML models. This qualitative study is mostly used to validate the language and the model management traceability support provided by ModelFlow but also provides insights regarding the conciseness of the workflow specification, its visualisation and user interaction facilities, the recovery of model management traces, limits to conservative executions, and it also discusses potential ModelFlow language optimisations. Finally, Sec. 6.4 and 6.5 describe how ModelFlow satisfies the extensibility and interoperability objectives defined in Sec. 3.2.2.

6.1 Case study: Component workflow

This section describes a contrived workflow that consists of three models and three tasks, each task has different complexities which increase with the number of elements in the source model. In this case study the workflow will be executed under different change scenarios that alter the contents of some of the resources used or produced by the workflow e.g. models, model management programs, generated files. The main goal of this case study is to evaluate the performance of this workflow under the different change scenarios and across two platforms:

Gradle and ModelFlow. The case study will also check the correctness of each scenario execution by ensuring that tasks affected by the changes are actually executed and that all required outputs are generated. Sec. 6.1.1 describes the background of the workflow, Sec. 6.1.2 presents the experimental setup, then Sec. 6.1.3 reports the results of the evaluation, while Sec. 6.1.4 provides the discussion and Sec. 6.1.5 reports on the threats to validity.

6.1.1 Background

Our motivating example describes a simplified process for generating a Java implementation of a component-based system. The process consists of three model management tasks: validation, model-to-model transformation, and model-to-text transformation. The dependencies between tasks, models, metamodels and file resources are illustrated in Figure 6.1. For simplicity, all models and metamodels in this example are built with EMF.

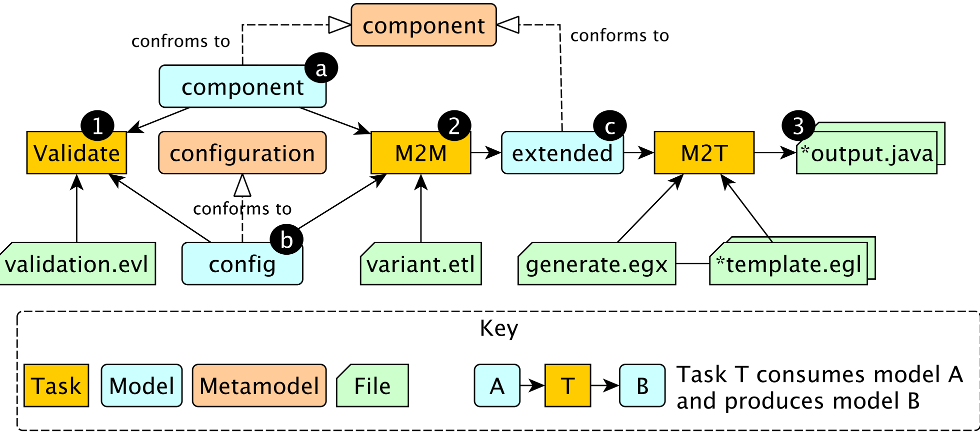
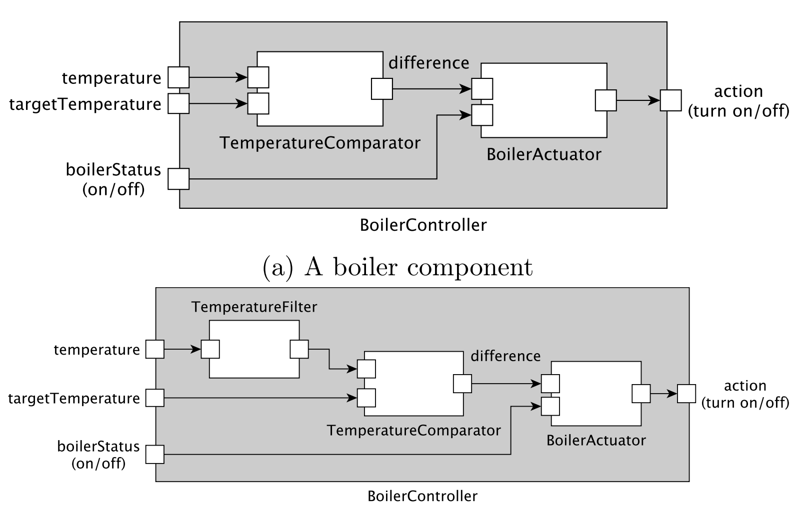


Figure 6.1: Component workflow dependency graph

The *component* model ⓐ represents a set of interconnected component blocks such as the one in Figure 6.2a. The goal of this process is to produce a variant version of the *component* model, referred to as the *extended* model ⓒ, that will later be used to generate code. An example of this variant model is shown in Figure 6.2b. The information used to produce this variant is captured in a *configuration* model ⓑ. The metamodels used by these models are shown in Figure 6.3.

In this workflow the *extended* model is a copy of the *component* model but with additional filter components. The configuration of these filters is defined in `Tolerance` elements in the *configuration* model. Each `Tolerance` element is translated into a signal `Filter` block in the *extended* model with a given tolerance value and connected to a given port. Ports are identified by the name of the containing component and the name of the port.

To proceed with the variant model generation, the wellformedness of models



(a) A boiler component

(b) An extended boiler component

Figure 6.2: Boiler components

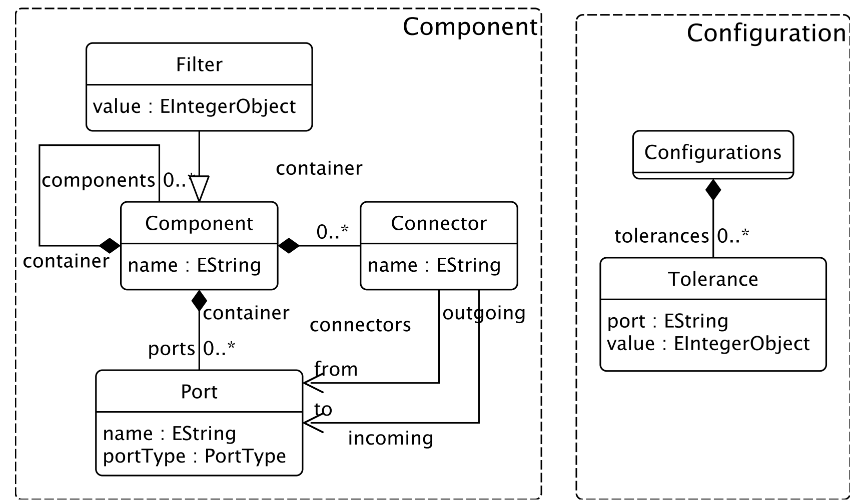


Figure 6.3: *configuration* and *component* metamodels

ⓐ and ⓑ is validated with task ① using EVL invariants[1] such as in Listing 6.1. The constraint `HasSource` in line 2 checks that all Connector elements in the *component* model have a source, by checking that their `from` property is defined. Similarly, the constraint `PositiveValue` in line 8 checks that the value of elements of type Tolerance in the *configuration* model are greater than zero.

```
1  context component!Connector {
2    constraint HasSource {
3      check : self.from.isDefined()
4      message : self.name + " has no source"
5    }
6  }
7  context config!Tolerance {
8    constraint PositiveValue {
9      check : self.value > 0
10     message : "Tolerance does not have positive value"
11   }
12 }
```

Listing 6.1: Sample EVL invariants

After the validation step, models ⓐ and ⓑ are consumed by an ETL model-to-model transformation ② to produce the *extended* model ⓒ. Each `Filter` component created from the *configuration* model is assigned with a tolerance value and placed at the appropriate connector's port. The Filter is created at the same level (container) as the component of the targeted port. The `TemperatureFilter` in Figure 6.2b and its new connections are the example added elements that result from this transformation.

The remaining operation is an EGX model-to-text transformation ③ which uses the *extended* model as input to generate Java code. The resulting code establishes the connections between components, but the developer is expected to handwrite their internal logic inside protected regions[2]. These regions are illustrated in Listing 6.2 where lines 4 and 6 indicate the start and end of a protected region which shall not be overwritten if the code generation is re-executed.

```
1  public class TemperatureComparator {
2    private Double temperature, targetTemperature,
         difference;
3    private void compute() {
4      /* protected region compute on begin */
5      this.difference = this.targetTemperature-this.
```

[1]The type of element that the constraints act upon and the model they belong to is indicated in the `context` environment e.g., `component!Connector` acts on Connector elements from the *component* model. The `message` in the constraint is displayed when the check fails.

[2]A section which should not be overwritten if the model-to-text transformation is re-executed.

```
        temperature;
6     /* protected region compute end */
7   }
8 }
```

Listing 6.2: Generated code of the *TemperatureComparator* component

6.1.2 Experimental setup

We have implemented and executed this workflow both in Gradle and ModelFlow under seven different scenarios. The first scenario represents a clean build, while all other scenarios represent realistic changes to resources (models, generated files) which affect subsequent executions. These scenarios are described in Sec. 6.1.3. Both build tools parse an equivalent build script that captures the workflow and uses the same model management tasks and resources. We measure the execution time of the different scenarios in both tools and also verify that each scenario executes affected tasks and generates required files. The correctness of the outputs is tested by running a Java program that uses the generated Java classes to run a boiler execution simulation which prints the values of some component ports in response to simulated input port values.

Gradle setup. We have extended Gradle to support the execution of the EVL, ETL and EGX tasks required by the workflow. In addition, we have also extended its DSL to support a custom data structure where models can be defined once. The Gradle workflow specification is presented in Listing 6.3. Lines 1-14 illustrate a custom data structure that we implemented to capture the models. Each model indicates its type in brackets, while configuration parameters are captured within curly braces. In this case study all models are EMF models and are persisted in files with an XMI format. The model management tasks of the workflow are declared in lines 15-33. Each task receives the names of its input and output models as parameters.

As a general-purpose build tool, Gradle does not support most of the desired MDE build tool features out of the box. Its conservative execution mechanism is based on inputs and *expected* outputs i.e., known before the task execution. In addition, dynamic resources such as models cannot influence the task execution order, there is no end-to-end traceability offered and outputs are not protected at any point.

Our Gradle task extensions for Epsilon have been implemented so that they resolve required input and output models from the model DSL extension and the model files are declared as dynamic inputs or outputs. We have some task parameters as inputs or outputs as we do in ModelFlow, however their hashes are computed with the default mechanism used by Gradle. Upon execution, our

task implementations iterate over required input and output models, loading all required models before execution and disposing all after the execution.

```
1  epsilon {
2    models {
3      config(EMF){
4        modelFile = file('resources/m/config.model')
5        metamodelFile =
             file('resources/mm/configuration.ecore')
6      }
7      component(EMF){
8        modelFile = file('resources/m/component.model')
9        metamodelFile =
             file('resources/mm/component.ecore')
10     }
11     extended(EMF){
12       modelFile = file('resources/m/extended.model')
13       metamodelFile =
             file('resources/mm/component.ecore')
14     }
15   }
16 }
17 task validate(type: EVL){
18   src = file('resources/mmt/validation.evl')
19   input = 'config'
20   input = 'component'
21 }
22 task m2m(type: ETL){
23   src = file('resources/mmt/extended.etl')
24   input = 'config'
25   input = 'component'
26   output = 'extended'
27   dependsOn validate
28 }
29 task m2t(type: EGX){
30   src = file('resources/mmt/generate.egx')
31   outputRoot = file('src-gen')
32   input = 'extended'
33 }
```

Listing 6.3: Gradle workflow

ModelFlow setup. We ran ModelFlow in non-interactive mode and configured it to discard any changes in the outputs of tasks. No model management traces were recorded. These actions were taken to make ModelFlow's execution similar to Gradle's except from how up-to-date checks for task parameters and

model resources.

```
1  param basedir;
2  model config is epsilon:emf {
3    src : basedir + "config.model"
4    metamodelFile : basedir + "configuration.ecore"
5  }
6  model component is epsilon:emf {
7    src : basedir + "component.model"
8    metamodelFile : basedir + "component.ecore"
9  }
10 model extended is epsilon:emf {
11   src : basedir + "extended.model"
12   metamodelFile : basedir + "component.ecore"
13 }
14 task validate is epsilon:evl
15     in config and component {
16   src : basedir + "validation.evl"
17 }
18 task m2m is epsilon:etl
19     in config and component
20     out extended
21     dependsOn validate {
22   src : basedir + "extended.etl"
23 }
24 task m2t is epsilon:egx
25     in extended {
26   src : basedir + "generate.egx"
27   outputRoot : "src-gen"
28 }
```

Listing 6.4: ModelFlow workflow

6.1.3 Results

Correctness. All scenarios in both tools were able to generate the required files to run the Java simulation.

We describe below the set of changes that the different scenarios involved, along with the observed behaviour of the tools.

1) *Clean execution:* This scenario represents a first-time execution where no caches are available. Both tools behaved as expected, that is, all tasks were executed.

2) *No changes:* After a clean execution, in this scenario we trigger a new one having made no changes to input or output resources. As such, we would not expect any task to be executed, which is the case for both tools in the experiment.

3) *Change in the source model:* In this scenario the *component* model file is modified after a clean execution by changing the name of a port in the *component* model. We expect everything to re-execute as *component* is an input model for the validation and model-to-model transformation tasks, and this property should be propagated to the extended model and into the generated code. In the experiment, this is the case in both tools.

4) *Change in intermediate output model:* In this scenario we modify the value of a filter element in the extended model after a clean execution. Using the non-protective execution mode of ModelFlow, we expect it to trigger the transformation to restore the consistency of this model and to skip the code generation. A similar behaviour is expected from Gradle. This is the observed behaviour on both.

5) *Template changes:* After a clean execution, this scenario consists of triggering an execution after modifying the template files required by the model-to-text transformation. As this is the only task affected, we expect both tools to only execute that task. This is the observed behaviour on both build tools.

6) *Non-protected changes in generated code:* In this scenario, we add a comment outside of the protected regions of a generated file. In contrast to the previous scenario, it is the task's outputs that are modified not its inputs. In this case ModelFlow only executed the model-to-text transformation, overwriting the not-allowed changes in the generated code, while Gradle skipped all tasks, leaving the changes in the generated code.

7) *Protected changes in generated code:* In this scenario, we add a print statement inside a protected region of a file from the generated code. We expect all tasks to be skipped as the output should be considered up to date for the code generating task. In this case, both build tools behave as expected.

Performance. We report on the execution times of the scenarios in which both tools reacted to changes in the same way. The time measures of their execution are shown in Figure 6.4. The value reported for the first scenario corresponds to the time of the first execution (clean), while all other scenarios report on the time of the second execution.

The workflow was configured to use a *component* model (24kB) that represents a system of controllers (such as the one displayed in Figure 6.2a), and a *configuration* model (686 bytes) that was used to create a filter for each controller. Each scenario was executed 20 times with 5 warm-up iterations. We used Gradle version 6.2.1 and invoked it with the Gradle Tooling API ensuring no cache files were available between iterations. The experiments were executed on an 8-Core Intel Core i9 CPU @ 2.3 GHz with 16 GB of RAM and the Java Virtual Machine was provided with up to 4GB of memory running with JDK 1.8.0_231.

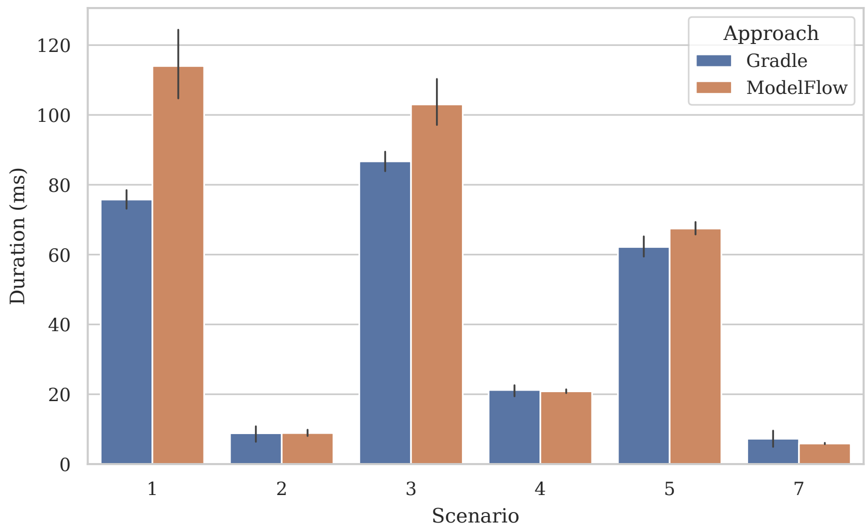


Figure 6.4: Execution time of each scenario in milliseconds.

6.1.4 Discussion

While most scenarios resulted in similar behaviour on both built tools, we now discuss those that did not.

In Scenario 4, ModelFlow can respond to changes in the *extended* model in two different ways: either to (a) use the modified model as source and trigger the code generation; or (b) discard the modifications in the model by triggering the transformation and skipping the code generation. However, this is not possible in Gradle which by default responds with the second approach which discards the changes invoking the transformation. Moreover, the reason why the model-to-model task is executed in Gradle is because the model file (declared as output) is known before the task execution.

In Scenario 6, ModelFlow executed the model-to-text transformation which was able to restore the build consistency while Gradle skipped the output analysis for the generated files which are known after the execution. Gradle did not trigger this execution because dynamic outputs are not used to determine whether a re-execution is needed. A workaround for this would involve declaring the output folder as an input directory so that changes in the generated files are used to determine whether a re-execution is needed.

In Scenario 7 both tools behave as expected but for different reasons. ModelFlow does not execute because it determines that the outputs have not been modified from previous executions, while Gradle simply skips the output analysis for the same reasons as in Scenario 6.

Regarding performance, the computation of changes to input resources was slower in ModelFlow, particularly in the first-time execution. However, subsequent executions were nearly identical to Gradle's. The computation

of output resources is more exhaustive in ModelFlow which may account for some of the overhead. While there was no mechanism to reuse loaded models in Gradle, the size of models used does not incur on a significant reloading overhead. It is unlikely that the use of larger models could highlight the impact of the mode reuse approach has on the workflow performance. This is because EMF models start presenting scalability issues when the models are very large i.e., in the order of millions of elements. In contrast, using models with a technology like Simulink could show the impact of the model reuse approach as the loading stage is much more expensive in time than EMF, even for small models.

6.1.5 Threats to validity

One clear threat to validity are the implementation differences between the two build engines, inherent to their own architectures. We have minimised this threat by implementing the code of the invoked tasks as equivalent as possible so that the results of the evaluation reflect the impact of the architectural decisions and not of the individual tasks. Furthermore, we opted for a custom data structure to declare the models, so that tasks can receive the models in a similar fashion as tasks in ModelFlow do, that is, declared globally so that tasks can indicate which are used as input or output but, in Gradle's case, requiring reloading per task execution. The purpose of the performance evaluation was to give a time context to the qualitative evaluation but was not intended to measure the scalability of the approaches as the size of the models used in the experiment are small. That said, we have removed from the performance evaluation those scenarios in which the behaviour of the two tools was different.

6.2 Case study: EuGENia

EuGENia is a tool that predates this research and consists of a workflow with multiple tasks and intermediate models used to generate graphical model editors. This case study explores the use of ModelFlow to reproduce this workflow and compares it with the original implementation of EuGENia in terms of performance and correctness under realistic change scenarios. In contrast, the previous case study evaluated the correctness and performance of ModelFlow across change scenarios affecting different types of input and output artefacts from the workflow (e.g., in/out/inout models, output files, source programs, protected/unprotected regions) and against another build tool.

Regarding the correctness evaluation, this case study reviews if the workflow responds to change scenarios by executing the appropriate tasks, whether the

models are loaded and disposed appropriately, and verifies that the workflow was able to recover traces from different types of tasks. Additionally, the generated code was manually inspected to check for differences with the output of the original implementation and, to ensure its correct behaviour, we verified that a BPMN editor could be launched and used. Regarding the performance evaluation, this case study measures the overhead of features like tracing and up-to-date checks on inputs and outputs, and compares the overall execution time with the original workflow implementation.

We introduce EuGENia in Sec. 6.2.1 then discuss how we re-implemented it with ModelFlow in Sec. 6.2.2 and then evaluate its correctness and performance in Sec. 6.2.3-6.2.5.

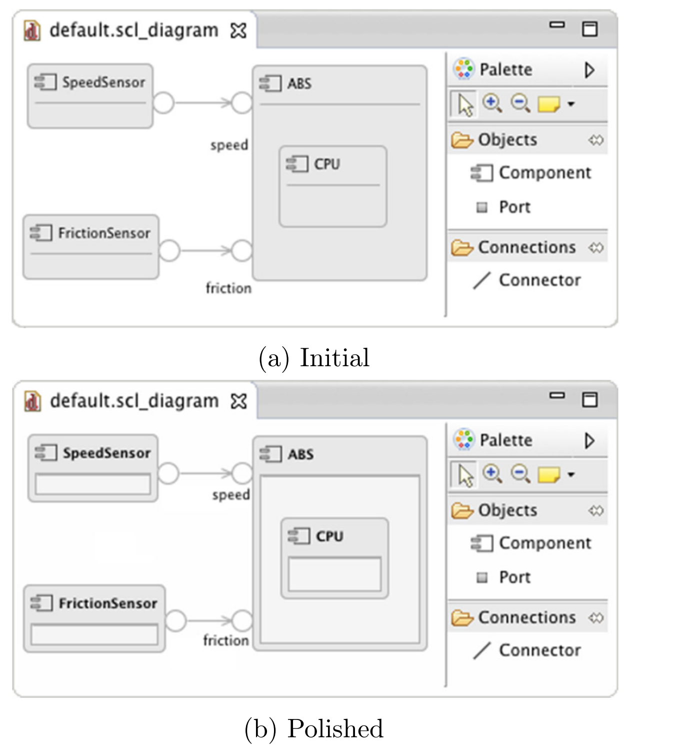
6.2.1 Background

EuGENia is an existing open-source tool that uses metamodel annotations and model transformations to streamline the process of generating graphical model editors based on EMF and GMF [106]. For example, Listing 6.5 shows a metamodel defined in Emfatic (a textual notation for Ecore) which describes a Simple Component-connector Language (SCL) [106]. This metamodel has been extended with `@emf` and `@gmf` annotations placed on top of the various metamodel constructs to specify aspects of the EMF and GMF generation processes. The result of EuGENia's execution on this metamodel is the graphical editor in Figure 6.5a.

```
1  @namespace(uri="scl", prefix="scl")
2  @emf.gen(basePackage="eugenia.examples")
3  package scl;
4  @gmf.diagram @gmf.node(label="name", color="2,3,2")
5  class Component {
6    attr String name;
7    @emf.gen(propertyMultiline="true")
8    attr String description;
9  }
10 @gmf.compartment(layout="free")
11 val Component[*] subcomponents;
12   @gmf.affixed
13   val Port[*] ports;
14 }
15 @gmf.link(source="from",target="to",label="name",target
      .decoration="arrow")
16 class Connector {
17   attr String name;
18   ref Port#outgoing from;
19   ref Port#incoming to;
20 }
```

```
21 @gmf.node(figure="ellipse",size="15,15",label.icon="
     false",label.placement="external",label="name")
22 class Port {
23   attr String name;
24   val Connector#from outgoing;
25   ref Connector#to incoming;
26 }
```

Listing 6.5: Annotated Emfatic metamodel of an SCL



(a) Initial

(b) Polished

Figure 6.5: SCL editor generated with EuGENia (Image from [106]).

To enable the generation of a graphical editor from a metamodel, EuGENia extends and integrates the built-in EMF and GMF code generation processes described below.

The EMF code generation process starts from the definition of the domain metamodel (abstract syntax) in `Ecore` or an `Emfatic` [183] file. Then built-in EMF model-to-model transformations are used to produce the EMF generator model (`GenModel`) from the metamodel. The `GenModel` captures Java implementation details and can be further customised. Finally, an EMF built-in model-to-text transformation consumes the `GenModel` and produces the Java code and required configuration files.

The Graphical Modeling Framework (GMF) provides a model-driven approach to the generation of Eclipse-based graphical editors for EMF-based DSLs. Its code generation process builds on the EMF code generation process. The first stage of this process involves the manual construction of models that specify

different aspects of the graphical syntax of the language. These models include the graph model (GmfGraph) which specifies the shapes, connections, labels, decorations, etc.; the tooling model (GmfTool) which specifies element creation tools; and the mapping model (GmfMap) which weaves the graphical elements in the GmfGraph model with the creation tools of the GmfTool model and the abstract syntax elements of the Ecore metamodel. The second stage of the process involves the production of a generator model (GmfGen) from the mapping model. The generator model contains the implementation details required by the graphical editor code generator and is produced from a model-to-model transformation. In the last stage, code is generated from the generator model.

The dependency graph in Figure 6.6 shows all the steps (groups of tasks of same colour) executed by EuGENia and the model resources they consume and produce.

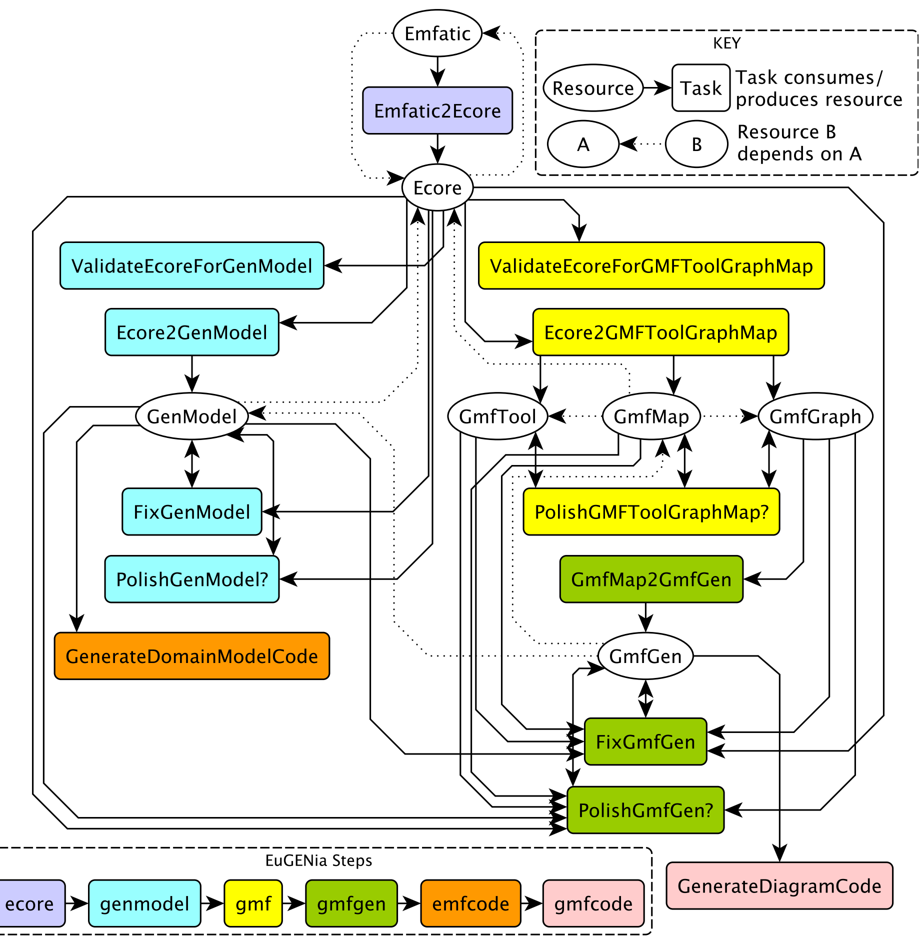


Figure 6.6: EuGENia task-resource and inter-task dependencies. The tasks in each step are sequentially ordered top-to-bottom. The solid arrows show the input and output models of the different tasks. Task names ending in a question mark denote optional tasks.

Without EuGENia, simple metamodel changes can be propagated to the

corresponding `GenModel` by an EMF built-in reconciler without overwriting any user-defined customisations. However, for more complex changes the `GenModel` would need to be regenerated and customised from scratch [106]. EuGENia provides a set of built-in metamodel annotations to attach implementation semantics which can be used to customise the `GenModel` after it is generated. Some of these (starting with `@emf`) are illustrated in Listing 6.5.

Similarly, while GMF provides built-in wizards for generating the `GmfTool`, `GmfGraph`, and `GmfMapping` models from the metamodel, the resulting models are very simple [106]. Consequently, these models need to be manually created and updated after any metamodel changes [106]. With EuGENia, another set of built-in metamodel annotations can be used to attach graphical semantics to metamodel elements and enable the automated derivation of these models from the metamodel. Some of these (starting with `@gmf`) are illustrated in Listing 6.5.

Overall, after model-to-model transformation tasks, EuGENia triggers built-in fixes derived from the used annotations but also allows the execution of polishing transformations, that is, user-defined in-place model transformations that can be used to fine-tune the models produced by predefined model-to-model transformations and model fixes. After fixing and polishing, the initial SCL editor from the previous example could look like Figure 6.5b. In addition, EuGENia provides and executes built-in metamodel validations that, upon failure, halt the execution of subsequent steps of the EMF and GMF processes.

EuGENia steps can be launched from Eclipse. The EuGENia/Eclipse UI has manually extended context-menus of the different workflow models to trigger "build targets" that use them as starting point. For example, the context-menu for an Emfatic file offers 3 build targets: to generate the GMF Editor (which is the complete workflow), to generate the Model code and to generate the Ecore model.

Alternatively, EuGENia can also be executed as an Ant task that can run the entire workflow or only a part of it. To execute a subset of its workflow the Ant task must specify the starting and/or ending step. Below is an example of a EuGENia workflow that will only execute the `gmf` and `gmfgen` steps. As an Ant script, EuGENia can be integrated into other model management workflows or be invoked automatically when the metamodel is changed.

```
<epsilon.eugenia src="my.ecore"
  firstStep="gmf" lastStep="gmfgen"/>
```

6.2.2 Approach

In this section we discuss how the EuGENia workflow has been re-implemented in ModelFlow. We have fragmented the complete workflow definition into

several listings for readability. We use the EuGENia steps described in the previous section to decompose and describe the workflow in groups of tasks that serve towards a common goal. The resolved dependency graph is shown in Figure 6.7. Overall, the ModelFlow workflow specification of EuGENia declares 6 models and 14 tasks out of which, 12 are model management tasks and 2 are helper tasks.

The following listings share parameters and variables across different task and model declarations. The parameters defined in lines 1-3 of Listing 6.6 are provided at runtime by the user. The `metamodelName` is the name of the source metamodel file (without extension) that will be used to drive the graphical editor components. All intermediate models that will be produced will share this name but differ in their file extension and the metamodel they conform to. The `pluginPrefix` is used to determine the prefixing name of the Java projects that will be generated with the code that will launch the graphical editor. Finally, the `copyrightStatement` is an optional parameter that points to a file from which copyright information shall be extracted.

The variables `base` and `pluginName` are defined in the `pre` block in lines 4-7 of Listing 6.6 and compute useful information based on the `metamodelName` and `pluginPrefix` parameters. In particular, the `base` refers to the directory in which the models are or will be persisted and the `pluginName` computes the base name for the Java projects that will be generated.

```
1 param metamodelName;
2 param pluginPrefix;
3 param copyrightStatement;
4 pre {
5   var base = "resources/model/" + metamodelName;
6   var pluginName = pluginPrefix + "." + metamodelName;
7 }
```

Listing 6.6: ModelFlow EuGENia workflow parameters and variables

The model declarations are shown in Listing 6.7 and these include `ECore`, `GenModel`, `GmfTool`, `GmfMap`, `GmfGraph` and `GmfGen`. All these models are of type `epsilon:emf` and share the same name differing in the file extension. Similarly, all these models conform to a different metamodel described by the parameter `metamodelUri`.

```
1 model ECore is epsilon:emf {
2   src : base + ".ecore"
3   metamodelUri : "http://www.eclipse.org/emf/2002/Ecore
      "
4 }
5 model GenModel is epsilon:emf {
6   src : base + ".genmodel"
```

```
7     metamodelUri : "http://www.eclipse.org/emf/2002/
          GenModel"
8   }
9   model GmfGen is epsilon:emf {
10    src : base + ".gmfgen"
11    metamodelUri : "http://www.eclipse.org/gmf/2009/
          GenModel"
12    expand : true
13    saveOpts {
14      var map = new Map;
15      map.put("ENCODING", "UTF-8");
16      map.put("SAVE_ONLY_IF_CHANGED","MEMORY_BUFFER");
17      map.put("SCHEMA_LOCATION",true);
18      map.put("LINE_WIDTH",1);
19      return map;
20    }
21  }
22  model GmfMap is epsilon:emf {
23    src : base + ".gmfmap"
24    metamodelUri : "http://www.eclipse.org/gmf/2008/
          mappings"
25    expand : true
26  }
27  model GmfTool is epsilon:emf {
28    src : base + ".gmftool"
29    metamodelUri : "http://www.eclipse.org/gmf/2005/
          ToolDefinition"
30  }
31  model GmfGraph is epsilon:emf {
32    src : base + ".gmfgraph"
33    metamodelUri : "http://www.eclipse.org/gmf/2006/
          GraphicalDefinition"
34  }
```

Listing 6.7: Models used in the ModelFlow EuGENia workflow

The Ecore step shown in Listing 6.8 has one task that transforms an Emfatic file into the `ECore` model. This task is captured by the `Emfatic2Ecore` task which is an EMF task of type `emf:emfatic2ecore`. The Emfatic file is used as the source of the task which parses the file and generates the `ECore` as output.

```
1   task Emfatic2Ecore is emf:emfatic2ecore
2       out ECore {
3     src : base + ".emf"
4   }
```

Listing 6.8: Ecore step tasks in the ModelFlow EuGENia workflow

In EuGENia the `genmodel` step consists of four tasks: `ValidateEcoreForGenModel`, `Ecore2GenModel`, `FixGenModel`, and `PolishGenModel`. Listing 6.9 shows how these tasks are captured in ModelFlow. The `ValidateEcoreForGenModel` task (lines 1-4) is an EVL task that validates that the `ECore` is well-formed and that it can be used to produce the `GenModel` model. The `Ecore2GenModel` task (lines 5-22) is an ETL task that takes as input the `ECore` model and produces the `GenModel`. This task also uses additional parameters, some that are passed by the user at runtime (e.g., `pluginName`) while others (e.g. `copyright` and `genPackages`) are the result of the execution of other tasks in the workflow. Line 14 shows how the map entry `usedGenPackages` of the `params` task definition property receives the result of the EOL task `genPackages` (lines 53-57) while lines 15-19 illustrate how the map entry `copyright` uses the file contents retrieved by the file reader task `copyright` (lines 58-58). The explicit use of other task results in the `Ecore2GennModel` task creates an implicit dependency between these tasks. Also, since the results of the `genPackages` task are only accessible in memory, it is annotated with `@always` to ensure that, regardless of its inputs, it always executes. Tasks of type `core:fileReader`, such as the `copyright` task, do not need an annotation as they are always executed.

The tasks `FixGenModel` (lines 23-36) and `PolishGenModel` (lines 37-52) are both of type EOL. They both receive as input the `ECore` model and modify the `GenModel`. In contrast to `FixGenModel`, `PolishGenModel` is an optional task that is executed when the polishing script is available as indicated by its guard (line 41). Because these two tasks use the same models in the same way (consume one and modify the other), to ensure that `PolishGenModel` is executed after `FixGenModel` (if the guard is valid), then a dependency must be created between these two tasks.

```
1  @always
2  task ValidateEcoreForGenModel is epsilon:evl
3    in ECore as Ecore {
4    src : "resources/task/Ecore2GenModel.evl"
5  }
6  task Ecore2GenModel is epsilon:etl
7      in ECore as Ecore
8      out GenModel
9      dependsOn ValidateEcoreForGenModel {
10   src : "resources/task/Ecore2GenModel.etl"
11   params {
12     var map = new Map;
13     map.put("pluginName", pluginName);
14     map.put("foreignModel", "Ecore2GenModel");
15     map.put("usedGenPackages", genPackages.result);
```

```
16      if (copyrightStatement.isDefined()){
17        map.put("copyright", copyright.contents);
18      } else {
19        map.put("copyright", "");
20      }
21      return map;
22    }
23  }
24  task FixGenModel is epsilon:eol
25      in ECore as Ecore
26      inout GenModel {
27    src : "resources/task/FixGenModel.eol"
28    params {
29      var map = new Map;
30      if (copyrightStatement.isDefined()){
31        map.put("copyright", copyright.contents);
32      } else {
33        map.put("copyright", "");
34      }
35      return map;
36    }
37  }
38  task PolishGenModel is epsilon:eol
39      in ECore
40      inout GenModel
41      dependsOn FixGenModel {
42    guard : self.src.exists()
43    src : "resources/task/polish/FixGenModel.eol"
44    params {
45      var map = new Map;
46      if (copyrightStatement.isDefined()){
47        map.put("copyright", copyright.contents);
48      } else {
49        map.put("copyright", "");
50      }
51      return map;
52    }
53  }
54  @always
55  task genPackages is epsilon:eol
56      in ECore {
57    src : "resources/task/genPackages.eol"
58  }
59  task copyright is core:fileReader
60      in ECore as Ecore {
61    guard: copyrightStatement.isDefined()
```

```
62    src {
63      if (copyrightStatement.isDefined()){
64        return copyrightStatement;
65      } else {
66        return "";
67      }
68     }
69  }
```

Listing 6.9: GenModel step tasks in the ModelFlow EuGENia workflow

The GMF step shown in Listing 6.10 consists of 3 tasks `ValidateEcoreForGMFToolGraphMap`, `Ecore2GMFToolGraphMap` and `PolishGMFToolGraphMap`. The `ValidateEcoreForGMFToolGraphMap` task (lines 1-4) is another EVL validation against the Ecore which checks for well-formedness on the graphical side. Then `Ecore2GMFToolGraphMap` (lines 5-10) uses as input the `ECore` to generate three models (`GmfMap`, `GmfGraph` and `GmfTool`) through an EOL program. Then `PolishGMFToolGraphMap` (lines 11-16) may execute if the user provides a polishing script that modifies the three generated models. Since `Ecore2GMFToolGraphMap` produces the three models (rather than modify them), it is executed before the polishing task.

```
 1  @always
 2  task ValidateEcoreForGMFToolGraphMap is epsilon:evl
 3      in ECore as Ecore {
 4    src : "resources/task/ECore2GMF.evl"
 5  }
 6  task Ecore2GMFToolGraphMap is epsilon:eol
 7      in ECore as Ecore
 8      out GmfMap and GmfGraph and GmfTool
 9      dependsOn ValidateEcoreForGMFToolGraphMap {
10    src : "resources/task/ECore2GMF.eol"
11  }
12  task PolishGMFToolGraphMap is epsilon:eol
13      in ECore
14      inout GmfMap and GmfGraph and GmfTool {
15    guard : self.src.exists()
16    src : "resources/task/polish/ECore2GMF.eol"
17  }
```

Listing 6.10: GMF step in ModelFlow EuGENia workflow

The GmfGen step shown in Listing 6.11 consists of three tasks: `GmfMap2GmfGen`, `FixGmfGen` and `PolishGmfGen`. The `GmfMap2GmfGen` task is a GMF task that takes as input the models `ECore`, `GmfMap` and `GenModel` to produce a `GmfGen` model. Then the `FixGmfGen` task executes an EOL program intended to modify the `GmfGen` model. Once more, the `PolishGmfGen` task is another EOL program

that depends on the previous task and only executes if the source file is provided by the user.

```
1  task GmfMap2GmfGen is gmf:gmfMap2gmfGen
2    in ECore as Ecore and GmfMap and GenModel
3    out GmfGen;
4  task FixGmfGen is epsilon:eol
5      in ECore as Ecore and GenModel and GmfMap and
          GmfGraph and GmfTool
6      inout GmfGen {
7    guard : self.src.exists()
8    src : "resources/task/FixGMFGen.eol"
9    params {
10     var map = new Map;
11     if (copyrightStatement.isDefined()){
12       map.put("copyright", copyright.contents);
13     } else {
14       map.put("copyright", "");
15     }
16     return map;
17   }
18 }
19 task PolishGmfGen is epsilon:eol
20     in ECore as Ecore and GenModel and GmfMap and
          GmfGraph and GmfTool
21     inout GmfGen
22     dependsOn FixGmfGen {
23   guard : self.src.exists()
24   src : "resources/task/polish/FixGMFGen.eol"
25 }
```

Listing 6.11: GmfGen step in ModelFlow EuGENia workflow

The last two steps of the EuGENia workflow are the code generation steps: one for the domain code (`GenerateDomainModelCode`), another for the graphical editor (`GenerateDiagramCode`). The domain code generator task (lines 1-5) is provided by EMF and uses the `GenModel` as input, and it receives additional parameters to configure the generation. The graphical editor generator task (lines 5-7) is provided by GMF and uses the `GmfGen` model as input.

```
1  task GenerateDomainModelCode is emf:genCode
2      in GenModel {
3    generateEdit : true
4    generateEditor : true
5    generateTests: true
6  }
7  task GenerateDiagramCode is gmf:genDiagram
```

```
8     in GmfGen;
```

Listing 6.12: EmfCode step in ModelFlow EuGENia workflow

The resolved dependency graph is illustrated in Figure 6.7 while the computed execution graph is shown in Figure 6.8.

6.2.3 Setup

To evaluate ModelFlow's adaptation of EuGENia we executed it under different change scenarios that are common in EuGENia workflows. We evaluate the correctness of the adaptation by analysing the tasks that are executed on the different mutation scenarios. Similarly, we measure the time it took for the workflow to execute in ModelFlow and compare it to its execution in the original implementation. Furthermore, we profiled the total and partial execution with ModelFlow to estimate the overhead of features such as conservative executions and model management tracing.

Both approaches (ModelFlow and the original implementation) were measured executing a workflow that used the same Emfatic file as input[3] which was originally presented in [106] and describes a simple BPMN process. Furthermore, we reused the two polishing scripts and the custom plugin provided in [106] to customise the graphical editor generation in both approaches. An example of a simple BPMN model in the generated graphical editor (using ModelFlow) is shown in Figure 6.9.

Scenarios

We have executed the EuGENia workflow in six different scenarios for both approaches. The first scenario represents the first-time execution, while all other scenarios represent changes to resources which affect subsequent executions. We describe below the set of changes that the different scenarios involved, along with the observed behaviour of the tools.

i) *First-time execution:* This scenario represents a first-time execution.

ii) *No changes:* This scenario represents a second execution of the workflow having made no changes to any artefacts.

iii) *GMF annotation change in Emfatic:* The second execution happens after the annotation `border.style` of the `@gmf.node` annotation of the `Group` class is changed from `dash` to `dot`. This change affects the graphical representation of the metamodel constructs.

iv) *Rename of EMF class in Emfatic:* This scenario represents a second workflow execution after the class `Activity` in the Emfatic is renamed to `Task`. This change affects the metamodel and its graphical representation.

[3] https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/examples/org.eclipse.epsilon.eugenia.bpmn

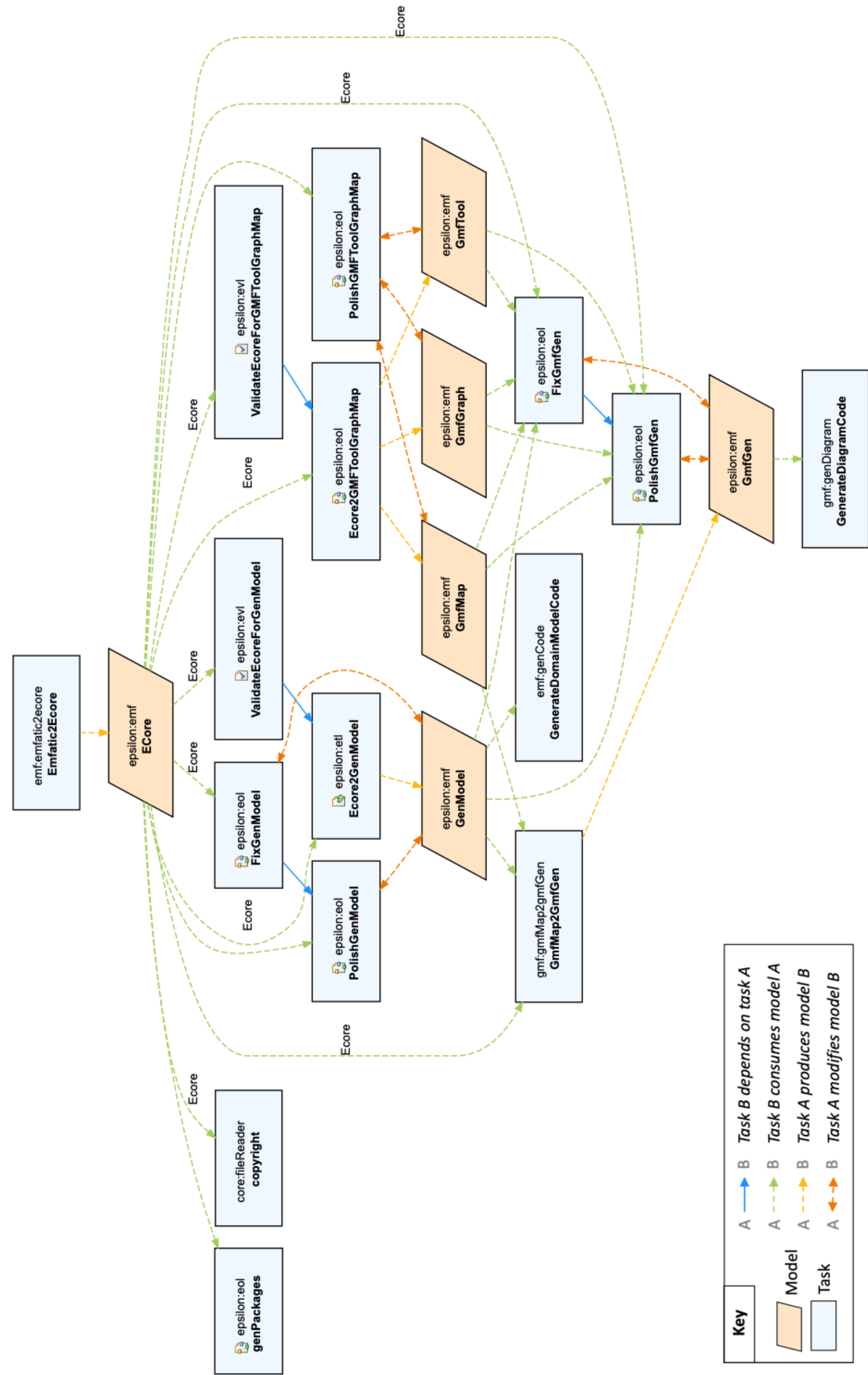


Figure 6.7: EuGENia ModelFlow dependency graph.

v) *GenModel annotation in Emfatic:* This scenario represents an execution after adding the `@emf.gen` annotation to the top of the Emfatic file, to specify

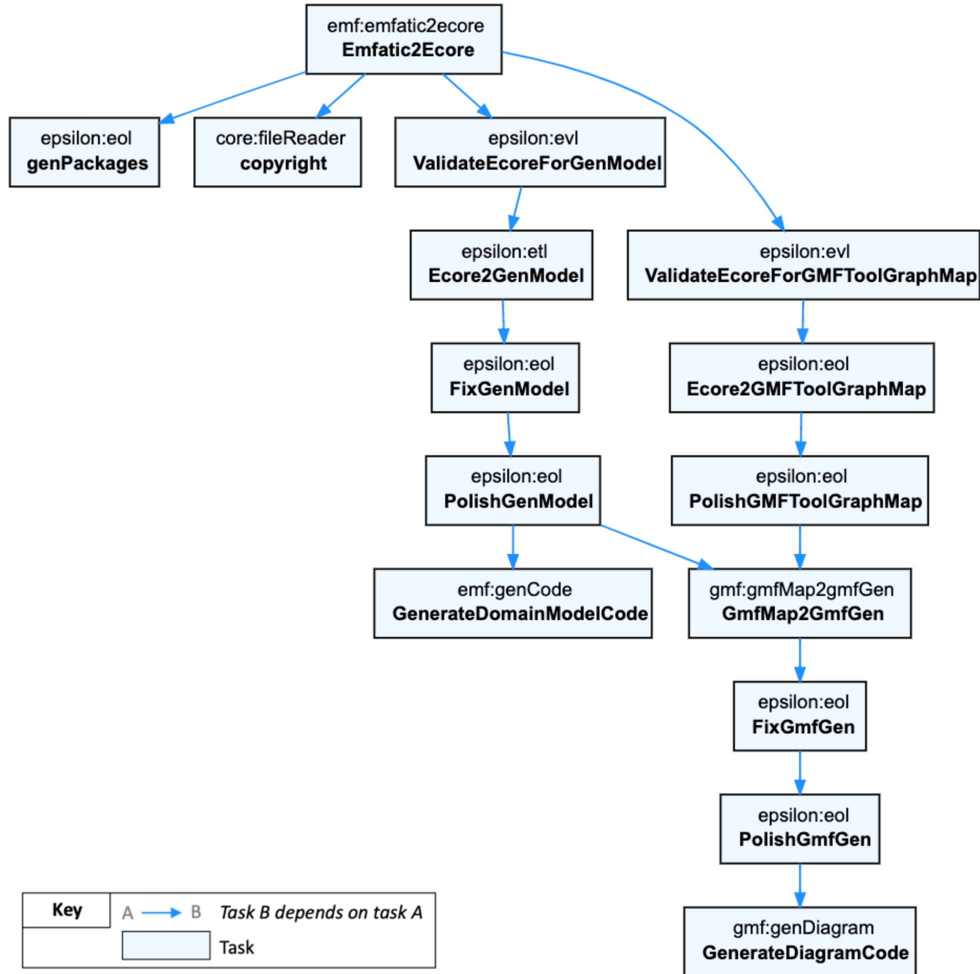


Figure 6.8: EuGENia ModelFlow execution graph.

a `basePackage`. This change affects how the metamodel code is generated and implicitly affects the graphical side as well.

vi) *Modify GMF polishing:* In this scenario the script `Ecore2GMF.eol` (invoked by the task `PolishGMFToolGraphMap`) is updated to change a label's font style and the border colour of a figure. The changes are shown in Listing 6.13 and have been slightly adapted from the polishing transformation in Listing 10 of [106] to be used with the BPMN metamodel used in the workflow.

```
1 // Add bold font to component label
2 var activityLabel = GmfGraph!Label.all.selectOne(l|l.
    name="ActivityLabelFigure");
3 activityLabel.font = new GmfGraph!BasicFont;
4 activityLabel.font.style = GmfGraph!FontStyle#BOLD;
5
6 //Set background color and border of the component
    compartment
7 var activityFigure = GmfGraph!RoundedRectangle.all.
```

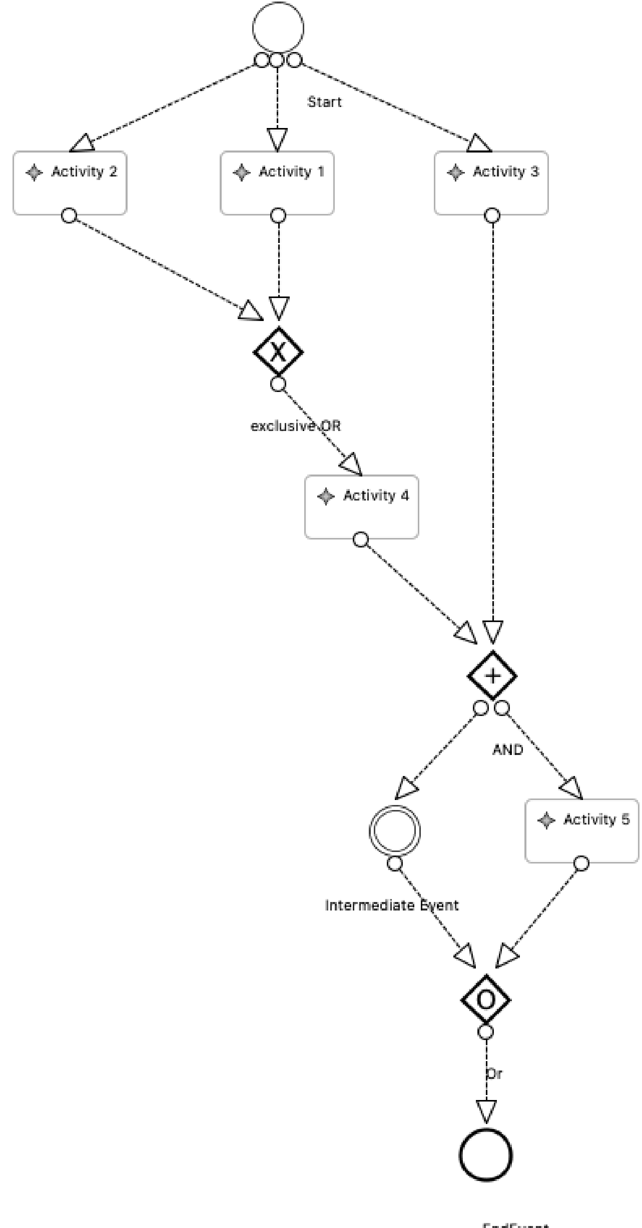


Figure 6.9: BPMN model created using the graphical editor generated with ModelFlow.

```
      selectOne(r|r.name="ActivityFigure");
 8 var lineBorder = new GmfGraph!LineBorder;
 9 lineBorder.width = 1;
10 activityFigure.backgroundColor = createColor
      (245,245,245);
11 activityFigure.border = lineBorder;
12
13 operation createColor(red : Integer, green : Integer,
      blue : Integer) : GmfGraph!RGBColor {
14   var color = new GmfGraph!RGBColor;
```

```
15    color.red = red;
16    color.blue = blue;
17    color.green = green;
18    return color;
19  }
```

Listing 6.13: Extension fragment to polish script

Additional tasks

While the EuGENia workflow relies on several Epsilon tasks that were initially integrated to ModelFlow, it also required additional tasks to invoke EMF and GMF tasks that are part of the EuGENIA workflow. More specifically, the additional tasks that were integrated to ModelFlow to support this workflow were:

gmf:gmfgen: Generates a graphical generator model (`GmfGen`) from a set of GMF models (`GmfGraph`, `GmfMap`, `GmfTool`).

gmf:gencode: Generates the graphical editor code from a graphical generator model (`GmfGen`).

emf:emfatic2ecore: Generates an Ecore file from an Emfatic file.

emf:gencode: Generates the metamodel code from a generator model (`GenModel`).

Differences from the original EuGENia workflow

The original EuGENia workflow works by sequentially executing a series of delegates (or tasks). The delegates that were executed in our experiments to represent the original workflow are shown in Listing 6.14. Some of those delegates that perform fixes also invoke the polishing tasks if their scripts are provided.

```
1  return Arrays.asList(
2    new Emfatic2EcoreDelegate(),
3    new GenModelEcoreValidationDelegate()
4      .setClearConsole(false),
5    new ToolGraphMapEcoreValidationDelegate()
6      .setClearConsole(false),
7    new Ecore2GenModelDelegate()
8      .setClearConsole(false),
9    new FixGenModelDelegate()
10     .setClearConsole(false),
11   new GenerateToolGraphMapDelegate()
12     .setClearConsole(false),
13   new GmfMap2GmfGenDelegate()
```

```
14       .setClearConsole(false),
15     new FixGmfGenDelegate()
16       .setClearConsole(false),
17     new GenerateEmfCodeDelegate(),
18     new GenerateDiagramCodeDelegate()
19       .setTargetPart(targetPart)
20   );
```

Listing 6.14: Original EuGENia delegate execution order.

All Epsilon programs used in EuGENia (e.g., `ECore2GMF.etl`) remain the same except for EOL tasks as they had to be extended to produce traceability information. In those cases, we have modified the program to use tracing facilities provided by the ModelFlow EOL task definition.

Regarding Epsilon task definitions such as EOL or ETL, their base execution remains the same, but their invocation had to be adapted to ModelFlow to determine whether input and outputs have changed and to collect traces. Regarding EMF and GMF task definitions, these underwent an invasive adaptation of their execution to recover generated files and have the ability to retrieve traceability information which is not readily available in their original implementation. In this workflow, this traceability information could be used to identify all generated files that come from a particular metamodel attribute or class, or to explore the variations in generated code or intermediate models based on the use of different annotations in the metamodel.

The execution parameters used in ModelFlow differ slightly from the ones used in the original implementation. In ModelFlow we use the `metamodelName`, `pluginPrefix` and `copyright` location as parameters, whereas the original workflow does not ask for `pluginPrefix` and assumes a default copyright location. Other conventions, such as the polish script location and naming remain the same in both implementations. These slight modifications are unlikely to have a significant impact in performance.

6.2.4 Correctness results

To determine the correctness of the workflow we examined the expected execution process (tasks skipped based on change scenarios), the number of times that models were loaded and disposed and the generated management traces. Additionally, the generated code was manually inspected to check for differences with the output of the original implementation and to ensure its correct behaviour we verified that a BPMN editor could be launched and used.

Executed tasks per scenario

Table 6.1 presents which tasks were executed by ModelFlow under the different change scenarios. Compared to the original implementation, EuGENia executes all the tasks regardless of the changes. In the table we can notice that the task `Emfatic2Ecore` is correctly skipped where the Emfatic file is not modified, that is, in scenarios *ii* and *vi* as the former is the scenario with no changes, and the latter the scenario in which the polishing script for the task `PolishGMFTool-GraphMap` is modified. The following four tasks in the table were supposed to be executed as per the `@always` annotation.

Table 6.1: Executed tasks per scenario

Task	i	ii	iii	iv	v	vi
Emfatic2Ecore	✔	x	✔	✔	✔	x
genPackages	✔	✔	✔	✔	✔	✔
copyright	✔	✔	✔	✔	✔	✔
ValidateEcoreForGenModel	✔	✔	✔	✔	✔	✔
ValidateEcoreForGMFToolGraphMap	✔	✔	✔	✔	✔	✔
Ecore2GenModel	✔	x	✔	✔	✔	x
Ecore2GMFToolGraphMap	✔	x	✔	✔	✔	x
FixGenModel	✔	x	✔	✔	✔	x
PolishGMFToolGraphMap	✔	x	✔	✔	✔	✔
PolishGenModel	x	x	x	x	x	x
GmfMap2GmfGen	✔	x	✔	✔	✔	✔
GenerateDomainModelCode	✔	x	✔	✔	✔	x
FixGmfGen	✔	x	✔	✔	✔	✔
PolishGmfGen	✔	x	✔	✔	✔	✔
GenerateDiagramCode	✔	x	✔	✔	✔	✔

The tasks `Ecore2GenModel` and `Ecore2GMFToolGraphMap` are executed when the `Emfatic` model is modified (even with annotations) as it produces a modified `Ecore` metamodel. The `GenModel` produced by the `Ecore2GenModel` task changes in *Scenarios i, iv,v.* As such, the `FixGenModel` task is expected to be executed in these scenarios. The reason why this task is also executed in *scenario iii* is that the `GenModel` produced by the `Ecore2GenModel` task is the same as in the previous execution but not the same as its latest version at the end of the previous workflow execution (after the task `PolishGenModel`). In particular, the latest version has additional information including copyright material which is extracted in the `copyright` task.

The `PolishGMFToolGraphMap` task is executed in *scenarios i* and *vi* because a polishing script is provided or modified, respectively. However, since the `GmfMap` model depends on the `Ecore` (see Figure 6.10) to compute its stamp (as indicated by the *expand* flag in the model declaration), changes to its dependencies or to itself trigger the execution of the task `PolishGMFToolGraphMap` where `GmfMap`

is used as input. As such, the task is also executed in *scenarios iii-v*.

The `GmfMap2GmfGen` task is executed whenever the `GmfMap` model or its dependencies change, that is, in *scenarios i, iii-vi*. Task `GenerateDomainModelCode` is executed when the `Emfatic` is modified, which excludes *scenarios ii* and *iv*. The remaining tasks (`FixGmfGen`, `PolishGmfGen` and `GenerateDiagramCode`) are always executed as they depend on `GmfGen` which also computes its stamp based on dependencies.

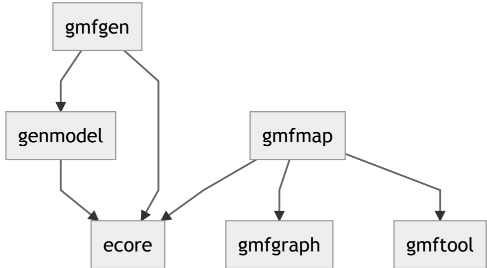


Figure 6.10: EuGENia model inter-dependencies. Arrows denote a "requires" relationship e.g., *genmodel* requires *ecore*

Model loading/disposal

In all scenarios except *Scenario ii - No modification*, ModelFlow correctly loaded and disposed each model exactly once. In *Scenario ii* only the `Ecore` model was correctly loaded and disposed once during the execution.

End-to-end traceability

Out of the 14 tasks executed by ModelFlow, 12 should produce traceability information because of how they interact with models. The EuGENia workflow was able to recover traces from tasks that provide them by default e.g., EVL and ETL, create them when they do not e.g., EOL and GMF/EMF tasks.

Figure 6.11 illustrates a sample of the recovered traces from the execution of the `Ecore2GenModel` ETL task. The figure shows the model element identifiers (hexagons) from the `Ecore` model (green) that produced model elements in the `GenModel` (brown). These models were produced in the context of the ETL rule with name `EStructuralFeature2GenFeature`.

As an example of created traces when tasks definition types do not provide them, Figure 6.12 shows the traces from the `Ecore2GMFToolGraphMap` EOL task which passed a trace utility to the EOL program at runtime.

For the `Emfatic2Ecore` task, which is an EMF task, we decided not to produce traceability as they both represent the same information but in a different notation. In the ModelFlow implementation of tasks `GmfMap2GmfGen`, `GenerateDomainModelCode` and `GenerateDiagramCode` we have modified the

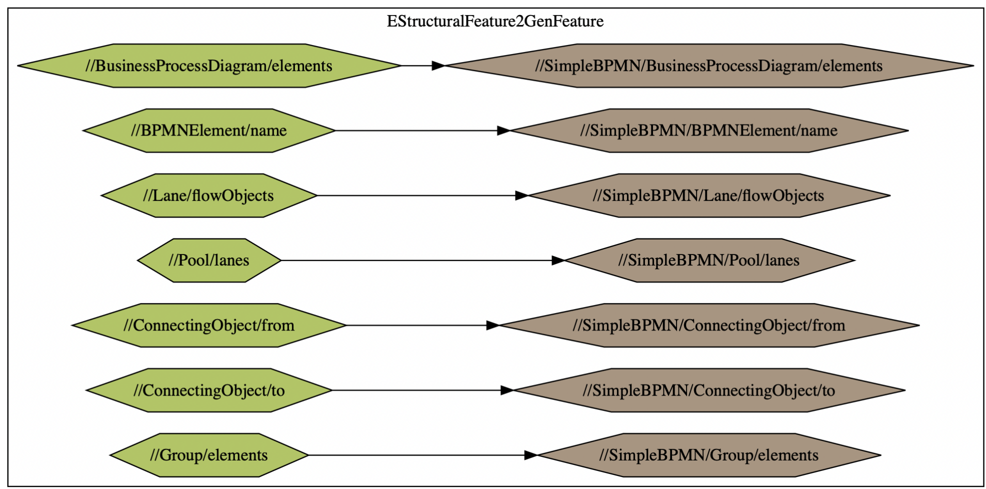


Figure 6.11: Trace view of the `EstructualFeature2GenFeature` rule of the ETL program executed by the `Ecore2GenModel` task. The view shows the model elements (hexagons on the right) that were produced by the ETL rule on the model `GenModel` from model elements (hexagons on the left) of the model `ECore`.

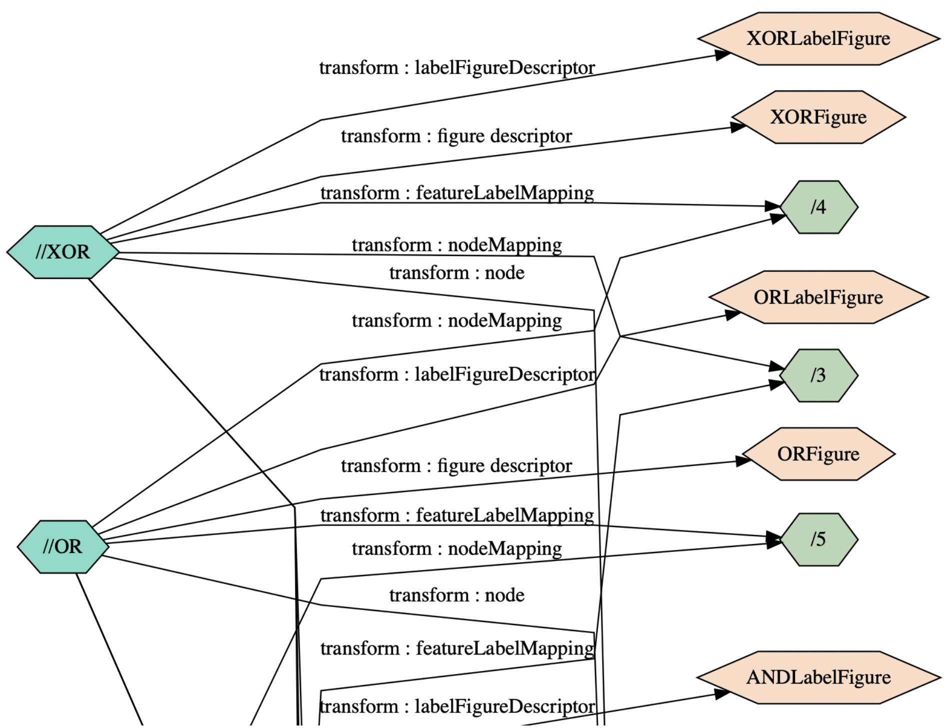


Figure 6.12: Partial management trace view of the EOL program executed by the `Ecore2GMFToolGraphMap` task. Model elements on the left belong to the `ECore` model while green model elements on the right belong to the `GmfGraph` model and beige ones to the `GmfMap` model. The arrows show the name of the trace relationship.

execution of their task definition types to listen for output files or elements being generated and capture those traces. As an example, Figure 6.13 shows a

fragment of the traces captured during the execution task `GmfMap2GmfGen`.

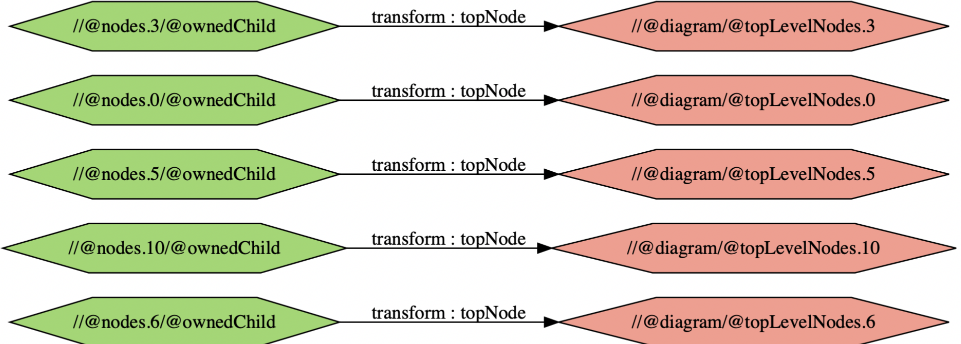


Figure 6.13: Partial management trace view of the `GmfMap2GmfGen` task. Green model elements on the left belong to the `GmfMap` model while beige ones on the right belong to the `GmfGen` model. The arrows indicate the trace relationship between the elements, in this case the stage at which output model elements were created.

6.2.5 Performance results

We report on the execution of the EuGENia workflow under the change scenarios described in Sec. 6.2.3 both in with the original implementation (`Orig`) and with its ModelFlow adaptation (`MF`).

The value reported for the first scenario corresponds to the first execution, while all other scenarios report on the second. Each approach reports the average of 20 executions that were performed for each scenario, having previously discarded 5 warm-up executions. Both the original and the ModelFlow approaches were automated in a similar fashion using parametrised Java tests and reusing the automated change scenarios. The original implementation had to be adapted for both the ModelFlow and *original* approaches to be comparable and measurable. We had to listen for asynchronous jobs to complete and to enable or disable delegates to have an equivalent execution to ModelFlow[4]. The experiments were executed on an 8-Core Intel Core i9 CPU @ 2.3 GHz with 16 GB of RAM and the Java Virtual Machine was provided with up to 4GB of memory running with JDK 1.8.0_231.

Results

Figure 6.14 shows the mean execution time in seconds per scenario and approach (`Orig, MF`). The black lines on top of the bars show the 95% confidence interval. Overall, the approaches were not significantly different in *scenarios*

[4]EuGENia is in active development, so we had to choose a snapshot to replicate it in ModelFlow which then evolved.

i, vi, whereas in *scenarios ii, v* MF outperformed Orig and in *scenarios iii, iv* Orig outperformed MF.

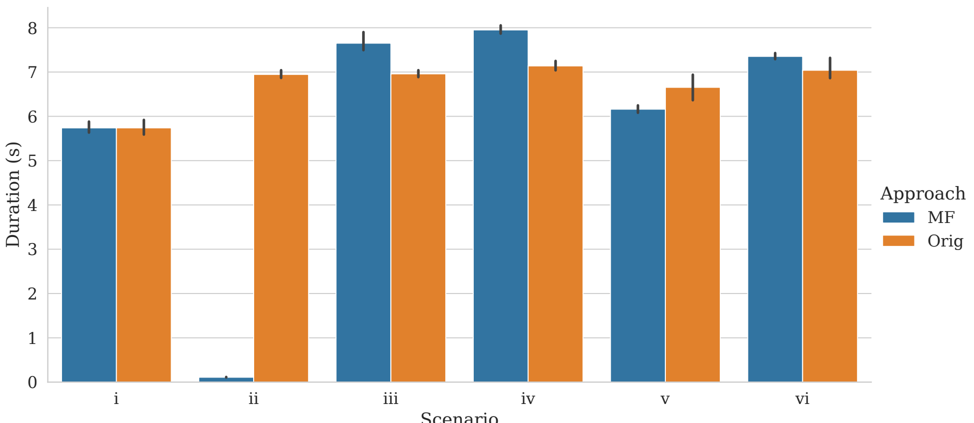


Figure 6.14: Mean execution time (s) per scenario and approach. Black lines at the top of each bar represent 95% confidence intervals for the mean.

Compared to Orig, ModelFlow's features (conservative executions, model management and management traces) can produce an additional overhead. However, some of these additional features like conservative execution and model management should pay-off when artefacts of the workflow change. To understand how these features affect the workflow execution performance we provide Table 6.2 which shows the time in seconds spent by MF in different execution phases. Among these execution phases we find the *core task logic* which is executed after the task has been configured and all the inputs (parameters and models) have been processed and just before any outputs or traces need to be processed. Also, the *execution graph* and *dependency graph* stages represent the time spent resolving these graphs. The stages *process inputs* and *process outputs* represent the time spend processing the stamps of input parameters such as the program of an EOL task, but also that of task outputs such as the generated files by a code generator. Similarly, the stages *process models before/after execution* represent the time spent processing the configuration of models and their stamps. The previous stages do not contain the time spent loading or disposing the models, these times are presented in their own stage: *Load* or *Dispose*. We can observe that the time spent loading models is very small. This is partly because models are small and of type EMF. Other model formats like Simulink take longer to load. In contrast, we can observe that the time spent disposing the models is more expensive. This is likely due to the overhead of the serialization and of write operations. Figure 6.15 provides a graphical view of the execution time of these phases (stacked) per scenario (excluding the core task logic). Overall, the core task logic execution phase is

Table 6.2: Execution time (ms) by execution stage for `MF`

Scenario Stage	i	ii	iii	iv	v	vi
Dependency graph	0.35	0.30	0.33	0.30	0.30	0.31
Execution graph	2.69	2.97	2.49	3.44	3.03	3.65
Process inputs parameters	1.36	0.42	1.26	1.32	1.33	0.73
Process models before execution	62.4	2.15	64.2	67.0	68.7	90.5
Load models	0.092	0.008	0.088	0.077	0.079	0.081
Core task logic	5564	50.8	7457	7753	5958	7131
Process outputs parameters	16.4	0.058	14.4	14.1	14.6	13.8
Process models after execution	47.2	0.052	45.3	49.1	48.3	42.0
Dispose models	5.59	0.89	5.48	5.44	5.73	4.14
Management Traces	0.25	0.045	0.22	0.23	0.26	0.14

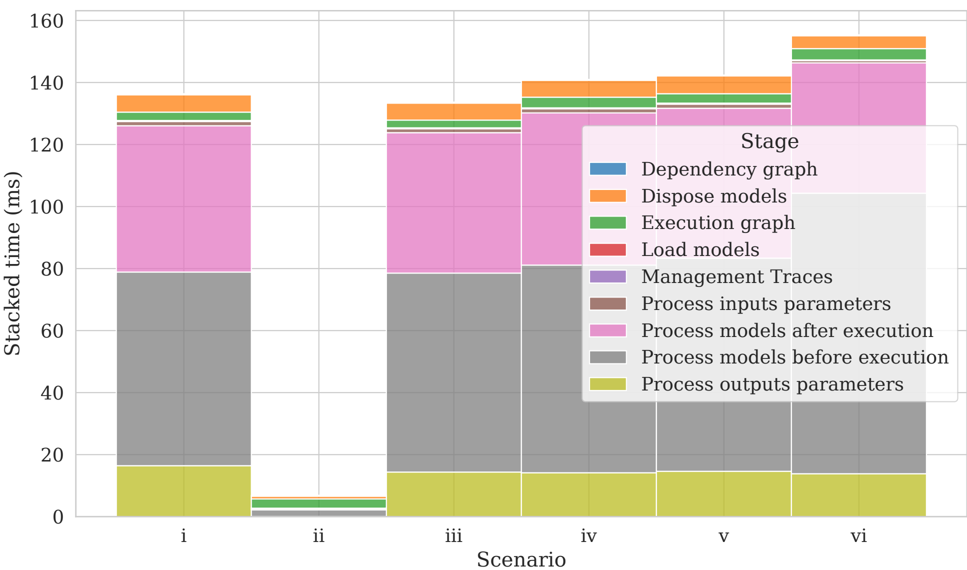


Figure 6.15: Time spent in ModelFlow features per scenario

above 96.7% of the total execution in *scenarios i,iii-vi*, which indicates a total overhead of 3.3% due to ModelFlow features and base processing. In these scenarios, the phases that have the most impact on the performance are the model processing phases which compute their stamps.

In all approaches the EuGENia workflow is more time consuming in subsequent executions (*scenarios ii-vi*) compared to the first execution, except for `MF` in *scenario ii*. This is partly explained by the execution of the `GenerateDiagramCode` task which takes longer to execute in executions where the `GmfGen` model already exists as shown in Figure 6.16. This figure shows the mean execution time of the *core task logic* execution phase of the different tasks in the workflow (in seconds) per scenario in the two `MF` approaches.

Figure 6.16: Core task logic execution times (in seconds) per scenario. This plot is provided to demonstrate the dominance of the `GenerateDiagramCode` task, particularly after the first execution.

6.2.6 Discussion

Regarding correctness, ModelFlow was able to capture the EuGENia workflow and to execute the required tasks in all the evaluated change scenarios based on their task and model dependencies. Contrary to our predictions, Scenario *iv*, in which the Emfatic is modified by adding a GMF annotation, required the `GenerateDomainModelCode` task to re-execute despite having the same outcome as previous executions. This is explained by the chain of tasks that modify a the `GenModel`: `Ecore2GenModel` (produces), `FixGenModel` (modifies) and `PolishGenModel` (modifies). In the second execution of this scenario, `Ecore2GenModel` is executed because the `Ecore` now has the new annotation, however the `GenModel` does not change from its past execution, but it changes from the last version that the workflow produces (after the modifications). As such, ModelFlow triggers both `FixGenModel` and `PolishGenModel` once more. While this is correct, in practice it would be convenient to have checkpoints for models that could be aware of these potential situations and prevent their execution.

Overall, `Orig` was faster than `MF` in scenarios where most (if not all) tasks had to be re-executed (*scenarios i, ii-v*). However, `MF` proved to be faster in scenarios where not all tasks were executed (*scenarios ii and vi*), making the input and output tracking worthwhile. EuGENia is a workflow in which tasks build on previous work and reuse the models throughout the workflow execution. As such, changes in a reused model such as the `Ecore` affect most of the workflow execution. Therefore, ModelFlow may be more useful in workflows where models are less interconnected with each other.

By far, processing models before and after execution, to determine if they have changed compared with the execution trace were the stages that caused most overhead excluding the core task logic execution. This suggests, that having an approach able to detect changes on models and to provide a mechanism to compute their stamps efficiently could improve the performance of `MF`. Similarly, more efficient ways to process the stamps of outputs and inputs could help improve ModelFlow's performance.

With the current implementation, the collection of management traces does not seem to impact the performance of `MF` significantly. Notice that the *end-to-end tracing* value reported in Figure 6.15 measures the moment where ModelFlow asks a task instance for its collected traces to merge them with the workflow's trace; it is not measuring tracing while the tasks instances are executing nor when they translate traces into ModelFlow's format. While some task definitions could have been instructed not to collect traces during their execution (e.g., gmf:genDiagram), others like `epsilon:etl` always produce such information as is required for execution. In the latter case, additional

overhead would be caused by the translation to match ModelFlow's metamodel and would depend on the task definition implementation.

There are technical challenges that should be considered in future investigations like the reuse of models across frameworks. For example, while it was possible to reuse Epsilon EMF model instances (e.g., `epsilon:emf`) in non-Epsilon tasks (e.g., `gmf:genDiagram`), it was challenging to keep them synchronised. It would be worth exploring mechanisms to reuse models like EMF across frameworks like Epsilon, GMF and ATL without losing information and with relative ease. For example, it would be convenient to use the same EMF model in an Epsilon validation program but also in an ATL transformation without requiring a dedicated EMF model definition for each framework (e.g., *atl:emf* and *epsilon:emf*) representing the same model.

6.2.7 Threats to validity

A threat to construct validity is the differences in the implementation of the task definitions between ModelFlow and the original implementation. To ensure that we could recover management traces and to identify inputs and outputs (e.g., generated files) we had no option but to modify some of the task definitions to extract such information. Similarly, some of the these were also modified to avoid loading or storing the models, as ModelFlow provides mechanisms to do this when required. While these modifications make core task logic executions between approaches not comparable, the overall workflow execution (which achieves the same objective in both approaches) is comparable. Lastly, some Epsilon scripts had to be rewritten to accommodate partial workflow executions. For example, instead of blindly adding a new element to a model, we had to check if it already existed and if not add it. To mitigate this threat to validity, all approaches (`MF` and `Orig`) were executed with the same modified scripts.

Regarding external validity, the performance results from the experiment are only valid in the context of the specific workflow and change scenarios. While the feature overhead suggests that some phases of the ModelFlow process may be more expensive in time than others, they are also inherently impacted by the type and size of models used in the experiment (e.g., affects model loading, disposal and model stamp computation) just as well as the complexity of the tasks executed (e.g., may produce more traces to be processed).

6.3 Case study: Industrial workflow

This section presents a sanitised version of an industrial case study where we use ModelFlow to support the software development of an Engine Electronic Controller (EEC) for a turbine engine which must be adapted based on the

EEC physical settings and the software features supported by the turbine.

The design tool used to capture the EEC requirements and to generate its code is in active development and has been built as model-based Integrated Development Environment (IDE) atop Eclipse. Aside from the design of the controller, the IDE will support the production of certification evidence, impact analysis, traceability analysis and synchronisation between artefacts. The development of the software that controls the EEC unit involves several models including Simulink (Design, Requirements, Tests) and EMF models. The EMF model is used to capture the core concepts and features of the EEC and several Simulink models are derived from it. The design tool uses MDE facilities to generate code, validate the models, and even guide users in the development process.

In this case study we selected a fragment of a model management workflow used by the design tool to support the generation of Simulink models and code from the main EMF model. This fragment workflow was also selected to demonstrate the use of heterogeneous models in ModelFlow, in particular, regarding the use of Simulink models. The goal of this case study is to qualitatively evaluate if ModelFlow can capture this workflow and to discuss the advantages it offers along with its shortcomings. In particular, we report observations regarding its conciseness, potential language optimisations, visualisation facilities, user interaction, recovered model management traces, and conservative executions.

6.3.1 Background

The selected workflow includes various model management tasks including a model-to-model transformation that generates a Simulink model and four model-to-text tasks that generate code. The workflow implemented by the design tool consists of a combination of Java files and Make files (see Sec. 2.3.3). In practice, the workflow consists of three activities that are executed manually and independently: Code generation from the EEC model, Simulink model generation from the EEC model, code generation from the Simulink model.

We describe the types of artefacts and data involved in this case study to illustrate the size of the project.

EMF model. The EMF model is the artefact that describes the functionality of the software that controls the EEC. It defines electronic, software, network, and service components along with their interconnections. The EMF model used in the case study has a total of 2451 model elements (4 of them are service components) and its Ecore metamodel contains a total of 234 classes.

EMF-to-Simulink transformation. This model-to-model transformation is executed for a given service component from the EMF model and it generates a corresponding Simulink model. The workflow uses the Simulink driver presented in chapter 5 to handle Simulink models during this transformation. The transformation is composed of 14 ETL files and 1 EOL program files that amount to approximately 3250 lines of code.

Simulink model. This is the model that is generated for each service component in the EMF model through the ETL transformation above. The generated Simulink model is later used to run simulations and to generate code from the model. For illustrative purposes, the largest generated Simulink model contains 1513 Simulink blocks and is persisted in a file of 116KB.

Code generation. Two approaches are used to generate code from the different models. The first approach consists of a combination of EGX/EGL programs used to generate C code from the EMF model. Currently, the workflow uses 7 EGX generators, 49 EGL templates and 7 EOL programs to support the code generation. Table 6.3 shows the number and type of files generated through EGX which in total amounts to 234 files.

The other approach consists in the execution of Make build targets that invoke MATLAB functions, which generate C code. The code generated from the EMF model is complementary to that generated from the Simulink models and is mostly used for initialisation purposes.

Traceability information. Trace information is extracted from the model-to-model transformation which is then saved in a text file. Similarly, the code generated with MATLAB functions produces a dedicated website is with code documentation and traces from Simulink blocks to lines in the generated code.

6.3.2 Approach

In this section we present a sanitised version of the EEC fragmented workflow captured with ModelFlow. We have identified three separated activities that are executed independently and captured them in the same ModelFlow workflow. As in the original implementation, this workflow is executed for a given service component. The result of the workflow is the generation of a Simulink model and of code artefacts derived from it and from the input EEC model. The dependency graph of the workflow is presented in Figure 6.17.

Parameters. Listing 6.15 shows the parameters that are required to configure the workflow. `repo` is the source path of the repository that contains all the transformations and utilities to support the generation of the EEC

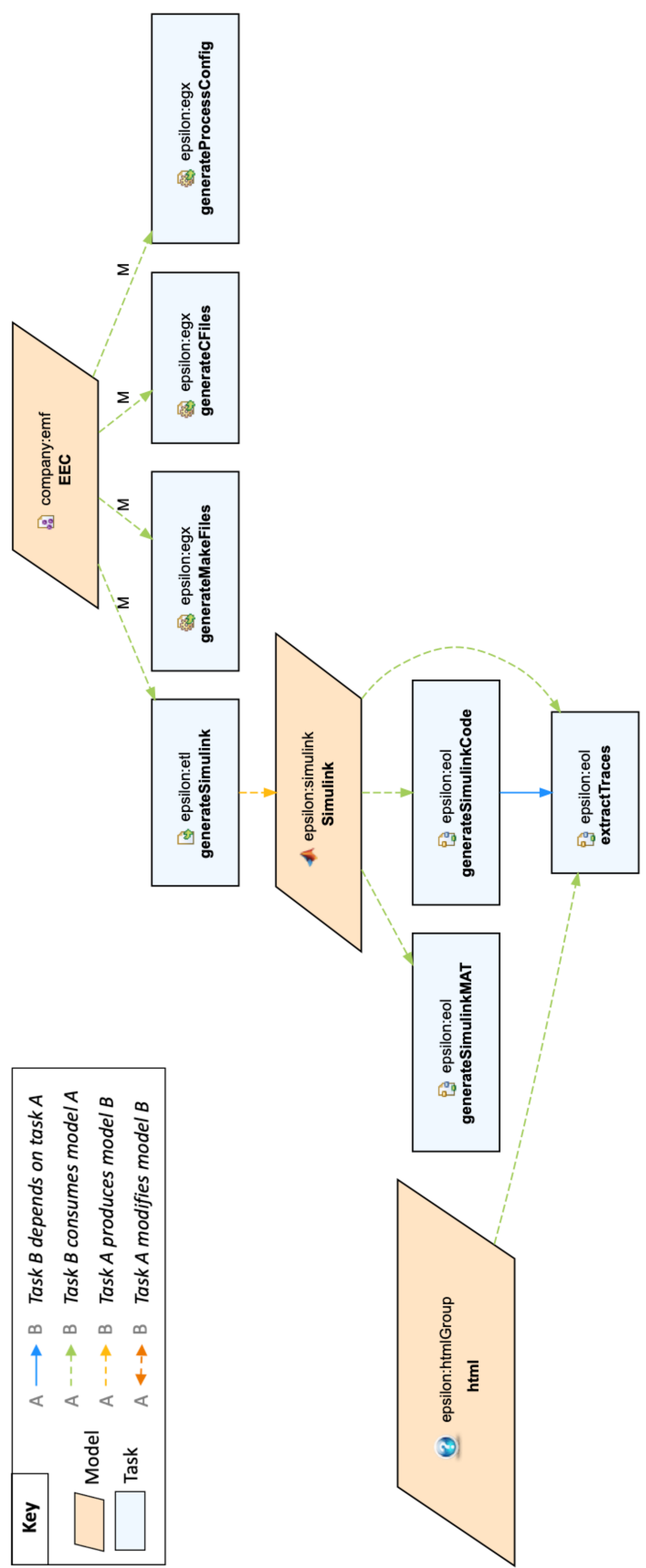


Figure 6.17: ModelFlow dependency graph of industrial case study.

Table 6.3: Generated files through EGX.

Task	Files	Extensions
generateMakeFiles@true	2	mk
generateMakeFiles@false	8	mk, bif
generateCFiles@true	131	c, cfg, psprj, h, mk, bif
generateCFiles@false	24,	c, cfg, h, mk
generateProcessConfig@1	8	c,h
generateProcessConfig@2	18	c,h
generateProcessConfig@3	0	
generateProcessConfig@4	6	c,h
generateProcessConfig@5	0	
generateProcessConfig@6	6	c,h
generateProcessConfig@7	0	
generateProcessConfig@8	7	c,h
generateProcessConfig@9	5	c,h
generateProcessConfig@10	12	c,h
generateProcessConfig@11	0	
generateProcessConfig@12	6	c,h
generateSimulinkMAT	1	mat

software. `outDir` is the output directory where Simulink models will be created. `serviceName` is the name of the service component for which the Simulink models and code will be generated. `pathHandler` is a Java utility passed at runtime that has information about the structure of the workspace and is used by the different tasks to locate elements or know where to create new ones.

```
1 param repo : String;
2 param outDir : String;
3 param serviceName : String;
4 param pathHandler; // Java Object
```

Listing 6.15: ModelFlow parameters for the industrial case study.

Models. Listing 6.16 shows the declaration of the two models used in the original workflow: the EEC EMF model and the Simulink model. Lines 1-8 show the EEC model, which is configured by indicating the source model file, the metamodel URIs involved, a dedicated URI mapping and the `expand` flag which is used to resolve proxy elements. Lines 9-15 show the configuration of the Simulink model which takes the source file, the MATLAB engine location, the MATLAB library path and the project of the model. By specifying the project of the model, we ensure that the model is loaded properly.

```
1 model EEC is epsilon:emf{
2   src : "model.system"
3   metamodelUri : "http://www.company.com/X/v1.0"
```

```
4   metamodelUri : "http://www.company.com/X/A/v1.0"
5   expand: true
6   uriMapping : Map{"uriToMap"="actualUri"}
7 }
8 model Simulink is epsilon:simulink {
9   src: outDir + "\\" + serviceName + ".slx"
10  project: "projectLocation.prj"
11  engine: matroot + "\\path\\engine.jar"
12  library: matroot + "\\path\\library\\"
13 }
```

Listing 6.16: `EEC` and `Simulink` models in ModelFlow.

Model-to-model transformation. Listing 6.17 shows the `generateSimulink` ETL transformation which is responsible for producing the Simulink model. It requires as input the name of the service component for which the model will be generated along with the location of the source repository.

```
1 task generateSimulink is epsilon:etl
2     in EEC as M
3     out Simulink {
4   src: "generate.etl"
5   params {
6     var map = new Map;
7     map.put("serviceName", serviceName);
8     map.put("sourcePath", repo);
9     return map;
10  }
11 }
```

Listing 6.17: `generateSimulink` ETL transformation in ModelFlow.

Code generation with EGX. Listing 6.18 shows the three code generation tasks with EGX. The first one is the multi-task `generateMakeFiles` (lines 1-12) which produces Make files. This task receives the `pathHandler` and a Boolean flag that affects the generated output. As indicated in line 3, it is executed for each of the values that `flag` can take, in this case `true` and `false`. A similar structure is used for task `generateCFiles` (lines 13-24) which instead of producing Make files, it generates C programs.

The last task `generateProcessConfig` is a special type of multi-task that iterates over *model elements* from the input EMF model. Line 28 shows the EOL statements that defines the set of parameters to be used to create task instances. Overall, these statements iterate over elements of type *Configuration* from the EMF model and selects those that have the `generate` property set to true. From this collection, sequence of maps is created, each with the keys

`flag` and `config`. This results in 12 combinations as there are 6 *configuration* model elements. Furthermore, line 27 shows how each task instance is assigned a dynamic name resolved from the information in the corresponding map. For example, for the map with `{flag=true, config="ConfigX"}` the task will be named `generateProcessConfig@ConfigX-true`.

```
1  task generateMakeFiles is epsilon:egx
2      in EEC as M
3      forEach flag as: flag.toString() in: Sequence{true,
           false}
4  {
5    src : "generateMake.egx"
6    params {
7      var map = new Map;
8      map.put("flag", flag);
9      map.put("path", pathHandler);
10     return map;
11   }
12 }
13 task generateCFiles is epsilon:egx
14     in EEC as M
15     forEach flag as: flag.toString() in: Sequence{true,
           false}
16 {
17   src : "generateC.egx"
18   params {
19     var map = new Map;
20     map.put("flag", flag);
21     map.put("path", pathHandler);
22     return map;
23   }
24 }
25 task generateProcessConfig is epsilon:egx
26     in EEC as M
27     forEach setup as: setup.get("config")+"-"+setup.get
         ("flag") in {
28       return EEC!Configuration.all().select(c|c.
           generate).collect(c| Sequence {true, false}.
           collect(flag| Map{"flag"=flag, "config"=c.name
           }).flaten()).flatten();
29     }
30 {
31   src : "generateProcessConfig.egx"
32   params {
33     var map = new Map;
34     map.put("flag", setup.get("flag"));
```

```
35        map.put("config", setup.get("config"));
36        map.put("path", pathHandler);
37        return map;
38      }
39    }
```

Listing 6.18: Code generating multi-tasks in ModelFlow.

Code generation from Simulink model. The original implementation of this task uses Make build targets to generate code from the Simulink model. In practice the make scripts were used to (a) invoke a MATLAB function that performed the code generation and (b) generate a MAT file. We have migrated the invocation of the MATLAB function to EOL to make use of the Simulink driver to invoke the code generation. Listing 6.19 shows the corresponding tasks `generateSimulinkCode` (lines 1-7) and `generateSimulinkMAT` (lines 8-13).

This translation had the advantage that we could re-use the `pathHandler` for information that did not need to be repeated, and, for the case of `generateSimulinkMAT` traces could be created within the EOL program. We set out to retrieve trace information for the task `generateSimulinkCode` in a separate task because we were unable to retrieve this information from the MATLAB function that produces the code. As such, the task was annotated with `@noTrace` to skip trace processing.

```
1   @noTrace
2   task generateSimulinkCode is epsilon:eol
3       in Simulink
4   {
5     src : "generateSimulinkCode.eol"
6     params : Map{"path"=pathHandler}
7   }
8   task generateSimulinkMAT is epsilon:eol
9       in Simulink
10  {
11    src : "generateSimulinkMat.eol"
12    params : Map{"path"=pathHandler}
13  }
```

Listing 6.19: Code generating tasks from the `Simulink` model in ModelFlow.

Recovering external traces. The invocation of the MATLAB function that generates code form Simulink models produces a set of HTML pages with documentation and trace information that links Simulink blocks, Stateflow blocks and MATLAB functions to the generated program files and the lines in which they are referenced. We have provided additional tasks to the workflow to

recover those traces from the HTML pages and integrate them in the workflow management trace model.

Task `extractTraces` in Listing 6.20 parses the set of HTML pages and extracts traceability information from them. Lines 1-3 show a custom HTML model group that is used as input for the trace extraction task. This model group represents a collection of Epsilon HTML models, each representing a single HTML page. To configure this model group, we specify the root of the website folder. At runtime this model will locate all HTML pages under that folder. Then lines 4-9 show the EOL task declaration that performs the trace extraction.

```
1  model html is epsilon:htmlGroup {
2    root: "generated/folder/with/traceInfo/"
3  }
4  task extractTraces is epsilon:eol
5      dependsOn generateSimulinkCode
6      in html and Simulink
7  {
8    src: "extract.eol"
9  }
```

Listing 6.20: HTML model group and trace extraction task in ModelFlow.

6.3.3 Results

Management traces. Table 6.4 summarises the number of model management traces produced from the workflow execution[5]. The *traces* column indicates how many `Trace` elements were created (see Sec. 4.4.4). Since the trace links can have multiple sources or target elements, the *combinations* column sums the product of the number of targets and sources of each link created by the task ($\sum_{t=1}^{n} |sources| * |targets|$). Overall, there were 223 unique files generated and 2296 different model elements involved in the workflow. Most of the traces linked one source element to a target element, except the `generateSimulink` ETL task in which transformation rules can generate multiple target model elements.

[5]The name of `generateProcesConfig` task instances has been sanitised to use numbers from 1 to 12 rather than information from model elements.

Table 6.4: Number of traces extracted by ModelFlow from the execution.

Task	Traces	Combinations
generateSimulink	80	1239
generateMakeFiles@true	28	28
generateMakeFiles@false	275	275
generateCFiles@true	40733	40733
generateCFiles@false	8255	8255
generateProcessConfig@1	531	531
generateProcessConfig@2	1383	1383
generateProcessConfig@3	0	0
generateProcessConfig@4	165	165
generateProcessConfig@5	0	0
generateProcessConfig@6	165	165
generateProcessConfig@7	0	0
generateProcessConfig@8	252	252
generateProcessConfig@9	157	157
generateProcessConfig@10	464	464
generateProcessConfig@11	0	0
generateProcessConfig@12	177	177
generateSimulinkMAT	1	1
extractTraces	6205	6205

6.3.4 Discussion

Conciseness. The original implementation provides a set of smaller workflows that are independent and manually invoked by the user. One of such workflows generates the Simulink model for a given service and is captured in a few Java classes (using 212 lines of code[6]) that invoke the ETL transformation. The workflow that generates the code from the EEC model requires a few Java classes (using 96 lines of code) to invoke all the EGX tasks for a given service. Finally, the workflow that generates the C code invoking MATLAB functions is distributed across 3 Make files.

In this case study, we put together all these tasks in the same ModelFlow workflow in a single declaration file with 118 lines of code. The Make files that generated C code were translated to EOL programs that were invoked from the workflow declaration.

Visualisation. To support design tool users, developers originally provided a dedicated procedural view that highlighted the different activities to manually

[6]Counting lines in the body of methods only

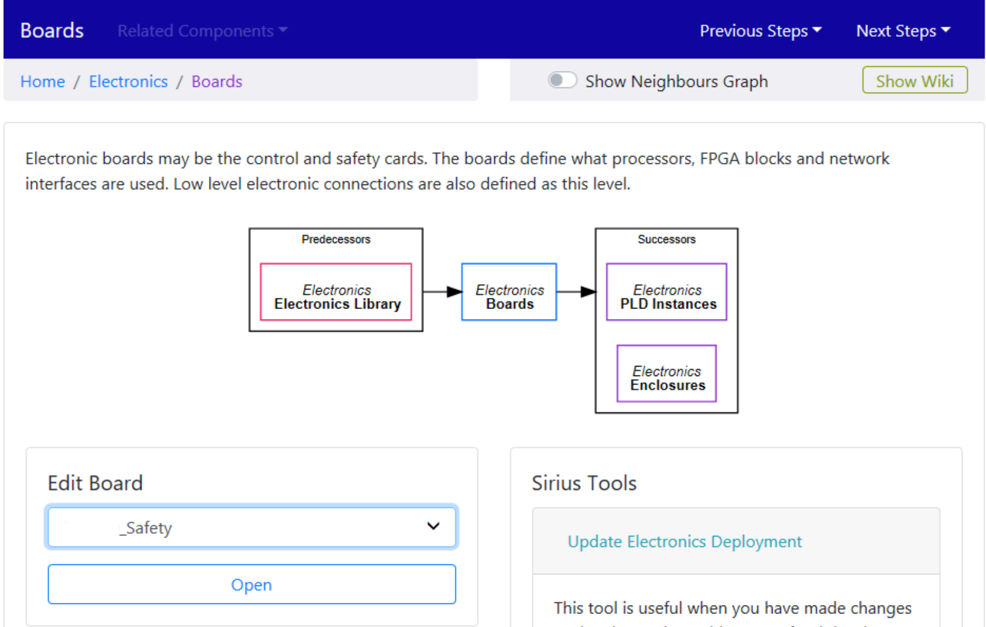


Figure 6.18: Step navigator view of the original EEC design tool.

execute (e.g., generate Simulink models or generate code) at different stages of the development process. This view has recently been updated to show a graph-based view of step interdependencies where each step offers a list of available tasks to trigger manually (Figure 6.18). While this is a helpful navigation tool for users it serves a different purpose than the visualisation provided by ModelFlow, as not all tasks are meant to be executed in a workflow (e.g., open a diagram view). The ModelFlow Picto view automatically generates a graph of task dependencies of an automated workflow making interactions between models and tasks visible to users. This view is not executable but making tasks in the workflow individually executable could facilitate user's interaction with the workflow or parts of it.

User interaction. Users of the original workflow work on different aspects of the model and tend to execute individual tasks on demand and in a prescribed order. With ModelFlow, only the tasks that need re-execution would be executed. We recognise that ModelFlow would benefit from user interface facilities that allow the execution of a single task from the workflow or to use a task as target of the execution[7] or as starting point[8]. Alternatively, users could also be given the ability to execute a group of tasks. For example, the EGX tasks could be grouped as in a `CodeGeneration` workflow or task group.

[7]Invoking the task dependencies and the task itself only
[8]Invoking the task and then any other tasks that depend on it

Management traces. ModelFlow has demonstrated its ability to retrieve traces from tasks that produce them, and to create them when they do not, as with EOL tasks. Currently, only EOL can create a custom trace at runtime although other tasks like ETL can be extended to both retrieve tasks and allow the creation of custom traces within their programs. This case study has also demonstrated a use case for the `@noTrace` annotation. With ModelFlow the management traces of the workflow are maintained through their executions. However, a future area of improvement for trace maintenance is giving users the ability to choose a custom trace metamodel that may use different naming and structural conventions.

Conservative executions. In the original workflow, conservative executions are supported for tasks that are defined in Make. Make provides these checks out-of-the-box as build targets specify their file dependencies. For the workflow tasks that are invoked through Java (executing Epsilon tasks) in the original workflow, the up-to-date checks are not available. With ModelFlow this functionality becomes available for all tasks.

The inherent limitation with Make is that all dependencies must be manually specified and that there is no support to dynamically declare them. The limitation with ModelFlow is that no implicit inputs or outputs can be determined at runtime beyond those resolved by the task definition type. We go over this in the next discussion point.

Regarding the response to changes, we observe with ModelFlow that no tasks are executed if there are no changes to any model or task artefacts. However, even small changes in the EEC model would trigger the re-execution of the ETL and EGX tasks. Because the ETL task is time-consuming, an approach to exploit the ModelFlow conservative execution facilities at the moment would involve the refactoring of the EEC model into various smaller models so that the ETL transformation is updated to use the relevant ones. Nevertheless, it would be convenient if ModelFlow, modeling frameworks like EMF or supporting model management frameworks like Hawk provided a mechanism to detect changes to parts of the model relevant to the task to be executed.

Runtime inputs and outputs. ModelFlow determines inputs and outputs from task definition types. For example, EOL tasks parse the source program to check for imported files and reports them as inputs before the task executes. However, this case study demonstrates the need to provide the ability to report implicit inputs at runtime. Take the tasks `generateSimulinkCode` and `generateSimulinkMat`, which invoke MATLAB functions from EOL to generate code or a MAT file, and the various other tasks that use the `pathHandler` parameter to locate input and output files at runtime. In the former case, the

MATLAB function being invoked is a static String in the EOL program which the task definition is unable to resolve as an input before the execution. Similarly, while the generated MAT file can be declared as the target of a management trace at runtime, it cannot be declared as an output to be analysed when determining if a task should be re-executed.

Currently, ModelFlow cannot declare inputs and outputs at runtime in a similar fashion as management traces can be created in EOL programs. To support this feature, ModelFlow would need to recover this information after the task's execution and use it in subsequent invocations to determine if it is up to date. Not being able the track this information can make build executions un-sound as tasks could be skipped when they need to re-execute.

Language optimisations. In the original workflow, users need to specify a service component to generate the corresponding Simulink model and code. However, the Make files that generate code are executed for the Simulink models of all service components (4 in total). The workflow captured with ModelFlow works exclusively for a single service component which means that 4 ModelFlow invocations with different parameters would be needed to generate all the required artefacts.

Ideally, the workflow should capture the execution for all available service components while allowing users to decide whether to execute the full workflow or just parts that affect a service component. In practice we could declare 3 more Simulink models, 3 more ETL tasks for each output Simulink model and adapt the EOL tasks to work with the 4 Simulink models. A useful addition to ModelFlow would be the support of extensible model and task declarations. For example, with this feature, concrete Simulink models could be created with different source files while sharing the rest of their configuration (see Listing 6.21).

```
1 @abstract
2 model AbstractSimulink is simulink {
3   engineJar: "..."
4   libraryPath: "..."
5 }
6 model SimulinkServiceX extends AbstractSimulink{
7   src: "serviceXmodel.slx"
8 }
```

Listing 6.21: Supporting model and task declaration extension

Alternatively, ModelFlow could dynamically generate models[9] and match them at runtime with tasks. For example, Listing 6.22 shows a proposition for the dynamic generation of Simulink models for each service in the ECC model

[9]As multi-tasks are created through the `forEach` construct.

(line 3). Similarly, line 9 shows how the `generateSimulink` task could use a statement block environment to locate the corresponding model for each task instance.

```
1  model EEC is emf;
2  model Simulink is epsilon:simulink
3      forEach x in: EEC!Service.all() as: x.name {
4    src: x + ".slx"
5  }
6  task generateSimulink is epsilon:etl {
7      forEach x in : EEC!Service.all() as: x.name
8      in EEC
9      out {
10       return Model.all().selectByType(simulink).select(
             m|m.name.endsWith(x))
11     }
12 {
13   src : "transform.etl"
14 }
```

Listing 6.22: Dynamically model generation ModelFlow proposal with model matching in tasks

Another approach to capture the full workflow would be to use nested workflows. Listing 6.23 illustrates a (currently unsupported) workflow (lines 2-13) able to declare a Simulink model along with all tasks that use it and which could be instantiated multiple times (line 4). Arguably, this would be one of the cleanest solutions and is discussed more in detail in Sec. 7.3.

```
1  model EEC is epsilon:emf {...}
2  workflow serviceProcess
3    in EEC
4    forEach service in: EEC!Service.all()
5  {
6    model Simulink is epsilon:simulink {
7      src: service.name + ".slx"
8    }
9    task generateSimulink is epsilon:etl
10     in EEC
11     out Simulink
12   {...}
13 }
```

Listing 6.23: Nested workflow ModelFlow proposal for industrial case study

6.4 Extensibility

This section summarises the instances in which ModelFlow has demonstrated to be extendible to support heterogeneous tasks and models. In particular, this has been demonstrated through three case studies. For example, in Sec. 6.2 we used added support for EMF and GMF tasks that are not part of the Epsilon family of languages. Similarly, in Sec. 6.3 we used the Simulink model definition presented in Sec. 5.6 and a new HTML model group. The support for heterogeneous model and task was considered in objectives *i.i* and *i.ii* from Sec. 3.2.2. The case studies demonstrate that these objectives have been achieved.

6.5 Interoperability

We have provided different mechanisms that can be used to execute ModelFlow. In addition to the core Java libraries that can be reused by Java projects, we have provided an Eclipse run configuration (Sec. 6.5.1) which allows Eclipse users to trigger executions from the user interface, and an integration with the Maven build tool (Sec. 6.5.2), which allows projects built with Maven to invoke ModelFlow workflows during the build process.

6.5.1 Eclipse

Run configuration. ModelFlow extends the run configuration of Epsilon languages to support its own. One difference to other Epsilon languages is that the `Models` tab is not needed as all models are declared in the program.

Through the run configuration, users can determine whether to run ModelFlow in interactive mode. In this mode the execution will ask the user for permission to re-execute a task that has the potential to be a destructive operation, that is, a task whose outputs have been externally modified. Alternatively, a user can disable these prompts and choose whether the execution should avoid destructive operations by enabling the option *protect outputs*. The run configuration also has a convenience button to clear execution traces to ensure that the next invocation triggers a full execution or to discard corrupted traces e.g., if tasks or models in the workflow are renamed. Similarly, a user can enable the recording and/or storage of management traces.

6.5.2 Build tools

Maven. Maven is a popular build tool for Java and Eclipse-based projects. It is an opinionated framework that expects projects to be formatted in a specific way and build processes to follow a pre-defined lifecycle. ModelFlow provides a prototypical integration with Maven that allows it to be invoked within

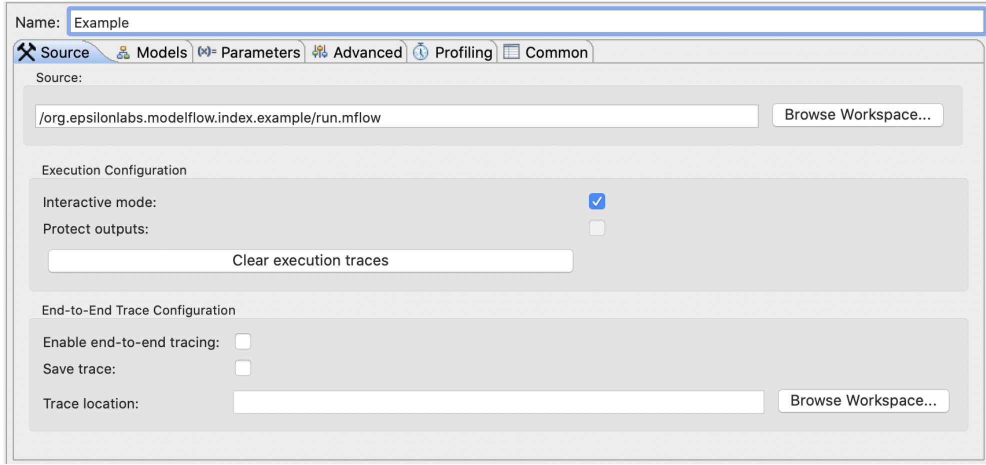


Figure 6.19: ModleFlow run configuration

this build process. Listing 6.24 shows an example usage of the implemented ModelFlow plugin identified by the version, artefact and group identifiers indicated in lines 4-6. This plugin requires the ModelFlow workflow definition file to be provided (line 10) and the execution goal `mflow` to be indicated (line 13). In this example, the ModelFlow execution would be invoked at the Maven `compile` phase as indicated by line 15.

```
1  <build>
2    <plugins>
3      <plugin>
4        <groupId>org.epsilonlabs.modelflow</groupId>
5        <artifactId>modelflow-maven-plugin</artifactId>
6        <version>1.0-SNAPSHOT</version>
7        <executions>
8          <execution>
9            <configuration>
10             <src>${project.basedir}/src/main/resources/
                   test.mflow</src>
11           </configuration>
12           <goals>
13             <goal>mflow</goal>
14           </goals>
15           <phase>compile</phase>
16         </execution>
17       </executions>
18     </plugin>
19   </plugins>
20 </build>
```

Listing 6.24: Sample Maven ModelFlow plugin

6.6 Summary

This chapter presented the evaluation of ModelFlow through three case studies. The first case study evaluated the execution of an artificial workflow in ModelFlow and Gradle under different change scenarios that affected various types of artefacts (e.g., source programs, intermediate models, generated files). The results of the case study suggest that both tools can be adapted to implement the model management workflow although ModelFlow was able to process dynamically generated outputs and was able to configure how to compute stamps of input and outputs which allowed it to discern between protected and non-protected regions in generated files.

The next case study evaluated the correctness and performance of ModelFlow when reproducing the EuGENia model management workflow which generates a graphical editor from an annotated metamodel. Like the first case study, this case study was also executed under different change scenarios that were realistic for the type of workflow. The results show that ModelFlow was able to correctly execute the workflow in response to the changes while also being able to collect traces from tasks that produced them or not by default, and to only load models once per execution and only when they are needed. Similarly, ModelFlow proved to be faster than the original implementation in the change scenarios that did not require the re-execution of all tasks. This study also demonstrated ModelFlow's task extensibility as it was adapted to execute EMF and GMF tasks.

Finally, the last case study evaluated the ability of ModelFlow to capture and reproduce an industrial case study. ModelFlow demonstrated its model extensibility to accommodate the management of Simulink and HTML models. This case study demonstrated the use of ModelFlow constructs able to generate multiple tasks based on information from the models. While overall, ModelFlow was able to capture the workflow for a selected service from the workflow domain, the case study highlighted that ModelFlow would benefit from more features such as nested workflows.

7 Conclusion

This thesis has investigated an approach to capture model management workflows that execute conservatively and produce traceability information as a side product. This thesis has contributed to the research hypothesis defined in Sec. 3.2.1:

> *The performance of repetitive executions of model management workflows can be significantly improved with the help of a conservative interpreter that consumes declarative workflow specifications capturing dependencies among models and model management tasks. At the same time, these inter-dependencies can be used to establish and maintain traces at model element level of granularity.*

The remainder of this chapter is structured as follows. Sec. 7.1 provides an overview of the main discussions in the thesis. Sec. 7.2 describes the main contributions to the field. Sec. 7.3 provides future work.

7.1 Summary

We have presented ModelFlow, a model management workflow language and interpreter which offers conservative executions, end-to-end traceability, and lazy model loading/disposal. **Chapter 2** provided an overview of the theory behind Model-Driven Engineering, traceability, build systems and other workflows while it also reviewed the capabilities of several state-of-the-art tools used in all these domains. **Chapter 3** analysed the findings of the review and stated the research framework including the problem, hypothesis, goals, and scope. **Chapter 4** presented the architecture, design, and implementation of ModelFlow's language and interpreter while also describing the details and rationale behind algorithms and processes used to support its main capabilities. **Chapter 5** presented the architecture and implementation of the Simulink bridge with the Epsilon family of model management languages and its integration with ModelFlow. **Chapter 6** evaluated ModelFlow using three case studies. The first case study focused on change scenarios affecting different resources of the workflow, that is, input program files, program dependencies like imported files, models of all types (input, output, inout), and generated files. The second case study replicated an existing workflow that generates graphical editors from annotated

models in ModelFlow and evaluated its response to relevant change scenarios more in detail. Finally, the last case study aggregated build tasks used in an industrial context to evaluate the ModelFlow language features.

7.2 Thesis contributions

We have categorised the main contributions of the thesis into two categories: novel tools and techniques (Sec. 7.2.1) and notable additional results (Sec. 7.2.2).

7.2.1 Novel tools and techniques

The main contribution of this work is the architecture of a model management workflow interpreter that (a) combines features from build tools that allow for partial executions that only re-execute tasks transitively affected by external changes to workflow resources, while also (b) gathering and creating traceability information in a structured format as a side product of its execution, and (c) introducing a novel approach to automatically handle models that loads them when first needed and disposes them when no longer required in the workflow. Additionally, to declare these workflows, this work has proposed a model management workflow language that differs to other frameworks in the way models are used to drive the execution and in its capability to generate tasks dynamically. This architecture and language have been embodied in a prototype called **ModelFlow** which has been evaluated in this work. We discuss in detail the novelty of these features below.

Conservative executions. Conservative executions are a common feature of state-of-the-art build tools like Gradle and even research tools like Pluto. However, this is a feature that is generally absent in model-centric workflow tools. In general, conservative executions in build tools are based on the principle that inputs are non-modifiable. In contrast, modifying models is often part of a model management workflow. In this work we have achieved conservative executions for model management workflows that may modify model resources (see Sec. 6.2) by not only tracking changes in tasks' inputs (e.g. programs and models) but also to their outputs (e.g. generated files and models) and by comparing them against the latest version tracked by the workflow execution.

Traceability as side-product. Traces are a common but not mandatory side product of model management activities. However, as in build tools, model management workflow tools do not usually collect this information in a structured way. In this work we have proposed a mechanism that allows

individual model management tasks to contribute their side-product traces to an optional overall workflow trace that may be used for debugging, analysis and even certification purposes. This overall model management workflow trace conforms to a trace metamodel which homogenises the traces produced by the heterogeneous tasks executed in the workflows, making it easier to process. Moreover, these traces are maintained up to date during subsequent workflow invocations. Additionally, we have proposed facilities that allow model management tasks that do not produce traceability information to generate this information during their execution to contribute to the overall workflow trace. This mechanism consists in providing a trace building interface that allows authors of model management tasks to build relevant traces within the task's execution (as demonstrated for GMF, EMF and EOL tasks).

Automated model loading/disposal. In model management tasks two important steps are (a) loading the models into memory and (b) disposing them (which may involve saving any changes). When working with build tools or model management tasks, these activities are sometimes handled by the tool (e.g. MTC-Flow, ChainTracker, MMINT) and sometimes the user is left to decide at which point to load or dispose the models (e.g. ANT, MWE2, Gradle). When the tool automatically handles these activities, it usually does it through one of two strategies: either loading all models at the start of the workflow and disposing all of them at the end, or loading required models before a task and disposing them all after the task execution. The advantage of the former strategy is that models are only loaded and disposed once at the expense of keeping them all in memory during the whole execution. In contrast, the latter strategy loads only the required artefacts when a task is about to execute at the expense of reloading (a potentially expensive activity) and re-disposing models if these are employed by multiple tasks in the workflow. In this work we have proposed a novel strategy that delays model loading until the model is first required in the workflow and disposes it when no other task will use it. This strategy avoids reloading models unnecessarily and also delays their loading until it is actually required. However, different model loading/disposal strategies may be convenient for different models depending on how these are used in the workflow (e.g., once, in all tasks, in sequential tasks, at the start and at the end of the workflow) and the expensiveness (time- and memory-wise) of keeping them loaded or reloading them.

Output awareness. In MDE workflows some outputs, such as intermediary models or program files, may receive further input from developers (e.g., generated files with regions protected from overwriting by regeneration in EGL). In general, neither build tools nor other model management workflow tools pay

much attention to output resources. This work has demonstrated the need to track outputs to be able to conservatively execute workflows when resources like models are modified. Additionally, we have proposed alternative approaches for conservative workflow execution to respond to external modification of outputs. The first approach consists in triggering a re-execution of a task if its outputs have changed. This is appropriate when outputs are not allowed to be externally modified. The second approach consists in preventing a task re-execution (and halting the workflow) if a task's outputs have changed. This approach is convenient when overwriting the external changes is a potentially destructive operation.

Models that drive the execution. In most build tools, task interdependencies drive the execution order. Similarly, in model management workflow tools, tasks are interconnected with each other to indicate the execution flow. While in general all these frameworks allow tasks to indicate which models they consume, modify or produce, these indications have no influence over the execution order. This work allows task interdependencies to drive the execution order (with the highest priority) but also proposes a novel technique that allows models to influence the execution order depending on how they are used by the tasks in the workflow (see Sec. 4.4.1). As such, a workflow does not have to explicitly create task-interdependencies for all tasks in the workflow as long as the use of models implicitly connects them all. This approach can reduce the verbosity of the workflow specification while still allowing users to amend the proposed workflow by specifying task interdependencies when needed. While some languages (such as Gradle) can be extended to support the declaration of models as inputs, outputs or modifiable resources of a task (see Sec. 6.1), these cannot be used to drive the execution order.

Dynamic task generation. In this work we have proposed a mechanism that allows for the dynamic generation of tasks that share some common features (e.g. the type of task, the consumed models) but also allows for some variability among the generated tasks. While some build tools offer this capability (e.g., Gradle intrinsically and ANT/Epsilon through a task extension) we are not aware of any MDE tools that support this. Our dynamic task generation approach uses information from input models to generate and configure the sub tasks.

7.2.2 Notable additional results

Simulink bridge with open-source model management frameworks. Another contribution of this work is an approach to bridge Simulink models with model management frameworks that uses on-the-fly and on-demand translation

of OCL-like statements into MATLAB commands presented in chapter 5. Given the widespread use of Simulink models in industry and the potentially large size of such models, our bridge offers an alternative approach to manage these without requiring their complete upfront transformation into an EMF-compatible representation therefore avoiding expensive transformation costs for large models and potential co-evolution procedures. Our implementation, built atop Epsilon, enables comprehensive and uniform Simulink model management that includes Stateflow elements.

We have evaluated our implementation against an existing approach that requires an upfront model transformation into EMF setup using facilities from the Massif project. This experiment measured the execution time of a model validation process evaluated on a sample of large publicly available Simulink models in GitHub (up to 1.141 MB and 9536 blocks) using both approaches. Our evaluation results support the claim that the transformation of large Simulink models into an EMF-compatible representation can be very time consuming and shows that our bridge was able to reduce the execution time by up to 80% (mainly due to the transformation) in the validation process of the experiment. Further evaluations showed that the cost of continuous MATLAB communication in our implementation is far from negligible which led us to introduce optimisations for operations that work on collections of Simulink and Stateflow model elements that were able to make these operations more efficient –by up to 99% in some cases.

7.3 Future work

Evaluation against other build tools. As seen in Sec. 2.3, there are multiple build tools (e.g., Gradle, Bazel, Buck, Ninja) available along with dedicated MDE tools (e.g., MWE-2, MTC-Flow) that support workflows. As future work it would be interesting to compare the features of these frameworks more in detail and to evaluate them under different workflows and change scenarios.

Nested workflows. Currently, the ModelFlow language does not support the nesting of tasks and/or models within sub-workflows although the metamodel already defines a workflow as a task to enable nesting. To support this feature, the concrete syntax of the language would have to be extended and the interpreter adapted to be able to execute them.

An example of what we envision the syntax for a nested workflow to be is presented in Listing 7.1. In this language proposal the *workflow* keyword would be added and would be required for all workflows even those without nesting. A workflow construct would not need a task type but could still declare input

and output models as well as task dependencies and *forEach* iterations. We have indicated in the commented lines the computed identifier of the different entities in the workflow.

```
1  // main
2  workflow main {
3    // main::sample
4    model sample is epsilon:emf { ... }
5    // main::transform
6    task navigate is epsilon:eol
7        in sample { ... }
8    // main::nested
9    workflow nested
10       dependsOn navigate
11       in sample
12       out product
13       forEach elem in sample!Element.all() as elem.id
14   {
15     // main::nested::elemId::product
16     model product is epsilon:simulink { ... }
17     // main::nested::elemId::transform
18     task transform is epsilon:etl
19         in sample
20         out product { ... }
21     // main::nested::elemId::generate
22     task generate is epsilon:egx
23       dependsOn transform { ... }
24   }
25 }
```

Listing 7.1: Workflow nesting proposal

Dynamic specification for models. We refer to dynamic specification as a convenience to dynamically generate elements based on a collection. In this work we proposed the use *forEach* constructs in task declarations (see Listing 4.1) to allow the dynamic resolution of multiple tasks based on input from a collection. However, a similar mechanism is missing for model declarations.

Additional task and model definitions. We have demonstrated that ModelFlow can be extended to support more task definitions such as GMF and EMF tasks. However, it would be interesting to add support for other popular MDE tasks such as ATL, and QVT. It would also be convenient to have immediate access to some of the utility tasks available in build tools like ANT e.g., to copy files. Regarding model definitions, we have also demonstrated extensibility capabilities by providing support for Simulink models.

The main difference between the model definition interface of ModelFlow with that used by Epsilon is that Epsilon captures CRUD operations at model and model element level which ModelFlow does not, and that in addition to the configuration of the model ModelFlow also determines how to check if the model has changed from one execution to another. Since there is some overlap between the two interfaces, it would be worth exploring if these could be consolidated to get immediate access to all the model definitions already supported by Epsilon.

Alternative schedulers. ModelFlow has been implemented to use a directed graph iterated in topological order as scheduler. This scheduler has the disadvantage that all task dependencies must be known in advance. Future work will involve experimenting with other types of schedulers e.g., restarting/suspending that may enable the discovery of dependencies at runtime.

Adapting the management trace metamodel. Sec. 2.2.5 discussed how the traceability tool Capra allows users to provide their own traceability metamodel. In contrast, ModelFlow uses a single management trace metamodel as presented in (Sec. 4.4.4). Capra achieves this extensibility through an implementation that adapts the main metamodel produced by the tool into the one that users require. This adapter can be used to filter uninteresting traces, or to extract information from them to modify it. ModelFlow could be extended to support such metamodel adaptation after or during the execution of the workflow.

Cleaning outputs. One feature that is missing from ModelFlow is the ability to clean outputs. Consider a model-to-text transformation that generates a file for each model element in a collection. Generated files can become stale if one of the elements in a collection is deleted. Having a mechanism that allows the workflow to discard files that are no longer needed would allow the workflow to remain consistent. This mechanism could be attached to pre and post executions of a given task which could use information from the execution trace to decide how to dispose stale outputs.

Pre/post task executions. While task definitions can already define custom pre and post actions, they may benefit from directly receiving input and output information from past executions (from the execution trace) to e.g., clean outputs or perform incremental computations. Additionally, task declarations in the workflow program would benefit users as they could allow logging or computing information from the inputs or outputs of a task.

Partial executions. Currently, ModelFlow processes all tasks in a workflow definition. However, MDE workflows such as EuGENia (Sec. 6.2) and even GMF (Figure 7.1) are often accompanied with support to run individual tasks or to start the workflow from or up-to a specific task. Similarly, as noted by [125], build systems are often executed using a target which triggers the execution all its dependencies and then the target itself. ModelFlow could be extended to support all these kinds of *partial* executions. Dedicated views and context-menus could also be derived from the workflow definitions to support these activities through the user interface.

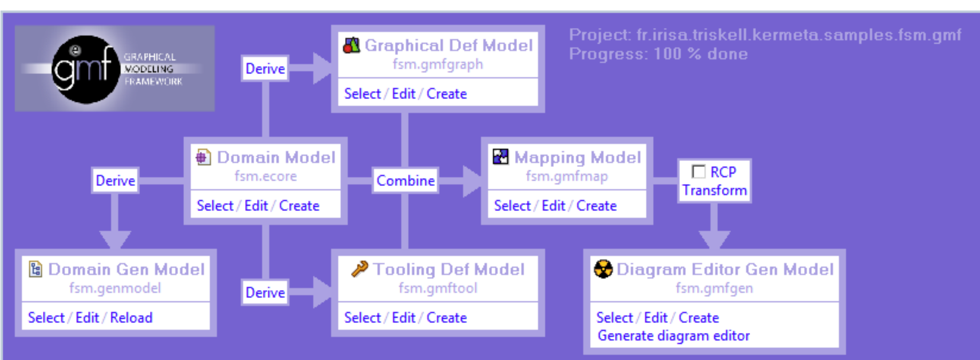


Figure 7.1: User Interface that supports the GMF execution process

Transient models. ModelFlow currently supports input, output and in/out models but it does not support transient models, i.e., those that only live in memory during a task execution. While these models would not influence the execution order, they may be traceable.

Customisation of inputs. For conservative workflows like ModelFlow or Gradle which determine whether to re-execute a task or not based on its inputs or outputs, it is imperative that the task can correctly identify these. It is the responsibility of the task writer to declare these properly. However, sometimes the programs that some of these tasks execute (e.g., EOL) could implicitly depend on resources that are not identified as inputs or outputs of the task. For example, in the industrial case study (Sec. 6.3), one of the tasks invoked external file scripts that lived outside of the EOL program being executed. In this example, while the EOL task definition can identify imported EOL programs because they are part of the EOL program structure (line 1 in Listing 7.2), it is unable to identify Strings that represent files, and which may be used within the EOL program (lines 3-4 in Listing 7.2). Without a mechanism to declare these resources as inputs or outputs, the decision to execute or not may not be entirely correct. Knowing this limitation, users can either adapt their workflow tasks (e.g., avoid invoking, reading, writing external resources within programs)

or choose to execute all tasks in the workflow as in a first-time execution. A similar mechanism to the one used to create management traces in tasks that do not provide them by default (such as in EOL programs) could be used to register inputs during a task execution. Another alternative would be to allow for declaration of additional dependent files in the task declarations. This could lead to information duplication (e.g., in the EOL program and the task declaration) but would lead to correct conservative executions.

```
1  import "library.eol";
2
3  var file = "myExternalProgram.sh";
4  executor.execute(file);
```

Listing 7.2: EOL explicit and implicit file dependencies

Incrementality. Currently, none of the model management tasks that ModelFlow supports are incremental. Incremental tasks execute only parts that are different from previous executions and they are only convenient if the cost of the execution (time-wise) is larger than the cost of determining which parts to execute. As a future research direction, incremental tools could be examined to evaluate how best to integrate them with ModelFlow.

Model change impact. In ensuing executions, ModelFlow only re-executes tasks affected by changes in their inputs (e.g., an EOL program) or used models. However, full model changes are not always indicative that a re-execution is required. Consider a state machine model that changes the name of a state and a task that reads transitions from the model. In this example, the task would detect that the model changed but for the purposes of reading the transitions, the model is the same. Currently ModelFlow cannot distinguish *relevant* changes for a given task beyond what has been defined as input or output in the task definition. While ModelFlow allows task definitions to indicate the types of changes they are looking for, a task instance cannot refine this criterion. This can result in task re-executions that may not be necessary.

Research on model impact analysis could be used to address this issue. An example of such an approach is given in [134] which tracks model element properties accessed during an execution and requires a mechanism to detect if these have changed from one version of a model to another, although it has shown some limitations with ordered collections and non-deterministic programs (e.g., with random number generators). Examples of entities that could be used to verify whether these property values have changed between model versions are both model indexes (e.g., Hawk [8]) and model comparison systems (e.g., EMFCompare [180], ECL [173]). To support the detection of relevant changes,

ModelFlow task definitions would have to be adapted to record the property accesses, while models would need to be able to verify these changes according to the record kept by the task.

Optional models. Currently, users can indicate in a ModelFlow's task declaration all the models that a task needs for its execution. However, in some occasions users may want some of the models used in the task declaration to be optional. These models would be part of the task declaration but would not prevent its execution if they were inaccessible e.g., if the model file did not exist, or if they were not produced. While at the moment ModelFlow does not support optional models, these could be supported in the future.

Alternative loading strategies. ModelFlow uses a loading strategy that aims to reduce the invocation of model loading and disposal procedures. To do this, ModelFlow loads a model when it is first used in the workflow and disposes it when it is deemed no longer useful. However, while some models are expensive to load, others can be costly to keep loaded in memory. As such, we recognise that, depending on the type of workflow, other strategies may be more suitable such as (a) loading all models at the beginning of the workflow and disposing them at the end or (b) loading and disposing them before and after each task, respectively. ModelFlow could be extended to support more model loading and disposal strategies that allow different types of workflows to deliver optimal performance. Similarly ModelFlow could allow users to decide which loading strategy is more appropriate for their workflows and even support the combination of strategies for different models within the same workflow.

Error handling. Currently ModelFlow offers limited support for error handling. The current behaviour is to halt the workflow execution when an exception occurs and to update the execution trace with tasks that were successful, indicating which task failed. For exceptions such as a task or model declaration misconfiguration identified at runtime, ModelFlow could be extended to add transactional support to workflow resources to undo changes when errors are detected. Transaction support should allow subsequent executions to start from a consistent state and continue with the rest of the workflow when the error has been fixed. For exceptions which are expected from a given workflow, ModelFlow could be extended to allow tasks (or nested workflows) to be invoked when a certain exception is thrown, possibly by a specific task. This would effectively allow for alternative exception-handling execution paths. Finally, for unexpected exceptions ModelFlow can be integrated with CI tools to notify about the issues so that the workflow structure or its resources can be repaired and then re-executed.

8 Bibliography

[1] Itemis AG. Yakindu Traceability. [Online], January 2017. Available: https://www.itemis.com/en/yakindu/traceability/. [Accessed 07 July 2018.

[2] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. Efficiently querying large-scale heterogeneous models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–5, Virtual Event Canada, October 2020. ACM. ISBN 978-1-4503-8135-2. doi: 10.1145/3417990.3420207.

[3] Camilo Alvarez and Rubby Casallas. MTC Flow: a tool to design, develop and deploy model transformation chains. In *Proceedings of the workshop on ACadeMics Tooling with Eclipse - ACME '13*, pages 1–9, Montpellier, France, 2013. ACM Press. ISBN 978-1-4503-2036-8. doi: 10.1145/2491279.2491286.

[4] Nicholas Annable. *A Model-Based Approach to Formal Assurance Cases*. PhD thesis, McMaster University, 2020.

[5] Caroline M Ashworth. Structured systems analysis and design method (SSADM). *Information and Software Technology*, 30(3):153–163, April 1988. ISSN 09505849. doi: 10.1016/0950-5849(88)90062-6.

[6] D. E. Avison. MERISE: A European methodology for developing information systems. *European Journal of Information Systems*, 1(3):183–191, August 1991. ISSN 0960-085X, 1476-9344. doi: 10.1057/ejis.1991.33.

[7] Mikaël Barbero, Frédéric Jouault, and Jean Bézivin. Model Driven Management of Complex Systems: Implementing the Macroscope's Vision. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2008)*, pages 277–286, Belfast, Northern Ireland, March 2008. IEEE. ISBN 978-0-7695-3141-0. doi: 10.1109/ECBS.2008.42.

[8] Konstantinos Barmpis. *Towards Scalable Model Indexing*. PhD thesis, University of York, 2016. Available: http://etheses.whiterose.ac.uk/14376/.

[9] Konstantinos Barmpis and Dimitrios S. Kolovos. Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models. *The Journal of Object Technology*, 13(3):3:1, 2014. ISSN 1660-1769. doi: 10.5381/jot.2014.13.3.a3.

[10] Konstantinos Barmpis and Dimitris Kolovos. Hawk: towards a scalable model indexing architecture. In *Proceedings of the Workshop on Scalability in Model Driven Engineering - BigMDE '13*, pages 1–9, Budapest, Hungary, 2013. ACM Press. ISBN 978-1-4503-2165-5. doi: 10.1145/2487766.2487771.

[11] Marc Bender, Karen Laurin, Mark Lawford, Vera Pantelic, Alexandre Korobkine, Jeff Ong, Bennett Mackenzie, Monika Bialy, and Steven Postma. Signature required: Making Simulink data flow and interfaces explicit. *Science of Computer Programming*, 113:29–50, December 2015. ISSN 01676423. doi: 10.1016/j.scico.2015.07.005.

[12] Amine Benelallam, Abel Gómez, Massimo Tisi, and Jordi Cabot. Distributed model-to-model transformation with ATL on MapReduce. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 37–48, Pittsburgh PA USA, October 2015. ACM. ISBN 978-1-4503-3686-4. doi: 10.1145/2814251.2814258.

[13] Amine Benelallam, Massimo Tisi, Jesús Sánchez Cuadrado, Juan de Lara, and Jordi Cabot. Efficient model partitioning for distributed model transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 226–238, Amsterdam Netherlands, October 2016. ACM. ISBN 978-1-4503-4447-0. doi: 10.1145/2997364.2997385.

[14] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental Evaluation of Model Queries over EMF Models. In *Proceedings of Model Driven Engineering Languages and Systems (MODELS)*, pages 76–90. Springer, Berlin, Heidelberg, 2010. ISBN 3642161448. doi: 10.1007/978-3-642-16145-2_6.

[15] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A Reactive Model Transformation Platform. In *Theory and Practice of Model Transformations*, volume 9152, pages 101–110. Springer International Publishing, Cham, 2015. ISBN 978-3-319-21154-1 978-3-319-21155-8. doi: 10.1007/978-3-319-21155-8_8. Series Title: Lecture Notes in Computer Science.

[16] J. Bezivin and O. Gerbe. Towards a precise definition of the OMG/MDA framework. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280, San Diego, CA, USA, 2001. IEEE Comput. Soc. ISBN 978-0-7695-1426-0. doi: 10.1109/ASE.2001.989813.

[17] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications,(2004)*, Vancouver, Canada, October 2004. Available: https://hal.archives-ouvertes.fr/hal-01222947.

[18] Jonas Boner, Dave Farley, Roland Kuhn, and Martin Thompson. Reactive Manifesto. [Online], September 2014. Available: https://www.reactivemanifesto.org. [Accessed 20 May 2021].

[19] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-Driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, September 2012. ISSN 2328-3319, 2328-3327. doi: 10.2200/S00441ED1V01Y201208SWE001.

[20] Loli Burgueno, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. Parallel in-place model transformations with LinTra. In *Proceedings of the 3rd Workshop on Scalable Model Driven Engineering*, 2015. Available: http://ceur-ws.org/Vol-1406/paper6.pdf.

[21] Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. On the concurrent execution of model transformations with Linda. In *Proceedings of the Workshop on Scalability in Model Driven Engineering - BigMDE '13*, pages 1–10, Budapest, Hungary, 2013. ACM Press. ISBN 978-1-4503-2165-5. doi: 10.1145/2487766.2487770.

[22] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In *Software Technologies: Applications and Foundations*, volume 10748, pages 146–153. Springer International Publishing, Cham, 2018. ISBN 978-3-319-74729-3 978-3-319-74730-9. doi: 10.1007/978-3-319-74730-9_12. Series Title: Lecture Notes in Computer Science.

[23] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the Large and Modeling in the Small. In *Model Driven Architecture*, volume 3599, pages 33–46. Springer Berlin Heidelberg, Berlin,

Heidelberg, 2005. ISBN 978-3-540-28240-2 978-3-540-31819-4. doi: 10.1007/11538097_3. Series Title: Lecture Notes in Computer Science.

[24] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Fréderic Jouault, Dimitrios Kolovos, Ivan Kurtev, and Richard F. Paige. A Canonical Scheme for Model Composition. In *Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations and Applications*, volume 4066, pages 346–360, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-35909-8 978-3-540-35910-4. doi: 10.1007/11787044_26. Series Title: Lecture Notes in Computer Science.

[25] Jordi Cabot and Ernest Teniente. Incremental Evaluation of OCL Constraints. In *Proceedings of Advanced Information Systems Engineering*, pages 81–95. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-34653-1. doi: 10.1007/11767138_7.

[26] 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, June 2000. ISSN 13891286. doi: 10.1016/S1389-1286(00)00040-2.

[27] Bassim Chabibi, Abdelilah Douche, Adil Anwar, and Mahmoud Nassar. Integrating SysML with Simulation Environments (Simulink) by Model Transformation Approach. In *Proceedings of the 2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 148–150, Paris, France, June 2016. IEEE. ISBN 978-1-5090-1663-1. doi: 10.1109/WETICE.2016.39.

[28] J. Cleland-Huang, G. Zemont, and W. Lukasik. A heterogeneous solution for improving the return on investment of requirements traceability. In *Proceedings of the 12th IEEE International Requirements Engineering Conference, 2004.*, pages 214–223, Kyoto, Japan, 2004. IEEE. ISBN 978-0-7695-2174-9. doi: 10.1109/ICRE.2004.1335680.

[29] Jane Cleland-Huang, Orlena C. Z. Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering - FOSE 2014*, pages 55–69, Hyderabad India, May 2014. ACM. ISBN 978-1-4503-2865-4. doi: 10.1145/2593882.2593891.

[30] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006. ISSN 0018-8670. doi: 10.1147/sj.453.0621.

[31] Åsa G. Dahlstedt and Anne Persson. Requirements Interdependencies: State of the Art and Future Challenges. In *Engineering and Managing Software Requirements*, pages 95–116. Springer-Verlag, Berlin/Heidelberg, 2005. ISBN 978-3-540-25043-2. doi: 10.1007/3-540-28244-0_5.

[32] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. NeoEMF: A multi-database model persistence framework for very large models. *Science of Computer Programming*, 149:9–14, December 2017. ISSN 01676423. doi: 10.1016/j.scico.2017.08.002.

[33] Dassautl Systems. Reqtify. [Online], December 2019. Available: https://www.3ds.com/products-services/catia/products/reqtify/. [Accessed 09 September 2020].

[34] Adam L. Davis. Gradle. In *Learning Groovy*, pages 65–70. Apress, Berkeley, CA, 2016. ISBN 978-1-4842-2116-7 978-1-4842-2117-4. doi: 10.1007/978-1-4842-2117-4_12.

[35] Marco Di Natale, Francesco Chirico, Andrea Sindico, and Alberto Sangiovanni-Vincentelli. An MDA Approach for the Generation of Communication Adapters Integrating SW and FW Components from Simulink. In *Model-Driven Engineering Languages and Systems*, volume 8767, pages 353–369. Springer International Publishing, Cham, 2014. ISBN 978-3-319-11652-5 978-3-319-11653-2. doi: 10.1007/978-3-319-11653-2_22. Series Title: Lecture Notes in Computer Science.

[36] Marco Di Natale, David Perillo, Francesco Chirico, Andrea Sindico, and Alberto Sangiovanni-Vincentelli. A Model-based approach for the synthesis of software to firmware adapters for use with automatically generated components. *Software & Systems Modeling*, 17(1):11–33, February 2018. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-016-0534-0.

[37] Alessio Di Sandro, Rick Salay, Michalis Famelis, Sahar Kokaly, and Marsha Chechik. MMINT: A graphical tool for interactive model management. In *Proceedings of the 2015 Model Driven Engineering Languages and Systems (MODELS) Demo and Poster Session*, 2015. Available: http://ceur-ws.org/Vol-1554/PD_MoDELS_2015_paper_6.pdf.

[38] Marcos Didonet, Del Fabro, Jean Bezivin, and Patrick Valduriez. Weaving Models with the Eclipse AMW plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe*, 2006.

[39] Zinovy Diskin. Model Synchronization: Mappings, Tiles, and Categories. In *Generative and Transformational Techniques in Software En-*

gineering III, volume 6491, pages 92–165. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-18022-4 978-3-642-18023-1. doi: 10.1007/978-3-642-18023-1_3. Series Title: Lecture Notes in Computer Science.

[40] Zinovy Diskin and Tom Maibaum. Category Theory and Model-Driven Engineering: From Formal Semantics to Design Patterns and Beyond. *Electronic Proceedings in Theoretical Computer Science*, 93:1–21, August 2012. ISSN 2075-2180. doi: 10.4204/EPTCS.93.1.

[41] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In *Model-Driven Engineering Languages and Systems*, volume 6981, pages 304–318. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-24484-1 978-3-642-24485-8. doi: 10.1007/978-3-642-24485-8_22. Series Title: Lecture Notes in Computer Science.

[42] Zinovy Diskin, Sahar Kokaly, and Tom Maibaum. Mapping-Aware Megamodeling: Design Patterns and Laws. In *Software Language Engineering*, volume 8225, pages 322–343. Springer International Publishing, Cham, 2013. ISBN 978-3-319-02653-4 978-3-319-02654-1. doi: 10.1007/978-3-319-02654-1_18. Series Title: Lecture Notes in Computer Science.

[43] Zinovy Diskin, Nicholas Annable, Alan Wassyng, and Mark Lawford. Assurance via Workflow+ Modelling and Conformance (an extended version). Technical report, McMaster Centre for Software Certification, 2019.

[44] Juri Di Rocco, Davide Di Ruscio, Johannes Härtel, Ludovico Iovino, Ralf Lämmel, and Alfonso Pierantonio. Understanding MDE projects: megamodels to the rescue for architecture recovery. *Software & Systems Modeling*, 19(2):401–423, March 2020. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-019-00748-7.

[45] Nikolaos Drivalos, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. Engineering a DSL for Software Traceability. In *Software Language Engineering*, volume 5452, pages 151–167. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00433-9 978-3-642-00434-6. doi: 10.1007/978-3-642-00434-6_10. Series Title: Lecture Notes in Computer Science.

[46] Marlon Dumas and Arthur H. M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *Proceedings of the 4th Interna-*

tional Conference on the Unified Modeling Language, pages 76–90, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45441-0. doi: 10.1007/3-540-45441-1_7.

[47] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Addison-Wesley signature series. Addison-Wesley, Upper Saddle River, NJ, 2007. ISBN 978-0-321-33638-5.

[48] Eclipse Lyo. Lyo Simulink Adapter. [Online], August 2014. Available: https://wiki.eclipse.org/Lyo/Simulink. [Accessed 07 July 2018].

[49] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805829.

[50] Jad El-Khoury, Cecilia Ekelin, and Christian Ekholm. Supporting the Linked Data Approach to Maintain Coherence Across Rich EMF Models. In *Modelling Foundations and Applications*, volume 9764, pages 36–47. Springer International Publishing, Cham, 2016. ISBN 978-3-319-42060-8 978-3-319-42061-5. doi: 10.1007/978-3-319-42061-5_3. Series Title: Lecture Notes in Computer Science.

[51] Georg Engel, Ajay Sathya Chakkaravarthy, and Gerald Schweiger. Co-simulation Between Trnsys and Simulink Based on Type155. In *Software Engineering and Formal Methods*, volume 10729, pages 315–329. Springer International Publishing, Cham, 2018. ISBN 978-3-319-74780-4 978-3-319-74781-1. doi: 10.1007/978-3-319-74781-1_22. Series Title: Lecture Notes in Computer Science.

[52] Eclipse Epsilon. Documentation. [Online], March 2020. Available: https://www.eclipse.org/epsilon/doc/. [Accessed 23 July 2021.

[53] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A sound and optimal incremental build system with dynamic dependencies. *ACM SIGPLAN Notices*, 50(10):89–106, December 2015. ISSN 0362-1340, 1558-1160. doi: 10.1145/2858965.2814316.

[54] Predrag Filipovikj, Guillermo Rodriguez-Navas, and Cristina Seceleanu. Bounded invariance checking of simulink models. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 2168–2177, Limassol Cyprus, April 2019. ACM. ISBN 978-1-4503-5933-7. doi: 10.1145/3297280.3297493.

[55] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007. ISSN 0164-0925, 1558-4593. doi: 10.1145/1232420.1232424.

[56] The Apache Software Foundation. Ant. [Online], July 2000. Available: https://ant.apache.org. [Accessed 23 July 2021].

[57] The Apache Software Foundation. Maven - Introduction to the Lifecycle. [Online], July 2021. Available: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html. [Accessed 23 July 2021].

[58] Mārtiņš Francis, Dimitrios S. Kolovos, Nicholas Matragkas, and Richard F. Paige. Adding Spreadsheets to the MDE Toolkit. In *Model-Driven Engineering Languages and Systems*, volume 8107, pages 35–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-41532-6 978-3-642-41533-3. doi: 10.1007/978-3-642-41533-3_3. Series Title: Lecture Notes in Computer Science.

[59] Jokin Garcia and Jordi Cabot. Stepwise Adoption of Continuous Delivery in Model-Driven Engineering. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, volume 11350, pages 19–32. Springer International Publishing, Cham, 2019. ISBN 978-3-030-06018-3 978-3-030-06019-0. doi: 10.1007/978-3-030-06019-0_2. Series Title: Lecture Notes in Computer Science.

[60] Vicente García-Díaz, Jordán Pascual Espada, Edward Rolando Núñez-Valdéz, B. Cristina Pelayo García-Bustelo, and Juan Manuel Cueva Lovelle. Combining the continuous integration practice and the model-driven engineering approach. *Computing and Informatics*, 35(2):299–337, 2016.

[61] Antonio Garcia-Dominguez, Konstantinos Barmpis, Dimitrios S Kolovos, Marcos Aurelio Almeida da Silva, Antonin Abherve, and Alessandra Bagnato. Integration of a graph-based model indexer in commercial modelling tools. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 340–350, Saint-Malo France, October 2016. ACM. ISBN 978-1-4503-4321-3. doi: 10.1145/2976767.2976809.

[62] Antonio Garcia-Dominguez, Konstantinos Barmpis, Dimitrios S. Kolovos, Ran Wei, and Richard F. Paige. Stress-testing remote model querying

APIs for relational and graph-based stores. *Software & Systems Modeling*, 18(2):1047–1075, April 2019. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-017-0606-9.

[63] Google. BigQuery. [Online], May 2011. Available: https://cloud.google.com/bigquery/. [Accessed 07 July 2018].

[64] Google. Bazel. [Online], March 2015. Available: https://bazel.build. [Accessed 23 July 2021].

[65] O. Gotel, J. Cleland-Huang, J. Huffman Hayes, A. Zisman, A. Egyed, P. Grunbacher, and G. Antoniol. The quest for Ubiquity: A roadmap for software and systems traceability research. In *Proceedings of the 2012 20th IEEE International Requirements Engineering Conference (RE)*, pages 71–80, Chicago, IL, USA, September 2012. IEEE. ISBN 978-1-4673-2785-5 978-1-4673-2783-1 978-1-4673-2784-8. doi: 10.1109/RE.2012.6345841.

[66] Orlena Gotel and Patrick Mäder. Acquiring Tool Support for Traceability. In *Software and Systems Traceability*, pages 43–68. Springer London, London, 2012. ISBN 978-1-4471-2238-8 978-1-4471-2239-5. doi: 10.1007/978-1-4471-2239-5_3.

[67] Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mäder. Traceability Fundamentals. In *Software and Systems Traceability*, pages 3–22. Springer London, London, 2012. ISBN 978-1-4471-2238-8 978-1-4471-2239-5. doi: 10.1007/978-1-4471-2239-5_1.

[68] Victor Guana. ChainTracker - Model Transformation Analysis. [Online], November 2017. Available: https://guana.github.io/chaintracker. [Accessed 23 July 2021].

[69] Victor Guana and Eleni Stroulia. ChainTracker, a Model-Transformation Trace Analysis Tool for Code-Generation Environments. In *Theory and Practice of Model Transformations*, volume 8568, pages 146–153. Springer International Publishing, Cham, 2014. ISBN 978-3-319-08788-7 978-3-319-08789-4. doi: 10.1007/978-3-319-08789-4_11. Series Title: Lecture Notes in Computer Science.

[70] Victor Guana and Eleni Stroulia. End-to-end model-transformation comprehension through fine-grained traceability information. *Software & Systems Modeling*, 18(2):1305–1344, April 2019. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-017-0602-0.

[71] Victor Guana, Kelsey Gaboriau, and Eleni Stroulia. ChainTracker: Towards a Comprehensive Tool for Building Code-Generation Environments. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 613–616, Victoria, BC, Canada, September 2014. IEEE. ISBN 978-1-4799-6146-7. doi: 10.1109/ICSME.2014.108.

[72] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. Inter-modelling: From Theory to Practice. In *Model-Driven Engineering Languages and Systems*, volume 6394, pages 376–391. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-16144-5 978-3-642-16145-2. doi: 10.1007/978-3-642-16145-2_26. Series Title: Lecture Notes in Computer Science.

[73] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental Model Transformation for the Evolution of Model-Driven Systems. In *Model-Driven Engineering Languages and Systems*, volume 4199, pages 321–335. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-45772-5 978-3-540-45773-2. doi: 10.1007/11880240_23. Series Title: Lecture Notes in Computer Science.

[74] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *Model Driven Architecture - Foundations and Applications*, volume 5562, pages 114–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-02673-7 978-3-642-02674-4. doi: 10.1007/978-3-642-02674-4_9. Series Title: Lecture Notes in Computer Science.

[75] Georg Hinkel, Robert Heinrich, and Ralf Reussner. An extensible approach to implicit incremental model analyses. *Software and Systems Modeling*, 18(5):3151–3187, 2019. ISSN 16191374. doi: 10.1007/s10270-019-00719-y.

[76] Fazilat Hojaji, Tanja Mayerhofer, Bahman Zamani, Abdelwahab Hamou-Lhadj, and Erwan Bousse. Model execution tracing: a systematic mapping study. *Software & Systems Modeling*, 18(6):3461–3485, December 2019. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-019-00724-1.

[77] Boris Holzer. Snapshots and change reports for requirements traceability data. [Online], August 2017. Available: https://blogs.itemis.com/en/snapshots-and-change-reports-for-requirements-traceability-data. [Accessed 07 July 2018.

[78] Akos Horvath, Istvan Rath, and Rodrigo Rizzi Starr. Massif - the love child of Matlab Simulink and Eclipse. [Online],

November 2015. Available: https://www.slideshare.net/kosHorvth2/massif-the-love-child-of-matlab-simulink-and-eclipse. [Accessed 15 June 2018].

[79] Bryan Hunt. MongoEMF. [Online], March 2014. Available: https://github.com/BryanHunt/mongo-emf. [Accessed 23 July 2021].

[80] IBM. IBM Rational DOORS. [Online], June 2021. Available: https://www.ibm.com/uk-en/products/requirements-management/details.

[81] IBM. IBM Knowledge Center - Extending Rational DOORS by using OSLC services. [Online], June 2021. Available: https://www.ibm.com/docs/en/ermd/9.7.2?topic=function-extending-doors-by-using-oslc-services.

[82] IEEE. IEEE 1076-2008: VHDL Language Reference Manual. Standard, Institute of Electrical and Electronics Engineers, 2008.

[83] ISO. ISO/IEC 12207:2008: Systems and Software Engineering - Software Life Cycle Processes. Standard, International Organization for Standardization, 2008.

[84] ISO. ISO 26262-1:2018: Road vehicles - Functional safety. Standard, International Organization for Standardization, 2018.

[85] Sorour Jahanbin, Dimitris Kolovos, and Simos Gerasimou. Intelligent run-time partitioning of low-code system models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–5, Virtual Event Canada, October 2020. ACM. ISBN 978-1-4503-8135-2. doi: 10.1145/3417990.3420198.

[86] Frédéric Jouault and Massimo Tisi. Towards Incremental Execution of ATL Transformations. In *Proceedings of the Third International Conference on Theory and Practice of Model Transformations*, pages 123–137, Malaga, 2010. Springer-Verlag. ISBN 3642136877. doi: 10.1007/978-3-642-13688-7_9.

[87] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like transformation language. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA '06*, page 719, Portland, Oregon, USA, 2006. ACM Press. ISBN 978-1-59593-491-8. doi: 10.1145/1176617.1176691.

[88] Andrew Kannenberg and Hossein Saiedian. Why Software Requirements Traceability Remains a Challenge. *CrossTalk The Journal of Defense Software Engineering*, 22(5):14–19, 2009.

[89] Vikash Katta and Tor Stålhane. A Conceptual Model of Traceability for Safety Systems. In *Proceedings of the Complex Systems Design & Management Conference*, 2011.

[90] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation.* Wiley-Interscience : IEEE Computer Society, Hoboken, N.J, 2008. ISBN 978-0-470-03666-2.

[91] Timothy Patrick Kelly. *Arguing Safety – A Systematic Approach to Managing Safety Cases.* PhD thesis, University of York, 1998.

[92] Wolfgang Kling, Frédéric Jouault, Dennis Wagelaar, Marco Brambilla, and Jordi Cabot. MoScript: A DSL for Querying and Manipulating Model Repositories. In *Software Language Engineering*, volume 6940, pages 180–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-28829-6 978-3-642-28830-2. doi: 10.1007/978-3-642-28830-2_10. Series Title: Lecture Notes in Computer Science.

[93] Knowledge Based Systems Inc. IDEF: Integrated Definition Methods. [Online], 1980. Available: https://www.idef.com. [Accessed 23 July 2021].

[94] Sahar Kokaly. Towards a Structured Workflow Language for Model Management. In *Proceedings of the Doctoral Symposium at MODELS'14*, 2014. Available: http://ceur-ws.org/Vol-1321/dsmodels14_3.pdf.

[95] Sahar Kokaly, Rick Salay, Valentin Cassano, Tom Maibaum, and Marsha Chechik. A model management approach for assurance case reuse due to system evolution. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 196–206, Saint-Malo France, October 2016. ACM. ISBN 978-1-4503-4321-3. doi: 10.1145/2976767.2976792.

[96] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Merging Models with the Epsilon Merging Language (EML). In *Model-Driven Engineering Languages and Systems*, volume 4199, pages 215–229. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-45772-5 978-3-540-45773-2. doi: 10.1007/11880240_16. Series Title: Lecture Notes in Computer Science.

[97] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture - Foundations*

and Applications, volume 4066, pages 128–142. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-35909-8 978-3-540-35910-4. doi: 10.1007/11787044_11. Series Title: Lecture Notes in Computer Science.

[98] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proceedings of the 2006 international workshop on Global integrated model management - GaMMa '06*, page 13, Shanghai, China, 2006. ACM Press. ISBN 978-1-59593-410-9. doi: 10.1145/1138304.1138308.

[99] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations*, volume 5063, pages 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-69926-2 978-3-540-69927-9. doi: 10.1007/978-3-540-69927-9_4. Series Title: Lecture Notes in Computer Science.

[100] Dimitrios S Kolovos, Richard F Paige, and Fiona A C Polack. A Framework for Composing Modular and Interoperable Model Management Tasks. In *Model-Driven Tool and Process Integration Workshop*, 2008.

[101] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Scalability: The holy grail of model driven engineering. In *First International Workshop on Challenges in Model Driven Software Engineering*, 2008.

[102] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6, Vancouver, BC, Canada, May 2009. IEEE. ISBN 978-1-4244-3714-6. doi: 10.1109/CVSM.2009.5071714.

[103] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Rigorous Methods for Software Construction and Analysis*, volume 5115, pages 204–218. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-11446-5 978-3-642-11447-2. doi: 10.1007/978-3-642-11447-2_13. Series Title: Lecture Notes in Computer Science.

[104] Dimitrios S. Kolovos, Massimo Tisi, Jordi Cabot, Louis M. Rose, Nicholas Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, and Dániel Varró. A research roadmap

towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering - BigMDE '13*, pages 1–10, Budapest, Hungary, 2013. ACM Press. ISBN 978-1-4503-2165-5. doi: 10.1145/2487766.2487768.

[105] Dimitrios S. Kolovos, Ran Wei, and Konstantinos Barmpis. An approach for efficient querying of large relational datasets with OCL-based languages. In *Proceedings of the Workshop on Extreme Modeling*, 2013. Available: http://ceur-ws.org/Vol-1089/6.pdf.

[106] Dimitrios S. Kolovos, Antonio García-Domínguez, Louis M. Rose, and Richard F. Paige. Eugenia: towards disciplined and automated development of GMF-based graphical model editors. *Software & Systems Modeling*, 16(1):229–255, February 2017. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-015-0455-3.

[107] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. Scalable incremental building with dynamic task dependencies. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 76–86, Montpellier France, September 2018. ACM. ISBN 978-1-4503-5937-5. doi: 10.1145/3238147.3238196.

[108] Felix Kossak, Christa Illibauer, Verena Geist, Jan Kubovy, Christine Natschläger, Thomas Ziebermayr, Theodorich Kopetzky, Bernhard Freudenthaler, and Klaus-Dieter Schewe. *A Rigorous Semantics for BPMN 2.0 Process Diagrams*. Springer International Publishing, Cham, 2014. ISBN 978-3-319-09930-9 978-3-319-09931-6. doi: 10.1007/978-3-319-09931-6.

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

[110] Sina Madani, Dimitrios S. Kolovos, and Richard F. Paige. Parallel Model Validation with Epsilon. In *Modelling Foundations and Applications*, volume 10890, pages 115–131. Springer International Publishing, Cham, 2018. ISBN 978-3-319-92996-5 978-3-319-92997-2. doi: 10.1007/978-3-319-92997-2_8. Series Title: Lecture Notes in Computer Science.

[111] Sina Madani, Dimitris Kolovos, and Richard F. Paige. Towards Optimisation of Model Queries: A Parallel Execution Approach. *The Journal of Object Technology*, 18(2):3:1, 2019. ISSN 1660-1769. doi: 10.5381/jot.2019.18.2.a3.

[112] Salome Maro and Jan-Philipp Steghofer. Capra: A Configurable and Extendable Traceability Management Tool. In *Proceedings of the 2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 407–408, Beijing, China, September 2016. IEEE. ISBN 978-1-5090-4121-3. doi: 10.1109/RE.2016.19.

[113] Salome Maro, Anthony Anjorin, Rebekka Wohlrab, and Jan-Philipp Steghöfer. Traceability maintenance: factors and guidelines. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 414–425, Singapore Singapore, August 2016. ACM. ISBN 978-1-4503-3845-5. doi: 10.1145/2970276.2970314.

[114] Salome Maro, Jan-Philipp Steghöfer, and Miroslaw Staron. Software traceability in the automotive domain: Challenges and solutions. *Journal of Systems and Software*, 141:85–110, July 2018. ISSN 01641212. doi: 10.1016/j.jss.2018.03.060.

[115] Salvador Martínez, Massimo Tisi, and Rémi Douence. Reactive model transformation with ATL. *Science of Computer Programming*, 136:1–16, March 2017. ISSN 01676423. doi: 10.1016/j.scico.2016.08.006.

[116] MathWorks. MATLAB Simulink. [Online], 1984. Available: https://www.mathworks.com/products/simulink.html. [Accessed 07 July 2018].

[117] MathWorks. Create a Simple Model. [Online], 2018. Available: https://uk.mathworks.com/help/simulink/gs/create-a-simple-model.html. [Accessed 07 July 2018].

[118] Mathworks. MATLAB Stateflow. [Online], March 2018. Available: https://uk.mathworks.com/products/stateflow.html. [Accessed 07 July 2018].

[119] Mathworks. Large-Scale Modeling. [Online], 2020. Available: https://uk.mathworks.com/help/simulink/large-scale-modeling.html. [Accessed 07 July 2018].

[120] Mathworks. Best Practices and Guidelines for ReqIF Round Trip Workflows. [Online], October 2020. Available: https://uk.mathworks.com/help/slrequirements/ug/best-practices-for-reqif-roundtrip-workflows.html. [Accessed 16 October 2018].

[121] MathWorks. Simulink Traceability. [Online], October 2020. Available: https://uk.mathworks.com/discovery/requirements-traceability.html. [Accessed 16 October 2020].

[122] Sergey Melnik. *Generic Model Management : Concepts and Algorithms*. PhD thesis, University of Leipzig, 2004.

[123] Marcus Mikulcak, Paula Herber, Thomas Göthel, and Sabine Glesner. Information Flow Analysis of Combined Simulink/Stateflow Models. *Information Technology And Control*, 48(2):299–315, June 2019. ISSN 2335-884X, 1392-124X. doi: 10.5755/j01.itc.48.2.21759.

[124] Modeliosoft. Modelio. [Online], September 2020. Available: https://www.modelio.org.

[125] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. Build systems à la carte: Theory and practice. *Journal of Functional Programming*, 30: e11, 2020. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796820000088.

[126] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989. ISSN 00189219. doi: 10.1109/5.24143.

[127] Benjamin Muschko. *Gradle in action*. Manning, Shelter Island, NY, 2014. ISBN 978-1-61729-130-2.

[128] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty H. C. Cheng, Philippe Collet, Benoit Combemale, Robert B. France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum, and Jon Whittle. The Relevance of Model-Driven Engineering Thirty Years from Now. In *Model-Driven Engineering Languages and Systems*, volume 8767, pages 183–200. Springer International Publishing, Cham, 2014. ISBN 978-3-319-11652-5 978-3-319-11653-2. doi: 10.1007/978-3-319-11653-2_12. Series Title: Lecture Notes in Computer Science.

[129] Nasser Mustafa and Yvan Labiche. Modeling Traceability for Heterogeneous Systems:. In *Proceedings of the 10th International Conference on Software Engineering and Applications*, pages 358–366, Colmar, Alsace, France, 2015. SCITEPRESS - Science and Technology Publications. ISBN 978-989-758-114-4. doi: 10.5220/0005520303580366.

[130] Nasser Mustafa and Yvan Labiche. Towards Traceability Modeling for the Engineering of Heterogeneous Systems:. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pages 321–328, ESEO, Angers, Loire Valley, France, 2015. SCITEPRESS - Science and Technology Publications. ISBN 978-989-758-083-3. doi: 10.5220/0005246103210328.

[131] Tomasz Nurkiewicz and Ben Christensen. *Reactive programming with RxJava: creating asynchronous, event-based applications*. O'Reilly Media, Inc, Sebastopol, CA, first edition, 2016. ISBN 978-1-4919-3165-3.

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

[133] Babajide Ogunyomi, Louis M. Rose, and Dimitrios S. Kolovos. On the Use of Signatures for Source Incremental Model-to-text Transformation. In *Model-Driven Engineering Languages and Systems*, volume 8767, pages 84–98. Springer International Publishing, Cham, 2014. ISBN 978-3-319-11652-5 978-3-319-11653-2. doi: 10.1007/978-3-319-11653-2_6. Series Title: Lecture Notes in Computer Science.

[134] Babajide Ogunyomi, Louis M. Rose, and Dimitrios S. Kolovos. Property Access Traces for Source Incremental Model-to-Text Transformation. In *Modelling Foundations and Applications*, volume 9153, pages 187–202. Springer International Publishing, Cham, 2015. ISBN 978-3-319-21150-3 978-3-319-21151-0. doi: 10.1007/978-3-319-21151-0_13. Series Title: Lecture Notes in Computer Science.

[135] Babajide Ogunyomi, Louis M. Rose, and Dimitrios S. Kolovos. Incremental execution of model-to-text transformations using property access traces. *Software and Systems Modeling*, 18(1):367–383, 2019. ISSN 16191374. doi: 10.1007/s10270-018-0666-5.

[136] OMG. Business Process Model and Notation (BPMN) version 2.0. Specification, Object Management Group, 2012. Available: https://www.omg.org/spec/BPMN/2.0/.

[137] OMG. Object Constraint Language (OCL). Specification, The Object Management Group, 2014. Available: https://www.omg.org/spec/OCL/.

[138] OMG. Model Driven Architecture (MDA) revision 2.0. Specification, Object Management Group, 2014. Available: https://www.omg.org/mda/.

[139] OMG. XML Metadata Interchange (XMI). Specification, Object Management Group, 2014. Available: https://www.omg.org/spec/XMI.

[140] OMG. Meta Object Facility (MOF) Core. Specification, The Object Management Group, 2016. Available: https://www.omg.org/spec/MOF.

[141] OMG. Query/View/Transformation (QVT). Specification, The Object Management Group, 2016. Available: https://www.omg.org/spec/QVT.

[142] OMG. Unified Modeling Language (UML). Specification, Object Management Group, 2017. Available: https://www.omg.org/spec/UML.

[143] OMG. Systems Modeling Language (SYSML) v.1.6. Specification, Object Management Group, 2019. Available: https://www.omg.org/spec/SysML.

[144] OASIS Open. Open Services for Lifecycle Collaboration (OSLC). [Online], June 2008. Available: https://open-services.net. [Accessed 07 July 2018].

[145] Oracle. Java Microbenchmark Harness (JMH). [Online], May 2018. Available: https://openjdk.java.net/projects/code-tools/jmh/. [Accessed 07 July 2018].

[146] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. A repository for scalable model management. *Software & Systems Modeling*, 14(1):219–239, February 2015. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-013-0326-8.

[147] Richard F. Paige, Gøran K. Olsen, Dimitrios S. Kolovos, Steffen Zschaler, and Christopher Power. Building Model-Driven Engineering Traceability Classifications. In *Proceedings of the 4th ECMDA Traceability Workshop*, 2008. Available: https://core.ac.uk/download/pdf/74235066.pdf.

[148] Richard F. Paige, Nikolaos Drivalos, Dimitrios S. Kolovos, Kiran J. Fernandes, Christopher Power, Goran K. Olsen, and Steffen Zschaler. Rigorous identification and encoding of trace-links in model-driven engineering. *Software & Systems Modeling*, 10(4):469–487, October 2011. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-010-0158-8.

[149] Richard F. Paige, Nicholas Matragkas, and Louis M. Rose. Evolving models in Model-Driven Engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272–280, January 2016. ISSN 01641212. doi: 10.1016/j.jss.2015.08.047.

[150] Vera Pantelic., Steven Postma., Mark Lawford., Alexandre Korobkine., Bennett Mackenzie., Jeff Ong., and Marc Bender. A Toolset for Simulink - Improving Software Engineering Practices in Development with Simulink:. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pages 50–61, ESEO, Angers, Loire Valley, France, 2015. SCITEPRESS - Science and Technology Publications. ISBN 978-989-758-083-3. doi: 10.5220/0005236100500061.

[151] Vera Pantelic, Steven Postma, Mark Lawford, Monika Jaskolka, Bennett Mackenzie, Alexandre Korobkine, Marc Bender, Jeff Ong, Gordon Marks, and Alan Wassyng. Software engineering practices and Simulink: bridging the gap. *International Journal on Software Tools for Technology Transfer*, 20(1):95–117, February 2018. ISSN 1433-2779, 1433-2787. doi: 10.1007/s10009-017-0450-9.

[152] Terence Parr. *The definitive ANTLR 4 reference*. The pragmatic programmers. The Pragmatic Bookshelf, Dallas, Texas, 2012. ISBN 978-1-934356-99-9.

[153] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, January 2001. ISSN 00985589. doi: 10.1109/32.895989.

[154] Patrick Rempel, Patrick Mäder, Tobias Kuschke, and Jane Cleland-Huang. Mind the gap: assessing the conformance of software traceability to relevant guidelines. In *Proceedings of the 36th International Conference on Software Engineering*, pages 943–954, Hyderabad India, May 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568290.

[155] Louis Rose, Esther Guerra, Juan de Lara, Anne Etien, Dimitris Kolovos, and Richard Paige. Genericity for model management operations. *Software & Systems Modeling*, 12(1):201–219, February 2013. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-011-0203-2.

[156] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. The Epsilon Generation Language. In *Model Driven Architecture - Foundations and Applications*, volume 5095, pages 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-69095-5 978-3-540-69100-6. doi: 10.1007/978-3-540-69100-6_1. Series Title: Lecture Notes in Computer Science.

[157] RTCA. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Standard, Radio Technical Commission for Aeronautics, 2011.

[158] Nicholas Charles Russell. *Foundations of Process-Aware Information Systems*. PhD thesis, Queensland University of Technology, 2007. Available: http://eprints.qut.edu.au/16592/.

[159] Nick Russell, Wil van der Aalst, and Arthur Ter Hofstede. *Workflow patterns: the definitive guide*. MIT Press, Cambridge, MA, 2015. ISBN 978-0-262-02982-7.

[160] Rick Salay, Sahar Kokaly, Marsha Chechik, and Tom Maibaum. Heterogeneous megamodel slicing for model evolution. In *Proceedings of the 10th Workshop on Models and Evolution*, 2016. Available: http://ceur-ws.org/Vol-1706/paper7.pdf.

[161] Rick Salay, Sahar Kokaly, Alessio Di Sandro, Nick L. S. Fung, and Marsha Chechik. Heterogeneous megamodel management using collection

operators. *Software & Systems Modeling*, 19(1):231–260, January 2020. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-019-00738-9.

[162] Beatriz Sanchez, Athanasios Zolotas, Horacio Hoyos Rodriguez, Dimitris Kolovos, and Richard Paige. On-the-Fly Translation and Execution of OCL-Like Queries on Simulink Models. In *Proceedings of the 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 205–215, Munich, Germany, September 2019. IEEE. ISBN 978-1-72812-536-7. doi: 10.1109/MODELS.2019.000-1.

[163] Beatriz Sanchez, Dimitris Kolovos, and Richard Paige. To build, or not to build: ModelFlow, a build solution for MDE projects. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 1–11, Virtual Event Canada, October 2020. ACM. ISBN 978-1-4503-7019-6. doi: 10.1145/3365438.3410942.

[164] Andrea Sindico, Marco Di Natale, and Gianpiero Panci. INTEGRATING SYSML WITH SIMULINK USING OPEN-SOURCE MODEL TRANSFORMATIONS:. In *Proceedings of the 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, pages 45–56, Noordwijkerhout, Netherlands, 2011. SciTePress - Science and Technology Publications. ISBN 978-989-8425-78-2. doi: 10.5220/0003593600450056.

[165] George Spanoudakis and Andrea Zisman. SOFTWARE TRACEABILITY: A ROADMAP. In *Handbook Of Software Engineering And Knowledge Engineering*, pages 395–428. WORLD SCIENTIFIC, August 2005. ISBN 978-981-256-273-9 978-981-4480-70-3. doi: 10.1142/9789812775245_0014.

[166] George Spanoudakis, Andrea Zisman, Elena Pérez-Miñana, and Paul Krause. Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105–127, July 2004. ISSN 01641212. doi: 10.1016/S0164-1212(03)00242-5.

[167] J.M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40, 1989. ISSN 02686961. doi: 10.1049/sej.1989.0006.

[168] John Spriggs. *GSN - The Goal Structuring Notation*. Springer London, London, 2012. ISBN 978-1-4471-2311-8 978-1-4471-2312-5. doi: 10.1007/978-1-4471-2312-5.

[169] Perdita Stevens. Bidirectional model transformations in QVT: semantic

issues and open questions. *Software & Systems Modeling*, 9(1):7–20, January 2010. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-008-0109-9.

[170] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling*, 17(4):1365–1393, October 2018. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-016-0571-8.

[171] M Taromirad. *A Modelling Approach to Multi-Domain Traceability*. PhD thesis, University of York, 2014. Available: http://etheses.whiterose.ac.uk/7822/.

[172] Jérémie Tatibouët, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. Formalizing Execution Semantics of UML Profiles with fUML Models. In *Model-Driven Engineering Languages and Systems*, volume 8767, pages 133–148. Springer International Publishing, Cham, 2014. ISBN 978-3-319-11652-5 978-3-319-11653-2. doi: 10.1007/978-3-319-11653-2_9. Series Title: Lecture Notes in Computer Science.

[173] The Eclipse Foundation. Epsilon. [Online], November 2012. Available: https://www.eclipse.org/epsilon/. [Accessed 23 July 2021].

[174] The Eclipse Foundation. Teneo/Hybernate. [Online], November 2012. Available: https://wiki.eclipse.org/Teneo/Hibernate. [Accessed 23 July 2021].

[175] The Eclipse Foundation. Eclipse Modeling Framework (EMF). [Online], September 2014. Available: http://www.eclipse.org/emf. [Accessed 23 July 2021].

[176] The Eclipse Foundation. Xpand. [Online], May 2016. Available: https://www.eclipse.org/modeling/m2t/?project=xpand. [Accessed 23 July 2021].

[177] The Eclipse Foundation. EMFStore. [Online], December 2018. Available: https://www.eclipse.org/emfstore/. [Accessed 23 July 2021.

[178] The Eclipse Foundation. Lyo. [Online], February 2018. Available: https://www.eclipse.org/lyo/. [Accessed 07 July 2018].

[179] The Eclipse Foundation. Acceleo. [Online], November 2019. Available: https://www.eclipse.org/acceleo/. [Accessed 23 July 2021].

[180] The Eclipse Foundation. EMFCompare. [Online], November 2020. Available: https://www.eclipse.org/emf/compare/. [Accessed 23 July 2021].

[181] The Eclipse Foundation. CDO Model Repository. [Online], March 2021. Available: https://www.eclipse.org/cdo/. [Accessed 23 July 2021].

[182] The Eclipse Foundation. Capra. [Online], May 2021. Available: https://projects.eclipse.org/projects/modeling.capra. [Accessed 23 July 2021].

[183] The Eclipse Foundation. Emfatic. [Online], March 2021. Available: https://www.eclipse.org/emfatic/. [Accessed 23 July 2021].

[184] The Eclipse Foundation. Xtext. [Online], March 2021. Available: https://www.eclipse.org/Xtext/. [Accessed 23 July 2021].

[185] Massimo Tisi, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. Lazy Execution of Model-to-Model Transformations. In *Model-Driven Engineering Languages and Systems*, volume 6981, pages 32–46. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-24484-1 978-3-642-24485-8. doi: 10.1007/978-3-642-24485-8_4. Series Title: Lecture Notes in Computer Science.

[186] Massimo Tisi, Salvador Martínez, and Hassene Choura. Parallel Execution of ATL Transformation Rules. In *Model-Driven Engineering Languages and Systems*, volume 8107, pages 656–672. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-41532-6 978-3-642-41533-3. doi: 10.1007/978-3-642-41533-3_40. Series Title: Lecture Notes in Computer Science.

[187] Massimo Tisi, Remi Douence, and Dennis Wagelaar. Lazy evaluation for OCL. In *Proceedings of the 15th International Workshop on OCL and Textual Modeling*, 2015. Available: http://ceur-ws.org/Vol-1512/paper04.pdf.

[188] Laurence Tratt. A change propagating model transformation Language. *The Journal of Object Technology*, 7(3):107, 2008. ISSN 1660-1769. doi: 10.5381/jot.2008.7.3.a3.

[189] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, June 2005. ISSN 03064379. doi: 10.1016/j.is.2004.02.002.

[190] Verocel. VeroTrace. [Online], 2021. Available: https://www.verocel.com/tools/lifecycle-management/. [Accessed 23 July 2021].

[191] Viatra. Massif: MATLAB Simulink Integration Framework for Eclipse. [Online], October 2014. Available: https://github.com/viatra/massif. [Accessed 07 July 2018].

[192] Ran Wei and Dimitrios S. Kolovos. Automated analysis, validation and suboptimal code detection in model management programs. In *Proceedings*

of the 2nd Workshop on Scalability in Model Driven Engineering, 2014. Available: http://ceur-ws.org/Vol-1206/paper_11.pdf.

[193] Ran Wei, Dimitrios S. Kolovos, Antonio Garcia-Dominguez, Konstantinos Barmpis, and Richard F. Paige. Partial loading of XMI models. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 329–339, Saint-Malo France, October 2016. ACM. ISBN 978-1-4503-4321-3. doi: 10.1145/2976767.2976787.

[194] Andrew J. Wellings. *Concurrent and real-time programming in Java*. John Wiley, Chichester, West Sussex, England ; Hoboken, NJ, 2004. ISBN 978-0-470-84437-3.

[195] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software & Systems Modeling*, 9(4):529–565, September 2010. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-009-0145-0.

[196] WMC. Workflow Management Coalition Terminology & Glossary. Technical report, Workflow Management Coalition, 1999. Available: http://www.workflowpatterns.com/documentation/documents/TC-1011_term_glossary_v3.pdf.

[197] XText. MWE 2. [Online], April 2019. Available: https://www.eclipse.org/Xtext/documentation/306_mwe2.html. [Accessed 23 July 2021].

[198] Athanasios Zolotas, Horacio Hoyos Rodriguez, Stuart Hutchesson, Beatriz Sanchez Pina, Alan Grigg, Mole Li, Dimitrios S. Kolovos, and Richard F. Paige. Bridging proprietary modelling and open-source model management tools: the case of PTC Integrity Modeller and Epsilon. *Software & Systems Modeling*, 19(1):17–38, January 2020. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-019-00732-1.