Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

Moola

A Groovy-based Model Operation Orchestration Language

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Stefan Weghofer
Matrikelnummer 0825465

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Prof. Mag. Dr. Manuel Wimmer
Mitwirkung: Alexander Bergmayr

Wien, 13.09.2017

(Unterschrift Verfasser) (Unterschrift Betreuung)

Moola

A Groovy-based Model Operation Orchestration Language

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Stefan Weghofer
Registration Number 0825465

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Prof. Mag. Dr. Manuel Wimmer
Assistance: Alexander Bergmayr

Vienna, 13.09.2017 ____________________ (Signature of Author) ____________________ (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Stefan Weghofer
Etzelstrasse 6, 8038 Zürich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

This work is dedicated to Birgit Bartmann. Without her continuous support, I would have never finished it or made it to Zurich to pursue my dream.

A big thank you goes to Alexander Bergmayr for his help and invaluable insights during the many many month it took to write this thesis. I hope I'll be able to make up for your patience and advice.

Finally, cheers to all my friends and colleagues to continuously nag me on the progress of this thesis. Although this looked like a never-to-be-finished project on multiple occasions, you kept me going and ruined not only a few peaceful weekends by reminding (and motivating) me to proceed. It turns out that good things simply may take a few months,.. or semesters,.. or years.

Abstract

A fundamental part of Model-Driven Engineering (MDE) is the use of models and operations. Models represent information of a target system on varying levels of abstraction, while operations allow performing actions on one or more models, including model validation, model transformation, model merging, etc. In recent years, more and more such operations and languages to describe them were introduced to allow MDE to be applied to a wide spectrum of use cases. Today, many advanced scenarios can be expressed by MDE and the use of new operation languages.

In every non-trivial project, multiple operations have to be executed in particular order to yield the final result. To orchestrate operations to so-called operation chain, tools and languages have been developed and included to development environments that help in defining complex operation chains and executing them whenever input models change.

In this thesis, existing tools and languages for model operation orchestration are analyzed and compared against each other. Inspiration is taken from these tools and other domains, such as Build Management and Workflow Management, to create a new tool for describing operation chains, called Moola. Based on a feature list derived from real-life use cases, Moola is designed and later implemented as domain-specific language (DSL) on top of Groovy. Finally, Moola is evaluated against use cases taken from the ARTIST project.

Kurzfassung

Ein zentraler Bestandteil der Model-getriebenen Software Entwicklung (MDE) ist der Einsatz von Modellen und Operationen. Modelle representieren Informationen eines Systems auf verschiedenen Abstraktionsniveaus, wohingegen Operationen das Ausführen von Aktionen auf Modellen erlauben. Solche Aktionen umfassen unter anderem das Validieren, Transformieren oder Zusammenfügen von Modellen. In den letzten Jahren wurden viele solcher Operationen bzw. Sprachen zur Beschreibung von Operationen veröffentlicht und erweitern somit das Spektrum, auf welches MDE angewendet werden kann.

In jedem nicht-trivialen Projekt müssen mehrere Operationen in vordefinierter Reihenfolge und unter Berücksichtigung vordefinierter Bedingungen ausgeführt werden, um das Endresult zu erhalten. Zur Erstellung solcher Orchestrierungen wurden Sprachen und Tools entwickelt und direkt in Entwicklungsumgebungen eingebettet. Dadurch können Operationsketten möglichst einfach Erstellung und wiederholt ausgeführt werden, z.B. wenn ein Modell sich ändert.

Im Rahmen dieser Arbeit wurden verschiedene existierende Ansätze zur Beschreibung von Orchestrierungen analysiert und miteinander verglichen. Basierend auf einer Feature-Liste, welche von diesen Ansätzen und Anwendungsfällen extrahiert wurde, wurde ein neuer Ansatz zur Beschreibung von Operationsketten entwickelt (Moola) und später als Domänenspezifische Sprache (DSL) in Groovy implementiert. Abschließend wurde Moola mittels Anwedungsfällen, die aus dem ARTIST-Projekt entnommen wurden, evaluiert.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 2
- 1.2 Aim of Work . . . 4
- 1.3 Methodological Approach . . . 4
- 1.4 Application Scenario . . . 7
- 1.5 Structure of the Work . . . 8

2 Background **9**
- 2.1 Model-Driven Engineering . . . 9
- 2.2 Domain-Specific Languages . . . 14
- 2.3 DSL Development in Groovy . . . 17

3 Approach **27**
- 3.1 Feature Analysis . . . 27
- 3.2 Integrating Operation Orchestration to Groovy . . . 30
- 3.3 Language Specification . . . 32
- 3.4 Relation to Activity Diagrams . . . 38

4 Implementation **41**
- 4.1 Model Types . . . 43
- 4.2 Models . . . 43
- 4.3 Operations . . . 45
- 4.4 Plug-ins . . . 47
- 4.5 From ... Include 47
- 4.6 Orchestration Code . . . 48
- 4.7 Parallel Execution . . . 49
- 4.8 Type Checking . . . 51
- 4.9 Plug-ins for Moola . . . 53
- 4.10 Moola Eclipse Plug-in . . . 54

5 Evaluation **57**
- 5.1 Feature Completeness . . . 57
- 5.2 ARTIST Scenarios . . . 59

5.3 Correctness 63
5.4 Threats to Validity 65

6 Related Work **67**

7 Conclusion & Future Work **73**

A Moola Style Guide **75**

Bibliography **79**

CHAPTER 1

Introduction

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."
— Lewis Carroll, *Alice in Wonderland*

Models are a core instrument in Software Engineering to describe static and dynamic aspects of a system. Many modeling languages and frameworks have appeared in the past and provide a big spectrum from which to choose. Undoubtedly, most developers know or have at least heard of the *Unified Modeling Language (UML)* [62], which is of high popularity, although not without criticism [2]. While modeling languages and tooling support grew more mature in recent years, developers continue to use models primarily in early development phases and as part of the documentation [22]. When strictly dividing models and code, extra effort has to be invested to keep both in sync and prevent models from getting out-of-date, or worse, inconsistent with the code base [68]. In such cases, models are often seen as additional work, as burden that need attention without giving any, or only limited, benefit in return.

Model-Driven Engineering (MDE) [38, 66] suggests a different role for models in the development cycle. Instead of using models only as upfront sketches of a system-to-be and later as diagrams in the documentation, MDE promotes models to first-class citizens of the development process. Giving them center stage comes with the need of introducing specific actions on models, so-called *model operations*. These operations include validation, comparison, execution and transformation of models, eventually enabling MDE to either directly execute models or use them as basis for code generation. To facilitate the task of working on models, a rich set of domain-specific languages (DSL) [26] appeared in the past, allowing for an expressive way to define model operations.

Using MDE or the more specific Model-Driven Software Development (MDSE) [72] in practice typically requires the development of several models and model operations. Those *modeling artifacts* are then combined to a workflow, in which the operations are executed on the models in a specific order to yield the implementation of the system. Such *operation chains*, as

these workflows are referred to in an MDSE context, bear resemblance to other workflows in Software Development, e.g.

- An operation chain resembles a *build process* [70]. Each operation represents a build step. Similar to build steps, operations may depend on each other and therefore require a certain execution order. In case of errors, a build process typically stops execution and reports the errors back to the user. This also applies to operation chains, in which unexpected errors lead to the termination of the chain. Furthermore, build processes often consist of tasks of varying nature. While the core of a build process typically is some form of compiler call, other tasks are possible (e.g. copying files, running tests, etc). The same applies to operation chains. While an operation chain mainly consist of model operations (e.g. validation, comparison, merging, transformation, etc), other operations from outside the immediate MDE world are possible (e.g. archiving files, calling external programs, collecting user input, etc).

- An operation chain can be seen as *orchestration* [84] of operations. Orchestrations describe the arrangement of components and how the control and data flows between them. Orchestrations therefore imperatively or declaratively describe how components interact. In this analogy, operations take the place of components, each providing a single, well-defined task on a set of input values and yielding a set of output values. The execution order of operations can be derived by connecting input and output values or by explicitly defining the control flow via designated keywords (in textual orchestration languages) or structures (in graphical orchestration languages).

In general, the operation chain defines the steps needed to transform a set of input models to the final outcome of the MDSE project. The final outcome can take various forms: it can be a partial or full implementation of the system by using code generation or it can be a set of executable models if model interpretation techniques are used.

During the development phase of an MDSE project, models and model operations are bound to change frequently. To evaluate the impact of those changes on the final outcome, the operation chain needs to be executed. In this thesis, I propose *Moola*, a domains-specific language (DSL) [23] to describe and execute operation chains. Moola draws inspiration from build tools, from orchestration languages and from existing approaches in the domain of operation orchestration.

1.1 Problem Statement

The development activities in MDSE projects focus around creating models and operations and orchestrating them to operation chains. These chains typically consist of several operations which may require to run in specific order, on specific constraints and for a specific number of times. To evaluate the impact of changes to models and operations on the overall outcome, the operation chain needs to be executed whenever modeling artifacts change. The developer has several choices on how to execute the operation chain:

- **Manual Execution**: Many development environments allow direct execution of operations from within the environment. Developers can choose to run all operations of a chain

one-by-one through the IDE. Depending on the size of the project, this may become impractical or infeasible at some point. It is not only cumbersome, but also error-prone, since the developer needs to ensure that all operations run in the correct order. Furthermore, it can be inefficient when it comes to resource utilization, since intermediate results have to be persisted until they are needed. This requires serialization and deserialization of models, which could otherwise be avoided.

- **Using General-purpose Languages**: Another option is to write code in a general-purpose language that describes the orchestration chain and can easily be rerun whenever changes occur. However, special attention has to be given to the design of the code to make it robust to future changes. This may lead to a full-blown project around the operation chain itself, including modularization, testing, build automation, etc. and may amount to considerable additional work.

- **Using Build or Workflow Tools**: A widespread approach to write executable orchestration chains is to use build tools such as Ant[1] or Gradle[2]. These tools expose powerful extension mechanisms and allow the integrating of model operations to a workflow. The strong resemblance of operation chains and build processes is exploited in such scenarios. However, operation chains and build processes do diverge in some aspects: build steps typically depend on each other and thereby form a directed, acyclic graph (DAG) [10]. The same cannot be said for operation chains, in which circular dependencies among operations may exist. This results in some operation chains not being easily describable as build process. Furthermore, build tools first need to be adopted to the MDE world by using their extension mechanism. If this is done properly, build tools can be used to describe orchestration chains, as the Epsilon project [46] has shown with its Ant-based workflow engine [45].

- **Using Orchestration Languages**: Another way to describe orchestration chains is to use dedicated orchestration languages. These languages are tailored to the needs inherent to the domain of operation orchestration and allow describing orchestration chains in the vocabulary of the domain. With the increasing popularity of MDE, more and more orchestration languages have emerged. Existing solutions differ in implementation style, modeling framework and language support and tooling integration. So far, none of the existing approaches has gained wide-spread adoption in the industry.

In contrast to other software engineering activities, in which the usage of build tools is a common occurrence, the MDE world has yet to find and widely adopt a way to describe operation chains. This thesis centers around the question on how to textually represented operation chains and how they can be efficiently executed. Referring to this question, current build tools and DSLs are compared and evaluated for their capabilities.

[1] http://ant.apache.org/
[2] http://gradle.org/

1.2 Aim of Work

In this thesis, I propose *Moola*, a domains-specific language (DSL) [23] specifically designed to describe operation chains. During the development of Moola, special attention was given to *performance* and *extensibility*.

Orchestration chains typically consist of several operations which may require to run in specific order, on specific constraints or for a specific number of times. To increase performance, independent operations can be executed in parallel. Furthermore, intermediate results can be passed in-memory, saving time that would otherwise be spent on serializing and deserializing them.

Many different frameworks can be used to describe models and operations. In the Java universe with its strong Eclipse community, the Eclipse Modeling Framework (EMF) [74] is of great importance. Tools such as *Acceleo*[3], *MoDisco*[4] and *ATL*[5] play a fundamental role for MDE in Java. Moola's plug-in system allows the integration of various modeling frameworks and operation languages. To highlight this point, a sample plug-in for EMF was implemented, which allows creating orchestration chains from the aforementioned tools.

A final point in the design of Moola has been the language markup and eco-system. DSLs typically suffer from a number of disadvantages, such as developers reluctant to invest into learning a new language with limited applicability and lacking tooling support [71]. To lower the entry threshold, Moola was designed to reuse core elements of Java such as control flow keywords, method calls, assignments, exception handling, etc. Additionally, an Eclipse plug-in was developed, enabling syntax highlighting for Moola files and allowing developers to run Moola from within the IDE.

All Moola artifacts can be found on GitHub (`https://github.com/We-St/moola`).

1.3 Methodological Approach

With the growing importance of DSLs, a number of patterns describing the process of developing DSLs have emerged in recent years. These reach from traditional, top-down-based approaches [53, 80, 81] to more agile, iterative approaches [85]. While all of them are based on a thorough analysis of the problem domain, they differ in later phases of the development process. A well-established, often-cited, top-down approach for developing DSLs was introduced by Mernik et al. [53] and consists of the following phases:

1. **Decision**: The fundamental decision whether to develop a DSL or not needs to be answered. Mernik et al. address organizations and companies in which this decision is pending by listing several decision patterns. These patterns describe scenarios in which DSLs have been successfully implemented in the past and may justify a decision in favor of developing a DSL.

[3] https://eclipse.org/acceleo/
[4] https://eclipse.org/MoDisco/
[5] https://eclipse.org/atl/

2. **Analysis**: During the analysis phase, a thorough understanding of the problem domain is acquired and potential features of and requirements on the DSL are gathered. Different technical artifacts can form the basis for this phase, including existing code in DSLs or GPLs, documents, etc. The analysis and its result can vary in formality depending on the analysis method used.

3. **Design**: Based on the features and requirements gathered in the analysis phase, a key decision is made on how to develop the DSL. Mernik et al. distinguish two primary options: *language invention* describes designing and implementing a DSL from scratch whereas *language exploitation* describes extending or specializing an existing language (GPL or DSL). Whichever option is chosen, the design phase then deals with creating appropriate specifications for use during implementation.

4. **Implementation**: The language is constructed based on the previously agreed-upon design. Depending on which option is taken, different ways to implement the DSL are possible, e.g. creating an interpreter or preprocessor, embedding the DSL in a (GPL) host language, extending a compiler, etc. For executable DSLs, the implementation phase also includes the creation or preparation of a suitable runtime environment.

5. **Deployment**: The DSL is shipped to the users, who are now able to apply the DSL to the application domain. If the DSL is executable, deployment includes the provision of a suitable runtime environment to the users.

While Visser [85] generally agrees on the development phases, he mentions *maintenance* as additional phase to illustrate the importance of continuous support after the initial development has finished. Furthermore, he suggests an iterative approach, where analysis, design and implementation phases are performed in closer relationship to agile software development than to classical top-down approaches. In such an iterative development process, each development cycle delivers a small but stable version of the DSL. Over several cycles, the initial version incrementally grows more sophisticated until a mature state is reached and the application domain is covered by the DSL to sufficient extent. The advantage of this approach is, as with other agile approaches [65, 75], a more flexible development process, which allows adjusting requirements and incorporating feedback of users early on to mitigate the impact of undesired developments.

In this work, an agile, iterative approach as suggested by Visser was used. Modifications to Visser's approach were made to account for the university environment in which Moola was developed. Figure 1.1 depicts the development process used for Moola.

During the *decision* phase, reasons to use DSLs for describing operation chains were gathered. Furthermore, the choice to add yet another DSL to a field dense with existing DSLs was motivated. A thorough analysis of the problem domain followed. This included investigating application scenarios and existing solutions for common notions and terminology. The result of the *analysis* phase was a list of features and key requirements. The *design* phase dealt with the question whether Moola should be a customization of an existing language or developed from scratch. After the choice was made to build a new language, an implementation pattern was selected, i.e. embedding in a host language, and an idea developed on how to naturally integrate the domain terminology and notions to the host language. While the model of Mernik et al.

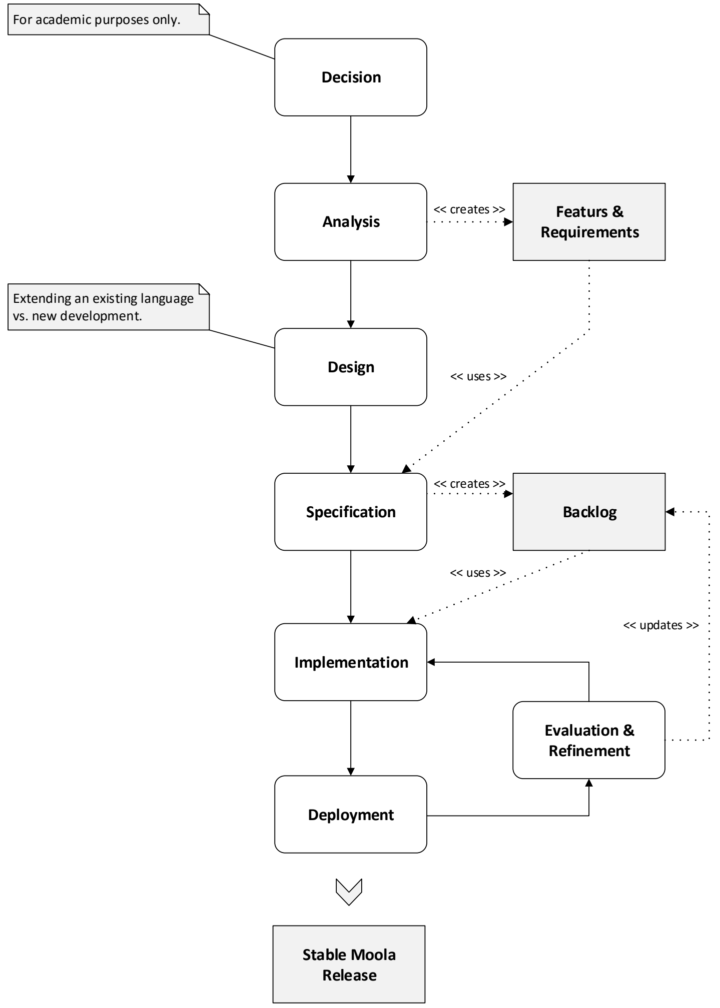

Figure 1.1: Development Phases for Moola.

suggests the creation of a specification as part of the design phase, a separate, follow-up *specification* phase was used for Moola. This phase dealt with creating Moola's syntax and semantics in compliance with the previously gathered feature list and in accordance to the chosen implementation pattern. The result of the specification phase was a backlog, i.e. a list of prioritized tasks needed to reach a mature state for Moola. During the *implementation* phase, a subset of these tasks was realized. The *deployment* consisted of a release of Moola on an internal code repository. After a brief evaluation, the backlog tasks were adjusted and, if necessary, reordered. This implementation-deployment-evaluation-refinement cycle was performed several times until a stable, mature version of Moola was implemented.

1.4 Application Scenario

Figure 1.2 shows a non-trivial application scenario in form of an operation chain. This scenario is referenced and gradually implemented throughout this work and shows how a system implementation (Java code and SQL scripts) can be generated by applying a sequence of model operations on a single input model. Outputs of one operation form the input for other operations, hence dependencies between these operations exist, which force a certain execution order.

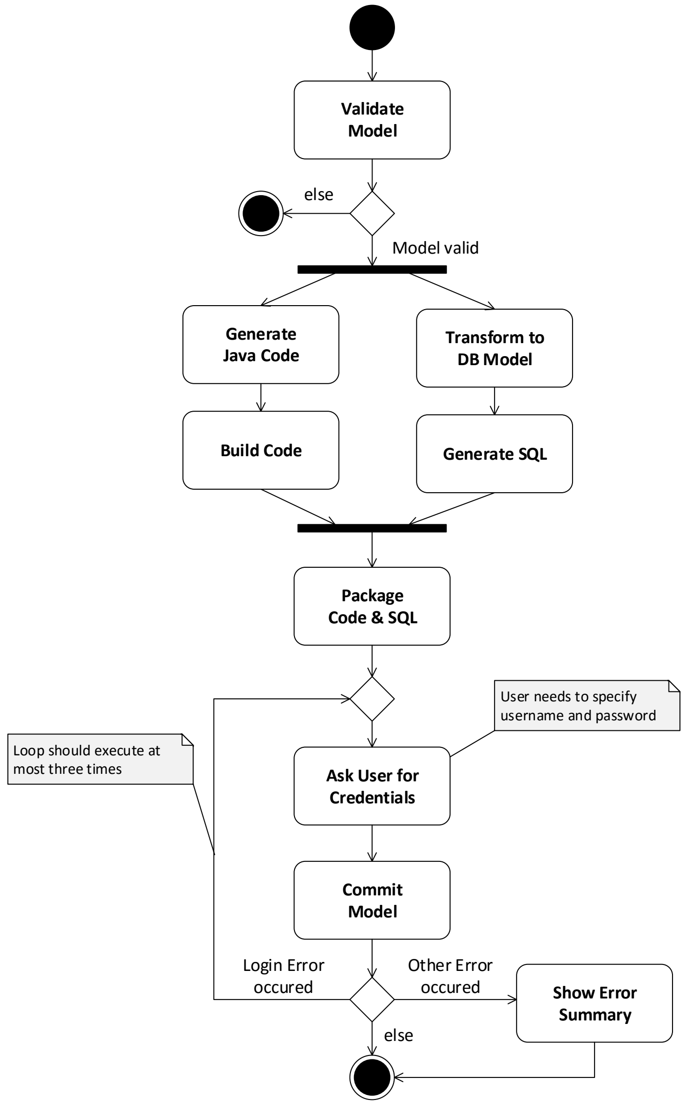

Figure 1.2: Application Scenario.

Starting from a domain model, which is first validated, the Java classes are generated and built by calling the Java compiler. Simultaneously, the domain model is transformed to a

database model, which in term is used to generate SQL scripts to create the database. After all artifacts (Java code and SQL scripts) have been generated, they are packaged and copied to a deploy folder. The final part of the orchestration chain is to commit the model to a version control system. To do so, users first need to enter their credentials (username and password). If the model cannot be committed due to login errors, users have two additional chances to enter correct credentials. If they fail to do so or if any other errors occur, the error is displayed before the operation chain terminates.

1.5 Structure of the Work

In chapter 2, background knowledge on MDE and DSLs is provided. Afterwards, Groovy's features for DSL development are introduced. Chapter 3 starts with a feature analysis of sample operation chains and existing orchestration approaches. It then introduces the design mentality behind Moola. It finishes with a language specification and a sample implementation of the application scenario. Chapter 4 shows how Groovy's features are used to implement the previously introduced language specification. In chapter 5, Moola is evaluated for correctness and feature completeness. It is then applied to real-life application scenarios taken from the ARTIST project to test its applicability. Chapter 6 introduces other approaches in the domain of operation orchestration and compares them to Moola. Finally, chapter 7 summarizes this work and presents areas for future work.

CHAPTER 2

Background

"Not all those who wander are lost."
— J. R. R. Tolkien, *The Lord of the Rings*

This chapter introduces the core concepts and methodologies used in this thesis. It starts off with an introduction to model-driven engineering (MDE) and domain-specific languages (DSL) and finishes on how Groovy can be used to develop DSLs.

2.1 Model-Driven Engineering

Model-driven engineering (MDE) [38] is a development methodology that puts models at the core of the development process. Instead of having several representations (e.g. code, documentation, models, etc.) of a single system independent from each other, MDE advocates the use of models as foremost development artifacts. By focusing on models, other representations can be automatically updated if changes occur. By the time MDE was adopted in practice, several sub disciplines have emerged which apply the model-centric view on different development tasks. Some of them are:

- **MDSD** [72]: Model-Driven Software Development focuses on creating executable software systems by either generating code from models or by defining executable models as input for model interpretation engines.
- **MDRE** [61]: Model-Driven Reverse Engineering can be used to analyze a given software artifact (e.g. a legacy system). In contrast to MDSD, MDRE does not generate an executable piece of software but rather generates knowledge of the analyzed system and results in one or more models describing the system.
- **MDT** [55]: Model-Driven Testing focuses on creating test cases from models. The final outcome of a MDT project is a suite of test cases that can be used to validate the implementation of a system.

In practice, many more sub disciplines exist and show the diversity of application areas for MDE. Although the final outcome varies from one sub discipline to another, all of them put models at the center of development activities. Aside from models, two other important types of artifacts are present in most MDE projects: *metamodels* and *transformations*. To fully understand the concepts behind MDE, an understanding of *models*, *metamodels* and *transformations* is required.

Models

A model is a representation of a system or real-world object by focusing on relevant features at a certain level of abstraction. A model, by definition, does not include all aspects in full detail, but rather describes a subset of important details to highlight them. Although models do not represent reality in full detail, their inherent reduction on relevant information allows viewing a system or object in reduced complexity.

Mellor et al. [51] define a model as *„a coherent set of formal elements describing something (for example, a system, bank, phone, or train) built for some purpose that is amenable to a particular form of analysis"*. The authors later describe some applications of models such as communication of ideas, completeness checking and transformation into implementations.

Kleppe et al. [42] define a model as *„a description of a (part of) systems written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer"*.

Notable about both definitions is the mentioning of formal rules for models. While Mellor speaks of *formal elements*, Kleppe's definition is more concrete and requires a *well-defined language*. Such formal rules on models can be achieved by using *metamodels*.

Metamodels

Models are an abstraction of one or more observed items. In Software Engineering, these items are often systems-to-be, processes or some real-world objects. However, models can also describe other models. So-called *metamodels* define the elements that may exist in a model, how they may be connected to each other and what the meaning of the elements and their relationship is. By defining syntactical rules, a metamodel describes a possibly infinitely large set of models that can be expressed to fulfill these rules. A model is said to *conform to* a metamodel if the model complies to all syntactical rules defined by the metamodel. Just as models can be used to describe other models, models can also be used to describe metamodels. Meta-metamodels such as MOF[1] and Ecore[2] are used to define modeling language families and thereby help fostering compatibility between metamodels [6].

Figure 2.1 illustrates the concepts of models, metamodels and meta-metamodels. The real-world object to model is a car (level M0). The first level of modeling is represented by UML's class diagram to describe the real-world object and thereby capture information that is important for the specific use case (level M1). The class diagram itself is described in the UML language

[1] http://www.omg.org/mof/

[2] https://eclipse.org/modeling/emf/

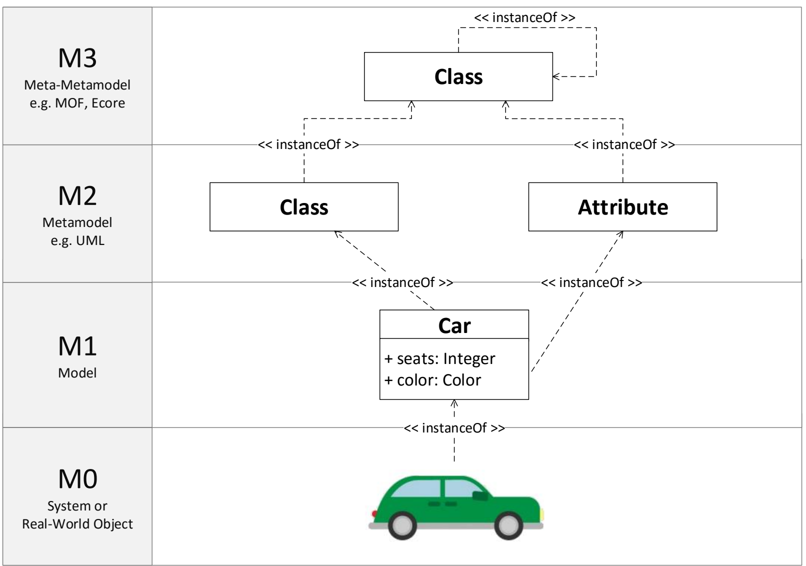

Figure 2.1: Abstraction levels from real-world (M0) to meta-metamodels (M3).

specification, which constitutes the metamodel. It describes the elements allowed (classes, attributes, etc.) and how they might be interconnected (level M2). Finally, the metamodel is described by a meta-metamodel, which in case of UML is MOF (level M3). It was shown in practice that meta-metamodels can describe themselves, so no further levels are required.

The *conforms to* relationship between a model an its metamodel is an important part of MDE. Since the metamodel governs which elements may occur in a model and how these elements may be connected, a metamodel can be seen as *type* of the model [78]. This is especially important in scenarios where models need to be replaced. If a new model conforms to the same metamodel and therefore can be seen as having the same type as the old model, replacing the old model with the new one is possible [73]. This idea can be extended to model subtyping [60], in analogy to object subtyping used in object-oriented languages.

In recent years, efforts were made to capture the whole MDE methodology with all its elements and concepts in a model, thereby creating what is called a *megamodel* [20]. This shows that MDE is capable of modeling complex software systems up to a recursive self-definition of itself.

Transformations

MDE advocates the use of models as core artifacts in the development process. To reflect a system in sufficient detail, typically several models of different nature are required. This can be a mix of static and dynamic models form the same language family (such as UML) or models on different levels of abstraction[3]. To yield the final outcome of the MDE project, operations need

[3]MDA uses models on several levels of abstraction and is introduced later in this chapter

to be executed that take the models as input. In a typical project, several such operations are required, transforming the models through intermediate steps to the overall outcome. A specific type of operation that transforms one or more input models to one ore more output models is called *transformation*. Transformations are a core element in the MDE world. They are defined on metamodel level, meaning that metamodels are used to describe the expected inputs and outputs of the transformation. After a transformation is defined and implemented, it can be executed on any set of input models conforming to the corresponding metamodels, making it easy to reuse the transformation in similar scenarios.

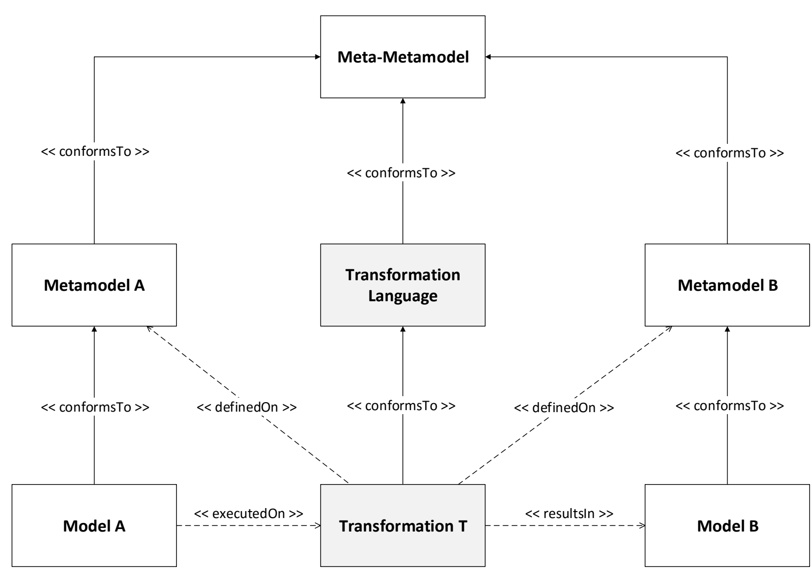

Figure 2.2: Transformation definition and execution.

Figure 2.2 illustrates the principle of defining transformations on metamodels and executing them on concrete models. Applying modeling concepts on transformations themselves, transformations can be seen as models, making it possible to define a metamodel for them. This leads to *transformation languages* that may even depend on the same meta-metamodel as the input and output metamodels, thus enabling the description of transformations with the means inherent to the MDE world. Describing transformations with a metamodel and in doing so merging the concepts of model and transformation in one entity has the advantage of allowing transformations to act as inputs to other transformations or enabling transformations to generate transformations, thus allowing for so-called higher-order transformations (HOT) [76].

Referring back to the *type* relation between a model and its metamodel, the signature of a transformation can be defined similarly to function signatures in programming languages [78]. The signature of a transformation T that takes exactly one input model conforming to a metamodel M and yields a model conforming to a metamodel M' can be defined as function $T : M \rightarrow M'$. This highlights the fact that the transformation T can be used on any model conforming to the input metamodel M and that T guarantees to deliver a model conforming to M'.

This allows for contravariance of input types and covariance of output types in transformations and enables reasoning on type safety when combining transformations and other operations to operation chains [31, 73].

Transformations exhibit their full potential when they are defined in an executable transformation language. By enabling developers to easily execute transformations, reusing them on different input models in the same or in a different project becomes much more feasible. In theory, any mapping between two metamodels can be seen as transformation. Describing such a mapping can be done in many ways: a general-purpose language such as Java or Groovy or DSLs with specific notions and concepts can be used [67]. Alternatively, the transformation itself can be generated from an abstract representation of the mapping by using higher-order transformations [8]. In recent years, many DSLs have emerged that allow describing transformations in an expressive way. Examples of such languages include QVT, ATL and Acceleo.

Model-Driven Architecture

One specific adoption of MDE in practice is model-driven architecture (MDA) [43]. MDA is a trademark of OMG[4], which realizes its vision of MDE by using OMG's own standards, including UML and MOF. MDA consists of a three-layered approach: at first, Platform Independent Models (PIM) are defined, which describe the problem domain and solution without specifying any implementation details. In a next step, implementation details are added to the PIMs, thereby creating platform specific models (PSM). Since PSMs contain all relevant information for implementing the system, a full or partial implementation of the system can be generated from them. Figure 2.3 shows the basic concept of MDA in coarse detail: PIMs are defined first and, by using transformations, refined to PSMs and later to code.

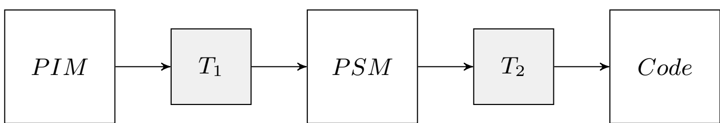

Figure 2.3: From PIM over PSM to code

By introducing this structure, MDA claims to improve productivity, portability, interoperability and maintainability [42]. Since the transformations from PIM to PSM and from PSM to code can be reused, MDA advocates the shift from code-centric development to model-centric development. In OMG's vision, developers should focus on writing PIMs. After transformations for a target platform are defined, these transformations can be easily reused in the context of new projects, thus increasing the *productivity* of developers. The *portability* is increased by focusing the development process on PIMs. Since they are platform independent by definition, they can represent the starting point for generating the implementation of a system for various platforms. If a system is targeted at multiple platforms, several PSMs can be generated out of the existing PIMs. If the target platforms exhibit limited communication features, the transformations can not only generate PSMs, but also *code bridges*, thus increasing *interoperability*. Finally,

[4] http://www.omg.org/

since PIMs are the starting point for any development (including changes to existing systems), they represent a single-point of truth and can be directly used in the documentation to increase *maintainability*. Evidence suggests that MDE at least partially fulfills these promises [35].

Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [74] is another adoption of MDE in practice and brings the ideas and concepts of MDE to the Eclipse platform. It is defined around the Ecore (meta-) metamodel and allows defining Ecore-conform models in various ways (e.g. as annotated Java code, using UML diagrams, via XML schemas, etc). The Ecore metamodel can be seen as subset of the class diagram specified by UML and allows for describing the static structure of a system. An Ecore-conform model can then be used to generate Java code (interfaces, implementing classes, etc), adapters and editors (for editing and displaying modeled classes)[5]. EMF automatically adds several features to the generated code such as support for change notification and an enhanced reflection API. Furthermore, EMF defines a default XMI serialization.

In recent years, an active community has gathered around EMF and Ecore, putting the latter at the heart of many MDE tools. Transformation languages such as ATL and Acceleo operate on Ecore-conform models. MDT/UML2[6] is an implementation of OMG's UML standard using Ecore, bringing UML models to the EMF world. Since MDT/UML2 was realized with Ecore, any transformation language compatible with Ecore-conform models is automatically compatible with models based on MDT/UML2. Similar projects exist for BPMN[7], IMM[8], etc.

Most of the aforementioned transformation languages are integrated to the Eclipse IDE via its extensive plug-in system. This allows MDE developers to use a single platform to define models and operations and later execute those operations to generate output. Although some of these tools offer a CLI or are available as standalone libraries, others can only be executed in the Eclipse context, i.e. when a running Eclipse workspace exists. This is especially true for MoDisco. In case such Eclipse-only tools need to be orchestrated, the orchestration language also needs to be able to run within the Eclipse context.

2.2 Domain-Specific Languages

A domain-specific language (DSL) [23] is a programming language tailored to a specific application domain. While a DSL is limited in its field of use, it is designed to include constructs, keywords and terminology of the problem domain and allows describing problems with the language and vocabulary inherent to the problem domain. In contrast, general-purpose languages (GPL) such as Java, C# or C++ can be applied to nearly any domain. Although they excel in versatility, programs written in a general-purpose language are often verbose and need boilerplate code in order to be applied to a specific domain.

[5] https://eclipse.org/modeling/emf/
[6] http://wiki.eclipse.org/MDT-UML2
[7] http://www.omg.org/cgi-bin/doc?bmi/2007-6-5
[8] http://www.omg.org/cgi-bin/doc?ab/2005-12-2

The term *domain-specific* language is a recent invention. Languages for a specific field of use have been implemented long before the term found widespread adoption. Some popular domain-specific languages such as *SQL* for querying databases, *HTML* for describing websites and *CSS* for styling websites are extensively used in practice.

When implemented and applied correctly, DSLs offer a number of advantages over GPLs. Since DSLs are designed to describe problems in a certain domain on a certain level of abstraction, the syntax can be adjusted to the target audience, thereby demanding more or less programming skills up to a level where even domain experts with limited programming skills can read or even write code in the DSL. In doing so, DSLs help foster collaboration between developers and domain experts [33]. Also, DSLs are typically defined on a high level of abstraction, hiding some of the complexity of the underlying domain. This impacts DSL programs, which tend to be more concise and readable than their GPL counterparts. An increase in productivity and maintainability can be observed when writing DSL programs [27].

However, DSLs also come with some serious disadvantages. DSL development requires knowledge in language design and the application domain. Therefore, creating new DSLs is a non-trivial task. Furthermore, DSL development is linked to considerable up-front cost that only repay themselves later and if the resulting DSL was designed and implemented correctly. The DSL needs to be maintained and updated, e.g. when the understanding of the problem domain grows or changes [27]. For all this reasons, developing a custom DSL is out of bounds for many organizations. Even if a third-party DSL is used, some disadvantages remain. First, the right DSL needs to be chosen. Secondly, independent of using a custom or third-party DSL, the users need to be educated in the DSL itself and the associated tooling [81].

Research has shown that successfully adopting DSLs in practice relies on a number of factors: the usability and tooling support of the DSL, how easy the DSL can be learned and how expressive the DSL is when it comes to describing problems in its problem domain [32]. Empirical studies have shown that DSLs, when applied properly, can help reduce time-to-market [37] and improve maintainability and extensibility [1].

The maturity of the field of DSL development expresses itself in a number of categories DSLs can be divided into. In the following section, *internal* and *external* DSLs are introduced. The discussion continues to (*non-*) executable DSLs and then introduces different language characteristics such as programming paradigms and type systems.

Internal vs. External DSLs

DSLs can be divided into *internal* and *external* domain-specific languages. Internal DSLs, also called *embedded* DSLs, sit on top of a host language. This reduces time and cost of DSL development since tooling support for the host language already exists. This tooling support can reach from development environments to compilers and finally to execution environments. Programs written in internal DSLs allow developers to include code of the host language, allowing the DSL to reuse host language features and lower the entry burden for developers with existing knowledge of the host language. The downside of embedded DSLs is the syntax limitations imposed by the host language [47]. Although many general-purpose languages such as Groovy, Scala and Ruby include specific features to allow internal DSL development, they come with a set of syntactical constraints that cannot easily be bypassed [14]. External DSLs on the other

hand do not rely on a host language and are therefore not limited by any syntax restrictions. They can be designed to exactly reflect the problem domain and hence allow modeling problems more expressively than internal DSLs can [23]. Several ways to implement external DSLs exist, e.g. adopting/extending existing compilers, creating new compilers using compiler frameworks, using preprocessors, etc. [25, 47].

Executable vs. Non-executable DSLs

DSLs can be *executable* or *non-executable*. An executable DSL comes with a suitable runtime environment that allows running DSL programs. The task of developing such a DSL does not only concern itself with the language itself, but includes the runtime environment for all targeted platforms as well. In contrast to GPLs, DSLs do not need to be executable [53]. Non-executable DSLs can be used as means of representing information and are applied in technical documents like documentation and specifications. Since non-executable DSLs still have well-defined syntax and semantics, tooling support for them can be implemented. An example for such a non-executable DSL is JSON[9], the JavaScript Object Notation. JSON is a data-exchange format based on lists and name-value pairs, which can be used to describe plain objects e.g. in interface definitions.

Language Characteristics

General-purpose and domain-specific languages alike exhibit certain characteristics by which they can be distinguished and categorized. In the following, some of the more important language characteristics for this work are introduced.

- **Programming Paradigm**: A programming paradigm describes the principles used in a programming language that govern the structure of programs (static aspect) and how single elements within a program interact (dynamic aspect). One programming language is not limited to one programming paradigm but can support several at the same time.
  A core distinction of programming languages is between *imperative* and *declarative*. An imperative programming language explicitly describes control and data flow in a program. In doing so, it specifies *how* a specific result should be reached. In contrast, declarative programming languages are used to describe the desired result. They focus on *what* should be achieved rather than how to do it.
  Object-oriented programming (OOP) [11] is a widespread paradigm for many general-purpose languages. Object-oriented languages use interacting objects as main design element and are typically imperative. Another important paradigm with widespread adoption is functional programming (FP) [86]. It promotes functions to first-class citizens and therefore allows higher-order functions (HOF). Functional languages are typically declarative in nature. Logic programming (LP) [49] allows expressing facts and rules about a system and relies on formal logic. As with FP, most languages using a logic-based paradigm are declarative.

[9] http://www.json.org/

Many widely used languages implement several paradigms. This is especially true for object-oriented languages, which are mostly imperative, but also incorporate declarative and functional aspects. Examples are Groovy, Java 8 (via streams) and C# (via Linq).

- **Type System**: A type system is used to guarantee a certain level of correctness within a program. By assigning a type to constructs such as variables, parameters, methods, functions and expressions, possible bugs can be found on type mismatch. Depending on the time of type checking, two approaches are distinguished. Programming languages using *static type checking* check for type safety before execution, typically by the compiler during compile time. In contrast, *dynamic type checking* is done at runtime. In practice, several languages exist that allow for both static and dynamic type checking. These *optionally-typed* languages are dynamically-typed by default but allow parts or all of the code to be statically checked.

  Static and dynamic typing each have their advantages. Statically-typed code guarantees a certain level of type safety at runtime. Due to the type information, compilers can optimize the code to a greater extent than dynamically-typed languages allow. Furthermore, the conceptual framework that types bestow on a program helps developers in understanding and maintaining code [30]. It also enables advanced tooling support since the additional type information helps in navigation and syntax highlighting. Despite those advantages, statically-typed languages are of limited use when features like late binding or runtime meta-programming are required. In such cases, dynamically- or optionally-typed languages are more commonly applied.

  The debate whether statically- or dynamically-typed languages provide advantages in development time and maintainability is still ongoing [29,40] and has not produced sufficient prove for either side.

- **Concrete Syntax**: DSLs can either be represented using a *textual* or a *graphical* syntax. A textual syntax consists of keywords and language constructs that can be used to assemble a program in textual form, e.g. plain text. In contrast, a graphical syntax defines visual elements and how they can be interconnected. The task of writing a program in a graphical DSL transforms to modeling the problem in a visual editor, thereby conforming to the rules of the graphical syntax, e.g. which elements are allowed and how they can be interconnected.

2.3 DSL Development in Groovy

Groovy is a language of choice for many DSL developers for its build-in features specifically targetted at DSL development. It is an optionally-typed, general-purpose language for the JVM and a superset of Java, meaning that any valid Java program is also a valid Groovy program and that all features and libraries available for Java can also be used with Groovy. Like Java, Groovy compiles to Java Byte Code and can therefore reference and be referenced from ordinary Java code. Furthermore, it runs on a Java Virtual Machine much the same way as Java does and is

therefore platform independent. Groovy was first mention in 2003[10] and finally released in 2007. With the release of Grails[11], a Groovy-based web development framework in 2008, Groovy received widespread attention. As a comprehensive overview to Groovy is out of scope for this thesis, interested readers are referred to [44]. In the remainder of this section, the concepts that make Groovy a language of choice for DSL developers are introduced. Further details on DSL development in Groovy can be found in [26].

Syntax

Since Java is a subset of Groovy, all keywords and constructs of Java retain their meaning in Groovy. This especially means that any valid Java program can be compiled with the Groovy compiler and later be executed on the JVM. Groovy relaxes the syntax of Java in several places to give a certain degree of freedom to DSL developers. Some of the relevant relaxations for DSL development are:

- No semicolons are required at the end of a statement.
- Methods with parameters can be called without parenthesis.
- No explicit *return* statement is required. Instead, the value of the last statement in a method or closure is returned.

Developers writing Groovy code can decide whether to use any of these syntax relaxations or to stick to the Java syntax, which is an equally valid way of writing Groovy code. When developing domain-specific languages, these relaxations can be used to adjust the syntax of the DSL to the target audience. Depending on the amount of relaxations used, DSLs can be tailored to resemble natural languages rather than programming languages.

Listing 2.1: Syntax Relaxations.

```
int add(int a, int b){
    a + b // Value of last statement is returned.
}

def result = add 10, 20 // No semicolon or parenthesis required.
```

In addition to calling methods without parenthesis, Groovy allows developers to omit dots on method calls in certain situations. This allows for a fluent definition of code, which closely resembles natural, written language.

Listing 2.2: Omitting Parenthesis and Dots.

```
// Regular calls by chaining methods.
get(position).of(king).on(chessboard)
move(king).to(right).by(1).on(chessboard)

// Chaining methods and removing dots and parenthesis.
get position of king on chessboard
move king to right by 1 on chessboard
```

[10] http://radio-weblogs.com/0112098/2003/08/29.html
[11] https://grails.org/

Groovy allows unwrapping a list of values to a list of variables in a single assignment. These statements are called *multiple assignments*. Doing so will assign the first item of the list to the first variable, the second item of the list to the second variable and so on. An example on how to use multiple assignments is shown in Listing 2.3. Multiple assignments can be extended to unwrap results of a method invocation, if the method returns a list of values.

Listing 2.3: Multiple Assignment.

```
def values = ['Winnie', 'Pooh', 7]
def (firstname, lastname, age) = values

def (title, artist, price) = getAlbum()
def getAlbum(){
    ["Cool␣Album␣Title", "Cool␣Artist", 9.99]
}
```

Closures

Groovy introduces concepts from functional programming through *closures*. Conceptually, a closure is comparable to an anonymous function that also captures the environment it was defined in. This allows closures to access variables and methods of their defining scope. The name originates from the idea of a *lambda expressions* „closing“ over the unbound variables to form an executable piece of code.

Closures can be used in many places to define inline methods. Similar to regular methods, closures can take parameters and return values. If no parameters are explicitly defined, Groovy adds the *it* parameters implicitly to each closure. Listing 2.4 demonstrates the versatility of closures.

Listing 2.4: Closures.

```
// Define an inline method for summing up objects.
def add = { a, b -> a + b }
def result = add 5, 10

// Execute a closure on each element of a list.
def names = ["Donkey", "Tigger", "Piglet", "Winnie", "Rabbit"]
names.each {
    println "Hallo␣\$it" // "it" is defined implicitly.
}
```

Closures are a fundamental part for DSL development in Groovy. The reason for this is the way Groovy resolves references to variables and methods within a closure, which is fully customizable by the DSL developer. Groovy introduces three scoping references: *this*, *owner* and *delegate*. *This* behaves like its equivalent in Java and references the object in which the closure was defined. *Owner* is the same as *this* in most cases, but refers to the outer closure in case several closures are nested. *Delegate* is the most interesting one from a DSL developer's point of view. By default, it holds the same value as *this*, but it is the only one of the three that can be changed programmatically. When a closure is executed and a method call or variable access is encountered, Groovy first tries to resolve the reference by looking at the object linked in the *this* reference. In case the method or variable could not be found, Groovy refers to the *owner* object and finally to the *delegate*. This so-called *Resolve Strategy* can be changed, allowing for a

different order on the scope look-ups. This especially allows for a *delegate-first* strategy, which enables developers to execute a closure on any object. Listing 2.5 demonstrates the default resolve and delegate-first strategy.

Listing 2.5: Resolve Strategy.

```
def combine = {
    firstname + lastname
}

def firstname = "Winnie"
def lastname = "Pooh"
combine() // Results in "Winnie Pooh".


combine.delegate = new Person()
combine.resolveStrategy = Closure.DELEGATE_FIRST
combine() // Results in "Christopher Robin".

class Person {
    def firstname = "Christopher"
    def lastname = "Robin"
}
```

If a method expects a closure as last parameter, it can be written as if the closure follows the method call. This allows for a shorter and more concise syntax. Listing 2.6 shows several ways of how to pass closures as parameters.

Listing 2.6: Closures as Parameters.

```
def names = ["Donkey", "Tigger", "Piglet", "Winnie", "Rabbit"]

names.each({
    print it // Closure as part of the parameter list.
})

names.each() {
    print it // Closure trailing the parameter list.
}

names.each {
    print it // Omitted parenthesis on method call.
}
```

Metaprogramming

One of the most powerful arrows in the quiver of a Groovy DSL developer is metaprogramming. Metaprogramming is the task of writing programs that operate on program code as input data. Such programs can read and manipulate external program code or *their own code* to change the behavior of that code. Groovy applies metaprogramming by allowing programs to manipulate their own code either at compile-time or at runtime. Depending on the phase in which metaprogramming is applied, Groovy distinguished between *compile-time* and *runtime* metaprogramming. Typical metaprogramming activities include adding properties and methods to classes, changing the behavior of existing methods or changing the inheritance hierarchy.

Runtime Metaprogramming

Runtime metaprogramming refers to changing the behavior of classes and objects at runtime. Similar to Java, each object in Groovy has a reference to its defining class. This class object represents the class at runtime and provides the facilities for *reflection* in Java. It can be accessed to read meta information on the class such as defined methods, properties, attributes, etc. The access thereby is strictly read-only. Java does not support changing the behavior of a class object at runtime. This behavior is the same in Groovy. However, in Groovy each object has an additional reference to a so-called *metaclass*. The metaclass is, again, a representation of the class at runtime. In constract to the class object, the metaclass can be changed at runtime and therefore allows adding methods, changing existing methods, etc. Changing a metaclass will affect all instances of the corresponding class. In general, the duality of class references can be seen as this: the class object describes the class at compile time, while the metaclass object defines the class at runtime. Listing 2.7 demonstrates how the metaclass of an object can be manipulated to add a method at runtime.

Listing 2.7: Adding Methods to the Metaclass.

```
def obj = new Object()
obj.metaClass.sayHello = {
    println "Hello!"
}
obj.sayHello()
```

To fully understand runtime metaprogramming in Groovy, the Groovy Metaobject Protocol (MOP) requires a short introduction. The MOP defines the steps the Groovy runtime takes in order to resolve calls to methods and properties. Since the process is the same for both methods and properties, only the process of method look-ups is describe here. However, the same principles apply for property look-ups.

When a method is invoked on an object, the Groovy runtime first looks at the metaclass. If the method is not defined there, the class of the object is checked. Furthermore, the MOP defines several places where the look-up process can be intercepted. One important way of intercepting the look-up is by overriding the *methodMissing* method, which is defined for all Groovy classes. If a method cannot be found in the metaclass or the class but the *methodMissing* is present, it will be called instead. It receives name and arguments of the unresolvable method call and can react at runtime. This allows developers to decide at runtime whether the object can handle the method invocation or an exception should be thrown. If a method is not found at runtime, it can be easily added in the *methodMissing* method. Creating methods dynamically is of special interest in the *Builder* design pattern [15].

Compile-time Metaprogramming

As the name suggests, compile-time metaprogramming takes places during the compilation phase. The aim of compile-time metaprogramming in Groovy is to change the behavior of the program by changing the Java Byte Code that is produced during compilation. In contrast to runtime metaprogramming, compile-time metaprogramming does not directly influence program execution. All effects of compile-time metaprogramming are reflected in the Java Byte Code and can be made visible by reverse engineering the class files.

The Groovy compiler defines several phases that are executed consecutively to transform Groovy source code to Java Byte Code. The phases, in order of their execution during compilation, are

1. **Initialization**: The environment is prepared and source files are opened.
2. **Parsing**: The source code is parsed to a Concrete Syntax Tree (CST).
3. **Conversion**: The CST is converted to an Abstract Syntax Tree (AST).
4. **Semantic Analysis**: The source code is checked for consistency and validity that go beyond grammar checks.
5. **Canonicalization**: The remaining parts are added to the AST, which is complete after this phase.
6. **Instruction Selection**: The instruction set for the compilation is chosen and the byte code generated.
7. **Output**: The generated byte code is written to the corresponding files.
8. **Finalization**: The environment is cleaned up.

Two important data structures during compilation are mentioned above: Concrete Syntax Tree (CST) and Abstract Syntax Tree (AST). Both are representations of the source code in memory. While the CST closely resembles the structure of the source code, the AST is a more abstract representation and therefore may not resemble the source code directly. By decoupling concrete and abstract syntax and focusing later compilation phases on the AST, the complexity behind having several identical programs with different markup can be reduced. Compiler optimizations and checks (e.g. for type safety, unreachable code, etc.) are performed on the AST.

The power behind Groovy's compile-time metaprogramming lies in providing hooks to the compilation process for certain phases. By exposing the AST of the program that is compiled, Groovy allows Abstract Syntax Tree transformations, which in term allow changing the AST at compile-time. This mechanism is a powerful way of manipulating the resulting Java Byte Code and allows, to some extent, to bypass the syntax restrictions that are enforced by the Groovy compiler. However, the Groovy compiler needs to be able to generate an AST in the first place. If the source code violates core syntax rules, later compilation phases are not reached and AST transformations can therefore not be applied.

AST transformations can be used to introduce new methods to classes or change the behavior of existing methods. They represent a way to implement aspect-oriented programming (AOP) [39] for Groovy code. Listing 2.8 shows a Groovy code snippet. Figure 2.4 shows a slightly simplified version of the corresponding AST.

Listing 2.8: Example Code for AST.

```
def calc = { a, b, c -> a + b * c }
def result = calc 2, 4, 10
```

When comparing the source code and the resulting AST, a clear resemblance can be determined. The AST consists of two declaration statements. The left side of both statements is

a variable. The right side is a closure on the first statement and its invocation on the second statement.

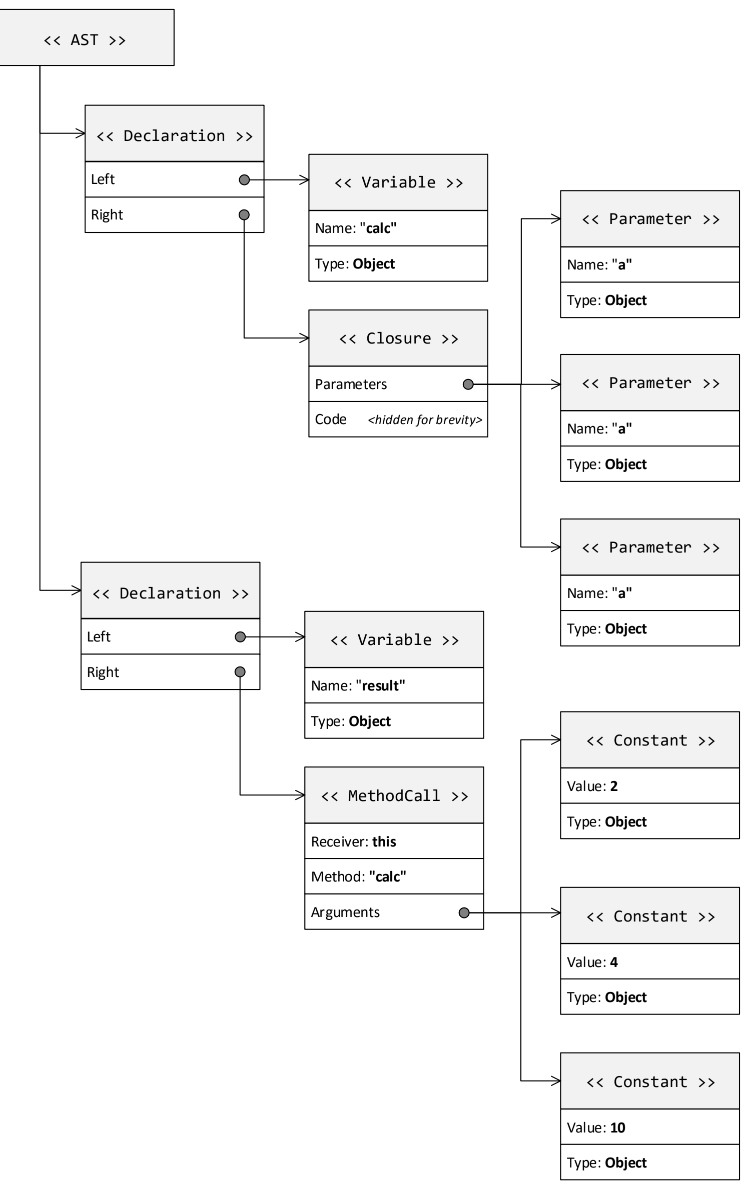

Figure 2.4: Simplified Abstract Syntax Tree for Listing 2.8.

As can be seen in Figure 2.4, the AST nodes have additional information attached to them. Only the most relevant is included in the diagram. By using the information on the AST nodes, an AST transformation developer can determine the receiver of the closure invocation, which in the example is the *this* pointer. Since no concrete types were specified for variables and parameters, the *object* type is assumed by Groovy. Remarkably, this is even true for constants.

By exposing the AST to user-defined operations at compile-time, Groovy allows for a great degree of freedom for DSL developers. In the example above, an AST transformation could globally change the precedence of multiplication over addition by performing the adequate changes on the AST. Aside from changing the behavior, AST nodes can be analyzed to extract information on control and data flow, detect unused code or code patterns known to be a source of error.

Groovy Scripts

A Groovy file can hold executable Groovy code without the need to wrap it in a class and in doing so can be used as script. When the script is run, Groovy automatically generates a class with a *main* method and places the script code correctly inside. This is especially interesting for DSL developers, since the boilerplate code required for defining an executable Groovy program is removed.

Listing 2.9: Java code executing a Groovy script.

```
class GroovyExecutor {

  public BaseClass run(String dslText){
    CompilerConfiguration cc = new CompilerConfiguration();

    // (1) Set base class for the script.
    cc.setScriptBaseClass( ... );

    // (2) Add Abstract Syntax Tree transformations.
    cc.addCompilationCustomizers( ... );

    // (3) Set default imports.
    ImportCustomizer ic = new ImportCustomizer();
    ic.addImports( ... );
    cc.addCompilationCustomizers(ic);

    // (4) Establish bindings for variables.
    Binding binding = new Binding();
    binding.setVariable("a", ... );

    // (5) Define a ClassLoader for the script.
    ClassLoader classloader = this.getClass().getClassLoader();

    GroovyShell groovyShell = new GroovyShell(classloader, binding, cc);
    return (BaseClass) groovyShell.evaluate(dslText);
  }

}
```

A Groovy script can be called from other languages. Listing 2.9 shows how a Groovy script can be included in a Java program by using the *GroovyShell*[12] class. GroovyShell allows extensive configuration of the runtime environment of the script. In reference to the numbers in the listen above, some of the supported configuration parameters are:

1. **Script Base Class**: This demonstrates how the base class for the script can be set. By doing so, the *this* pointer inside the script will refer to a newly created instance of the base

[12] http://docs.groovy-lang.org/latest/html/api/groovy/lang/GroovyShell.html

class. This feature is especially important to DSL developers, since it helps define new language keywords as methods and properties of the base class.

2. **AST Transformations**: Groovy exposes the Abstract Syntax Tree to developers for compile-time metaprogramming. AST transformations can also be applied to scripts, which allows DSL developers to bypass some syntax restrictions imposed by Groovy.

3. **Default Imports**: If Groovy or Java classes are used within the script, specific packages can be imported by default.

4. **Variable Bindings**: If certain variables are used within the script, they can be bidirectionally bound to values from the calling program.

5. **Class Loader**: Groovy exposes the same *ClassLoader* mechanism that is used in Java. An instance of a ClassLoader can be specified for the script, which can be used to load classes in a custom way.

Since Groovy scripts do not require any boilerplate code and can be included in other programs, they are ideal candidates for hosting Groovy-based DSLs. An application exposing a DSL interface to its users can require a certain Groovy script file to be present or ask for the path to a file at runtime. The host application can be developed in any language that integrates with Groovy.

CHAPTER 3

Approach

"Two roads diverged in a wood, and I— I took the one less traveled by, and that has made all the difference."
— Robert Frost

A domain-specific language is made to capture the needs of a specific application domain. A well-designed DSL allows developers to express problems from the problem domain with terminology and notions from that domain. A key step in DSL development is selecting the domain-specific notions to use and incorporating them to a DSL, since these notions have a direct impact on developer productivity [53]. To find the relevant terminology, domain-specific notions and features to incorporate to Moola, operation orchestration scenarios and existing approaches were investigated. The findings of this analysis are introduced in this chapter and are then used to derive a concept of how Moola can integrate the required features to its host language, Groovy. Based on this concept, a language specification is developed and applied to the canonical example.

3.1 Feature Analysis

A thorough understanding of the problem domain is required to develop a DSL. This was achieved by taking real-world scenarios from the ARTIST project [4] and from the ATL Transformation Zoo[1]. The scenarios were analyzed for common characteristics such as nature of operations (e.g. validation, transformation), nature of interactions between operations, metamodels used, tools and libraries involved, etc. Furthermore, existing approaches for operation orchestration were analyzed and common features extracted. The result of this analysis was a list of features that a newly created DSL needs to exhibit to be considered a viable alternative to existing approaches. Figure 3.1 shows the outcome of the analysis phase as feature model [13]. A description of the features follows.

[1] http://www.eclipse.org/atl/atlTransformations/

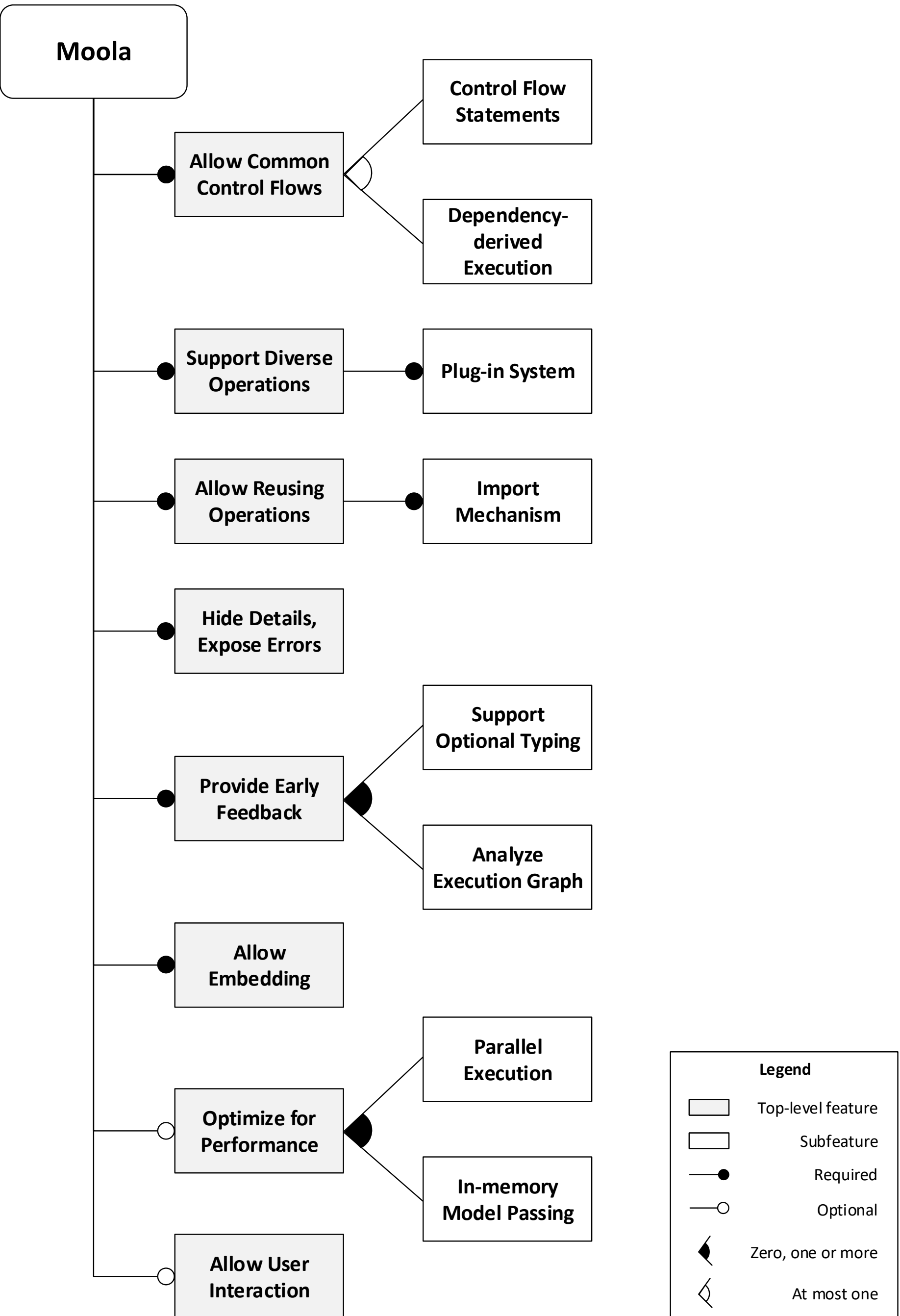

Figure 3.1: Feature Model for Moola

- **Allow Common Control Flows:** Operation chains typically consist of several operations. These can occur in sequence or in parallel. Furthermore, some operations may be optional or may be required to run more than once. A core requirement of any orchestration language is the possibility to represent these common control flow scenarios. To achieve this, two ways were analyzed. One approach is to include dedicated keywords to the language that allows a developer to influence the control flow directly. This is common in imperative programming languages such as Java or Groovy, which use keywords such as *if*, *while* or *for* to control the order of statement execution. Alternatively, dependencies between operations can be used to derive an execution order. This can be done by construction a graph (or similar representation, e.g. a petri net [18]) and using well-known algorithms such as topological sort [58] to derive an execution order.

- **Support Diverse Operations:** Typical operation chains apply more than one operation on the input models, aside from loading and saving models. These operations may be specific to the metamodel of the input models (e.g. validation or transformation) or may be independent of the inner structure of a model (e.g. copying or moving models). An orchestration language needs to account for the diversity of model operations by exposing a mechanism to extend the current set of operations. This is especially important when considering newly appearing transformation languages. A common way to provide such an extension mechanism is by incorporating a plug-in system to the DSL.

- **Allow Reusing Operations:** Many operations are general in a way they can be reused in different operation chains. Such operations should be defined and implemented only once, but later included wherever needed. To support this, an import mechanism for existing operations is required.

- **Hide Operation Details, Expose Execution Errors:** From an orchestration developers' point of view, any operation in a chain can be viewed as black box. Although the source code of the operation might be available to the orchestration developer, a clearly defined interface should be all that is required to invoke the operation. This interface includes a description of the inputs and outputs, as well as a list of possible errors that might occur during operation execution. While the operation internals are hidden during execution, all errors should be exposed to the caller and either allow the orchestration developer to react on them by including error control structures, or be displayed to the user to support easier troubleshooting.

- **Provide Early Feedback About Chain Validity:** The execution of operation chains can take up considerable amounts of time depending on the number and complexity of operations involved. In case runtime errors occur in a later stage of an operation chain, the chain needs to be re-executed after the problematic code was corrected. To avoid unnecessary runtime errors and through this, speed up development, static analysis on the validity of an operation chain should be supported, e.g. by using a type system or analyze the execution graph of the operation chain before running it. A type system can help avoid runtime errors by making sure input and output types of operations are compatible with the respective operation definition. Furthermore, misspelled operation and model names

can be detected. Additionally or alternatively, if the operation chain is present as a graph, performance bottlenecks can be determined and certain characteristics (such as reachability of all operations) evaluated [79]. When problems are encountered, feedback can be given to the user without any operation being executed.

- **Allow Embedding in Other Processes:** Operation chains may be executed within the context of other processes (e.g. as part of a bigger build or in the context of integrated development environments). One way to achieve this is by providing a CLI to the DSL that can be called from any process. By using this approach, a process can indirectly include operation chain execution by running the CLI command and spawning a new operating system process. In contrast, the DSL can exhibit an API that allows direct embedding to processes sitting on the same technology stack as the DSL without spawning a new process just for the operation chain execution. This allows for easier in-memory exchange of complex data between host and child processes, which can be used to either pass data (e.g. models) or status information (log messages).

- *(optional)* **Optimize for Performance:** When using an MDE approach, execution of operation chains is a frequent activity comparable to running a build on conventional software projects. To limit the amount of time spent waiting on results, operation chains should incorporate the following features: *parallel execution* allows independent operations to run at the same time. Operations are independent if there is no specific order in which the operations need to be executed. Such an order is naturally given if one operation operates on the direct or indirect output of another operation. *In-memory model passing* allows passing models between operations without the need to save them to disk. Especially for intermediate results, saving models is unnecessary and should be avoided.

- *(optional)* **Allow User Interaction:** During the execution of operation chains, feedback needs to be provided to the user to show the progress and current state of the execution. In several scenarios, user input can be gathered to change the course of execution (e.g. which operation to perform) or specify settings (e.g. location for model saving).

3.2 Integrating Operation Orchestration to Groovy

To integrate the concepts of model operation orchestration to Groovy while enabling the previously listed features, the similarities between Groovy and operation chains are first highlighted. Operation chains have a control flow defining the order of operation execution. They have a data flow describing how values are passed from one operation to another. In case an operation fails, an alternate operation may be executed or the whole chain aborted, so a way to handle errors is required. In a Groovy script, the control flow and error handling is explicitly defined by using keywords. The data flow can be derived by following variable usage in e.g. assignments and function calls.

Since Groovy has mechanisms to describe control and data flows and error handling within a script and operation chains require similar mechanisms, it is a natural step to try and reuse the keywords and structures of Groovy. This lowers the entry threshold for developers new to

Moola, but familiar with Groovy or Java. To seamlessly integrate the concepts of model-driven engineering to Groovy scripts and thereby reuse the existing Groovy keywords, the first-class citizens of MDE, models and operations, need to be mapped to Groovy concepts appropriately. Table 3.1 shows how Moola maps concepts from MDE to Groovy language constructs.

Moola	**Groovy**	**Correlation**
Model Type	Class	Serves as blueprint for data.
Model	Variable	Represents data for the execution. Can be typed.
Operation	Function	Expects certain input values and returns certain output values. Can be typed. Throws exceptions in case of errors.

Table 3.1: Mapping Between MDE and Groovy concepts.

A *model type* defines the inner structure of a model. A process accessing a valid model of some model type can rely on certain guarantees the model type imposes on the inner structure of the model, such as existing elements and connections between elements. Hence the model type can be seen as a MDE counterpart to what a *class* is in object-oriented programming.

A *model* represents the data of the operation chain. It *conforms to* a model type and serves as input and output of operations. A model typically resides on disc or in memory and needs to be made available to the operation chain by means of a pointer to were the actual data is located. Such pointers to data are represented through variables and properties in object-oriented programming languages such as Groovy.

An *operation* performs an action on one or more input values and yields one or more output values. Both input and output values may be models, or any other type of data, e.g. strings, numbers, etc. Typical operations include validation (which checks the conformity of a model against its model type and returns a boolean value), transformation (which takes one or more input models and yields one or more output models of possibly different model types), etc. Furthermore, an operation may fail and inform its caller on the cause of the failure. A counterpart to the notion of operation is a *function* in object-oriented programming. A function operates on input and yields output values. These values can be model references, thereby relying on the mapping between *model* and *variable*. In case of error, a function may throw an exception.

To tie the approach together, operation chain developers needs to take the following steps to implement an operation chain:

1. **Define Model Types, Models and Operations**: Developers need to specify which model types they intend to use. Similarly to classes, these can later be used to type variables, i.e. models, and input and output values for operations. Developers then need to define which models are going to be used and where to retrieve them from, e.g. from a file. All models will be made available to the orchestration code via Groovy variables. Finally, developers need to define the operations they want to use and their interface, i.e. the type of input and output values. All operations will be made available to the orchestration code as Groovy

functions. Special keywords are introduced in Moola to define model types, models and operations.

2. **Orchestrate Operations**: Developers can use standard Groovy control and error handling keywords to create the operation chain. Additional keywords are introduced for situations in which Groovy does not provide native structures for, e.g. parallel execution. Models are made available via variable references, operations are made available as functions. This allows developers to use Groovy and Java knowledge and best-practices to create the operation chain.

Two-Phase Execution

Moola, like Gradle, operates in a two-phase approach. In the *configuration* phase, the Moola script is parsed and all artifacts (metamodels, models, operations) are collected. This includes reading files, applying plug-ins and resolving operations imported from other Moola scripts. The gathered information is then used to type-check the orchestration code. Since the artifacts are not directly used in the configuration phase, their order of appearance in the Moola script is not relevant. Developers can choose to group artifacts by type (e.g. all models before all operations), by task (e.g. all models and operations of one specific task together) or any other form. After the configuration phase has collected all artifacts and the orchestration code passed type checking, the *execution* phase runs the orchestration code.

This two phase approach has several advantages. Each artifact can be checked for validity (e.g. if all required information is supplied, if the specified files exist, etc.). All artifacts combined can be used to check the validity of the orchestration code. In doing so, common errors can be eliminated without the need to execute any operation or execute any parts of the orchestration code. Errors that can be avoided this way include type mismatches, misspelled model and operation names, missing files, etc.

Classification of Moola

Reusing the Groovy keywords for control flow and error handling places Moola in the class of imperative languages, since the order of operations is explicitly defined. In contrast, the developers of Gradle, which also uses Groovy as host language, choose not to reuse the control flow structures and go for a declarative approach. By sitting on top of Groovy, Moola inherits the full power of Groovy's functional programming capabilities. Since Moola reuses many aspects of Groovy's concrete syntax, it uses a textual syntax to describe operation chains.

Groovy is optionally-typed, meaning that developers can choose to include type declarations. Moola, on the other hand, is a strongly-typed language in which classes and model types can be used alike to type variables, models and operation input and output values.

3.3 Language Specification

An important property of any domain-specific language is the tuning of the syntax to allow for expressive problem solving in the application domain. This section will introduce Moola's

syntax and explain the semantics of the language constructs.

Syntax

The syntax of Moola is described in the following using EBNF. To make the grammar easier to read, the standard EBNF syntax is slightly altered. This includes using the regular expression operations ? (option), + (repetition with minimum 1) and * (repetition with minimum 0) instead of the standard EBNF operations. Non-terminal symbols are written in angled brackets, terminal symbols are written in single quotes (’) and **bold**. The .. operator describes sequences, e.g. ’a’..’z’ refers to all lowercase letters from a (inclusive) to z (inclusive). Since Moola’s syntax sits on top of Groovy’s syntax, non-terminal symbols starting with *groovy-* refer to Groovy language constructs and their original meaning in Groovy and are, for the sake of scope, not elaborated in detail in this grammar.

⟨*program*⟩ ::= (⟨*include*⟩ | ⟨*plugins*⟩ | ⟨*definition*⟩)* ⟨*run*⟩?;

⟨*include*⟩ ::= **’from’** ⟨*path*⟩ **’include’** ⟨*operation-list*⟩+;

⟨*path*⟩ ::= ⟨*groovy-g-string*⟩;

⟨*operation-list*⟩ ::= ⟨*operation-name*⟩ (**’,’** ⟨*operation-name*⟩)+;

⟨*plugins*⟩ ::= **’plugins’** ⟨*plugin-list*⟩;

⟨*plugin-list*⟩ ::= ⟨*plugin-name*⟩ (**’,’** ⟨*plugin-name*⟩)*;

⟨*plugin-name*⟩ ::= **”** (⟨*alphanum*⟩ | **’_’**)+ **”**;

⟨*alphanum*⟩ ::= **’a’..’z’** | **’A’..’Z’** | **’0’..’9’**;

⟨*definition*⟩ ::= ⟨*modeltypes*⟩ | ⟨*model*⟩ | ⟨*operation*⟩;

⟨*modeltypes*⟩ ::= **’modeltypes (’** ⟨*modeltype-list*⟩* **’)’**;

⟨*modeltype-list*⟩ ::= ⟨*modeltype-def*⟩ (**’,’** ⟨*modeltype-def*⟩)+;

⟨*modeltype-def*⟩ ::= ⟨*modeltype-name*⟩ **’:’** ⟨*path*⟩;

⟨*modeltype-name*⟩ ::= **’A’..’Z’** (⟨*alphanum*⟩ | **’_’**)+;

⟨*model*⟩ ::= **’model’** ⟨*model-name*⟩ **’{’** ⟨*model-details*⟩ **’}’**;

⟨*model-name*⟩ ::= (⟨*alphanum*⟩ | **’_’**)+;

⟨*model-details*⟩ ::= ⟨*groovy-statements*⟩+;

⟨*operation*⟩ ::= **’operation’** ⟨*operation-name*⟩ ⟨*operation-params*⟩? ⟨*operation-settings*⟩?;

⟨*operation-name*⟩ ::= (⟨*alphanum*⟩ | **'_'**)+;

⟨*operation-params*⟩ ::= **'('** ⟨*groovy-named-params*⟩ **')'**;

⟨*operation-settings*⟩ ::= **'{'** (⟨*groovy-statement*⟩ | ⟨*expects*⟩ | ⟨*returns*⟩)+ **'}'**;

⟨*expects*⟩ ::= **'expects'** ⟨*param-def*⟩ (**','** ⟨*param-def*⟩)+;

⟨*returns*⟩ ::= **'returns'** ⟨*param-def*⟩ (**','** ⟨*param-def*⟩)+;

⟨*param-def*⟩ ::= ⟨*model-name*⟩ **':'** (⟨*modeltype-name*⟩ | ⟨*groovy-class*⟩);

⟨*run*⟩ ::= **'run {'** ⟨*orchestration-code*⟩ **'}'**

⟨*orchestration-code*⟩ ::= (⟨*groovy-statement*⟩ | ⟨*parallel*⟩ | ⟨*await*⟩)*;

⟨*parallel*⟩ ::= (⟨*groovy-variable*⟩ **'='** | **'await'**)? **'parallel ('** ⟨*closure-list*⟩ **')'**;

⟨*closure-list*⟩ ::= ⟨*groovy-closure*⟩ (**','** ⟨*groovy-closure*⟩)+;

⟨*await*⟩ ::= **'await'** ⟨*groovy-variable*⟩;

Semantics

To tailor Groovy to the domain of model operation orchestration, several new keywords and language constructs are introduced. Their syntactical outline and where they can be placed within a Moola script can be taken from the EBNF of the previous section. In this section, their meaning and how they map to the aforementioned requirements is explained.

- **modeltypes** This keyword allows the definition of model types. Each model type consists of a name and a path to a metamodel. In subsequent parts of the Moola script, model types are used to type models as well as input and output parameters of operations. Through the model type mechanism, strong typing is supported in Moola and rich checks for validity can be applied to Moola scripts before their execution. A model type can be used interchangeably with a Groovy class inside a Moola script (and is, indeed, transformed to an internal class as will be shown in Chapter 4).
- **model** The *model* keyword allows to define models, which will be made available to the orchestration code later on. The model definition includes the corresponding model type and a place to load the model from, typically some file on the disc or some other resource.
- **operation** A model operation can be implemented using various underlying techniques. For example, domain-specific languages exist for various model-centered operations such as transformations, validations, etc. Other operations may include calling third-party programs (e.g. the Java compiler) or programs written in a general-purpose language (e.g. Java). The means to implement operations are manifold. To include such operations

of varying underlying implementation in an orchestration chain, an operation definition is needed. It describes which inputs an operation expects, which outputs it returns and which other settings need to be applied to successfully execute it. In Moola, the *operation* keyword is used to describe an operation and make it available to the orchestration code.

- **expects**, **returns** These two keywords can only occur in an *operation* definition and are used to describe the operation's interface. The *expects* keyword describes the input values, while *returns* describes the output values. Each input and output value definition consists of a name and type. All valid Groovy types can be used, including primitive types and classes. If the operation receives or yields models, model types can be used to specify their type.

- **from .. include ..** Before model operations can be orchestrated to an operation chain, the interface of the operation must be known to Moola. Also, any details on how to invoke an operation (e.g. settings) need to be specified before a call to the operation can be successful. This can happen through either writing an operation definition directly in the orchestration file via the *operation* keyword, or by including an orchestration definition from another Moola file. Through this mechanism, operations can be defined in one file and included in several operation chains. This allows reusing operations and makes the task of including an operation to an operation chain easier, since a Moola file describing a certain operation can be added to the implementation of the operation.

- **plugins** A plugin is a set of *operation types* that can be made available to an operation chain. After the *plugins* keyword, a list of plugins can be specified. When an operation is defined, the type of the operation needs to be specified. This operation type defines which underlying language was used to implement the operation and therefore governs which settings must be specified in order to call the operation. Furthermore, input and output model types depend on the operation type, although this may vary or even be dynamic depending on the operation implementation.

- **parallel** The *parallel* keyword integrates concurrent execution of one or more operations directly into Moola. It expects a list of Groovy closures, which will be called concurrently via separate threads. When called, *parallel* returns a promise object. The main thread continues execution without waiting on the *parallel* threads to finish. If the main thread should be blocked until all parallel threads have finished, the *await* keyword can be used on the promise returned by *parallel*. More details on concurrency and how locking and error handling are implemented in Moola can be found in Chapter 4.

- **await** This keyword expects the promise returned by an invocation of *parallel* as first argument. It then blocks the current thread until all child threads started by the specific parallel invocation have finished execution. If an exception occurs in one of the child threads, *await* will throw the same exception, making it possible to react on exceptions in child threads. If more than one child threads fail due to an exception, a specific exception will be thrown indicating multiple errors. Since *await* blocks the current thread, it can be used within *parallel* to allow for nested concurrent execution.

- **run** The *run* keyword is followed by the *run closure* and marks the part of the Moola script responsible for the orchestration code. All defined models are available as variables within the run closure. Similarly, all defined or imported operations are available as functions within the run closure. Since Moola in an imperative programming language, the run closure is executed top-down, one statement at a time (except for concurrent execution via *parallel*). Any valid Groovy code can be used within the run closure to orchestrate the operations, including Groovy's built-in functions and functional programming concepts.

Aside from the newly introduced keywords, all Groovy keywords retain their meaning. This is especially important for the control flow keywords (*if*, *switch*, *while*, *for*) and error handling keywords (*try*, *catch*, *finally*).

An Implementation of the Canonical Example

Summarizing the syntax and semantics of Moola, Listing 3.1 shows one possible orchestration for the canonical example (Figure 1.2) introduced in Chapter 1. It includes several other files via the *from-include* keyword pair, which themselves contain the operation definitions. One such file is shown in Listing 3.2, which contains the model operations for the SQL part of the canonical example.

Listing 3.1: Orchestration of the Canonical Example.

```
from "./java" include GEN_JAVA, BUILD_JAVA
from "./sql"  include GEN_DB_MODEL, GEN_SQL
from "./misc" include PACKAGE
from "./repo" include COMMIT

plugins "EMF"


// ---------- Model Types and Models ---------- //

modeltypes (
  Uml: "http://www.eclipse.org/uml2/5.0.0/UML",
  DbModel: "../meta/dbmodel.ecore"
)

model domainModel (
  type: Uml,
  path: "./models/domain.uml"
)


// ---------- Orchestration ------------------- //

run {
  if ( !VALIDATE( domainModel ) ) {
    error "Domain␣model␣is␣not␣valid."
    exit
  }

  await parallel ({
      GEN_JAVA( domainModel, "./build/java" )
      BUILD_JAVA( "./build/java" )
```

```
    }, {
      sqlModel = GEN_DB_MODEL( domainModel )
      GEN_SQL( sqlModel, "./build/sql" )
    })

  zipPath = PACKAGE( "./build/java/**/*.class", "./build/sql" )

  errorCount = 0
  while ( errorCount < 3 ) {
    try {
      username = ask "Username:␣"
      password = ask "Password:␣"

      COMMIT( username, password, zipPath )
    } catch ( AuthenticationError ex ) {
      info "Username/Password␣incorrect,␣please␣try␣again."
      errorCount++
    } catch ( Exception ex ) {
      error ex
      exit
    }
  }
}
```

Listing 3.2: SQL Operation Definitions of the Canonical Example.

```
plugins "EMF"


// ---------- Model Types ---------- //

modeltypes (
  Uml: "http://www.eclipse.org/uml2/5.0.0/UML",
  DbModel: "../meta/dbmodel.ecore"
)


// ---------- Operations ----------- //

operation GEN_DB_MODEL( type: "ATL/EMFTVM" ) {
  expects input: Uml
  returns output: DbModel
  path = "."
  module = "SqlToDBModel"
}

operation GEN_SQL( type: "Acceleo" ) {
  expects input: DbModel, path: String
  project = "GenSQL"
  main = "GenSQL.common.Generate"
}
```

The Moola script in Listing 3.1 starts with importing all required operations from other Moola files. It then registers the *EMF* plug-in, which is needed to gain access to the operation types of the EMF world and some standard operations (such as *VALIDATE* in this example). Afterwards, the model types *UML* and *DBModel* are defined by specifying the locations of their corresponding metamodels. In the case of the *UML* model type, a namespace is provided which is used to resolve the metamodel. The *DBModel* model type on the other hand directly points to

the file holding the metamodel in the *Ecore* format. The model type definitions are followed by the definition of the only used model, *domainModel*, through specifying its model type and file location. The orchestration code within the *run closure* orchestrates the operations by mainly using standard Groovy keywords (*if*, *while* and *try ... catch ...*) and *await* and *parallel* for concurrent execution. As was described previously in this chapter, all imported operations are available as functions within the orchestration code and can be called accordingly. Similarly, all defined models are available as variables and can be used as such.

Aside from operations, several standard functions are available in the orchestration code to cover common aspects of operation chain development. These include functions for printing text to the console on various severity levels (*verbose*, *info*, *warn*, *error*), prompting users to provide input (*ask*) and stopping the current execution (*exit*).

3.4 Relation to Activity Diagrams

Activity diagrams [24] are a general-purpose notation to describe workflows within a system [17, 19] and were shown to be effective in, although not universally applicable to, various domains, including business process modeling [63]. The resemblance of operation chains and other types of software development workflows was already noted in Chapter 1. It thus follows that general workflow notations, such as activity diagrams, can also be applied to the domain of MDE and be used to describe operation chains [83].

The canonical example in Figure 1.2 and its implementation in Moola in Listing 3.1 indicate a relation between activity diagrams and Moola programs. Such a mapping from activity diagrams to a programming language called Esteral was shown to exist [7]. In this section, a similar mapping from activity diagrams to Moola is presented by demonstrating how the most important elements and control flow patterns of activity diagrams can be expressed in Moola.

Activity Diagram Element	**Moola Equivalent**
Activity	Model operation as function call in *run closure*
Initial state	Beginning of *run closure*
Final state	End of *run closure*, **exit** function

Table 3.2: Activity diagram elements and their Moola equivalent.

Table 3.2 shows three core elements in every activity diagram and how they relate to Moola. An activity in an activity diagram is a model operation in Moola. Referring back to the mapping of MDE and Moola in Table 3.1, model operations are present as functions within the *run closure*. Start and end states of an activity diagram are explicitly marked by elements. These are implicitly defined by the beginning respectively the end of the *run closure* in Moola. Additionally, the orchestration of Moola can exit from anywhere using the globally available *exit* function. Table 3.2 shows that activity diagram elements relate to concepts in and around the *run closure*, which intuitively makes sense, since activity diagrams describe workflows, which

map to the orchestration code of operation chains, which itself fully resides in the *run closure*. Other Moola code deals with setting up the model operations and execution environment and is therefore not reflected when using a workflow-centered notation like activity diagrams.

Control Flow Pattern	**Activity Diagram**	**Moola Code**
Sequence	A B	`A()` `B()`

Choice	cond A else B	`if (cond) {`
		`  A()`
		`} else {`
		`  B()`
		`}`

Parallel Split	A B	`parallel ({`
		`  A()`
		`}, {`
		`  B()`
		`})`

Synchronization	A	`await ...` `A()`

Iteration	cond A else	`while (cond) {`
		`  A()`
		`}`

Table 3.3: Control flow patterns and their representation in activity diagrams and Moola code.

Table 3.3 shows common control flow patterns and how they can be modeled in activity diagrams using *decisions*, *splits* and *joins*, and one way to write them in Moola. Other Moola implementations are feasible, e.g. Moola's *switch* statement could be used instead of the proposed *if* statement for choices, a *for* loop can be used instead of *while* for iterations. Looking at the Moola code in Table 3.3 highlights the aspired similarity to Java and Groovy, which is

achieved by re-using keywords from these languages in Moola.

Aside from describing the control flow, activity diagrams can also model the object flow between activities. Object flows can be used to explicitly describe data exchange in an activity diagram. In Moola, passing data into model operations and receiving data from model operations is implemented by using function parameters respectively function return values of the operation function in the *run closure*. Like Java and Groovy, Moola has several variable scopes that control the visibility of variables. The top-most scope, i.e. the global scope, can be used to exchange data between any (even independent) operations and can thus be used to implement any object flow that can be modeled in activity diagrams.

CHAPTER 4

Implementation

"I love deadlines. I like the whooshing sound they make as they fly by."

— Douglas Adams

Embedding a DSL on top of a host language can be realized in different ways. Groovy has concepts for DSL development integrated as core parts of the language. In Chapter 2, relevant mechanisms were introduced to provide a conceptual framework on DSL development in Groovy. After describing Moola's approach on how to incorporate the terminology and domain-specific notions of model operation orchestration to Groovy in the previous chapter, this chapter introduces the implementation details of Moola and how the **eleven** new keywords are realized.

One of the most important bits of code in DSL development is the definition of a proper Groovy script base class. An instance of the base class is used as *this* pointer within the Groovy script and is therefore called when e.g. a variable cannot be resolved. In this case, the instance of the base class will be searched for a property of the variable name. Similarly, if a method cannot be resolved in the Groovy script, the instance of the base class is searched for a method with that name. By adding a method to the base class, a keyword-like structure can be introduced to the DSL. Listings 4.1 and 4.2 show this concept on an excerpt of the *ScriptBase*[1] class of Moola.

Listing 4.1: Excerpt of the base class for Moola.

```
import groovy.lang.Script

class ScriptBase extends Script {

  void modeltypes(values){
    // ...
  }

  void model(args) {
    // ...
```

[1] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/phases/config/ScriptBase.groovy

```
  }

  void operation(args, @DelegatesTo(Operation) configClosure) {
    // ...
  }

}
```

Listing 4.2: Calling the newly added keywords from within the script.

```
// Calling the base class methods implicitly.
modeltypes (...)
model (...)
operation (...)

// Or explicitely via the this pointer.
this.modeltypes (...)
this.model (...)
this.operation (...)
```

As the listings show, the methods of the script base class can be called from within the script by either calling them with or without the *this* pointer. By adding syntax highlighting to the editor and using the methods without *this* pointer, keyword-like structures can be introduced. These can be used to manipulate the state of the instance of the base class or execute certain functions directly.

Keyword	**Defined On**
modeltypes	ScriptBase
model	ScriptBase
operation	ScriptBase
expects	Operation
returns	Operation
plugins	ScriptBase
from	ScriptBase
include	OperationImporter
run	ScriptBase
parallel	ExecutionContext
await	ExecutionContext

Table 4.1: Keywords and the classes they are defined on.

This also shows the importance of a properly defined script base class. Aside from abstract syntax tree (AST) transformations, nearly all interactions between the Groovy script holding the DSL code and the process executing the code (e.g. CLI or other JVM process) happen over the script base class. Table 4.1 shows all eleven Moola keywords and the classes they are defined on as methods. Most keywords are defined on the ScriptBase. Two more, *expects* and *returns* are used to set up an operation and are therefore defined on the Operation class. Lastly, two keywords are used to facilitate parallel execution and are defined on a special class called

ExecutionContext. In the remainder of this chapter, details are described on how all keywords are implemented.

4.1 Model Types

Model types allow users to specify which metamodels are used in the operation chain. Once a model type is defined, it can be used to type models and operations (inputs and outputs) and works similarly to traditional Groovy classes and interfaces when typing is concerned. Like most keywords, the *modeltypes* keyword is a method on Moola's ScriptBase class. It accepts a dictionary of key-value pairs, which in Groovy can be passed in a named-parameter style of syntax. Listing 4.3 shows how the keyword can be used.

Listing 4.3: Defining model types.

```
modeltypes (
  Uml: "path/to/a/metamodel/file",
  SysML: "other/path/to/metamodel/file",
  ...
)
```

To allow using model types similarly to Groovy classes, an AST transformation is used on the *modeltypes* keyword that creates a real Groovy class for every dictionary item. After Groovy has parsed the Groovy script holding the DSL code and has generated the preliminary AST, a transformation checks the AST for model type definitions and adds nodes to the AST that define a Groovy class for every definition it encounters. The AST transformation is illustrated in Figure 4.1. The code can be found in the *GenerateModelClassesTransformer*[2] class.

The AST transformation checks for *MapEntry* items in the argument list of the *modeltypes* method call and adds a *ClassNode* for every map entry it finds. This creates inline Groovy classes in the script, which can be used to type variables and methods. From a programming point of view, these classes are identical to other Groovy classes, which means that they can be used just as any other class, e.g. in definitions, cast expressions, etc.

4.2 Models

Models are a first-class citizens of any MDE-related activity. In Moola, a model is defined using the corresponding *model* keyword and later made available to the orchestration code as variable. An example model definition can be seen in Listing 4.4.

Listing 4.4: Defining a model.

```
// Call with name as parameter.
model ( name: "domainModel", type: Uml, path: "path/to/model/here" )

// Call with name in front of parameter list.
model domainModel ( type: Uml, path: "path/to/model/here" )
```

The *model* keyword is, again, implemented as method on the ScriptBase class. It expects a named-parameter list (similar to the *modeltypes* keyword). Mandatory items are *name*, *type*

[2] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/dsl/ast/GenerateModelClassesTransformer.groovy

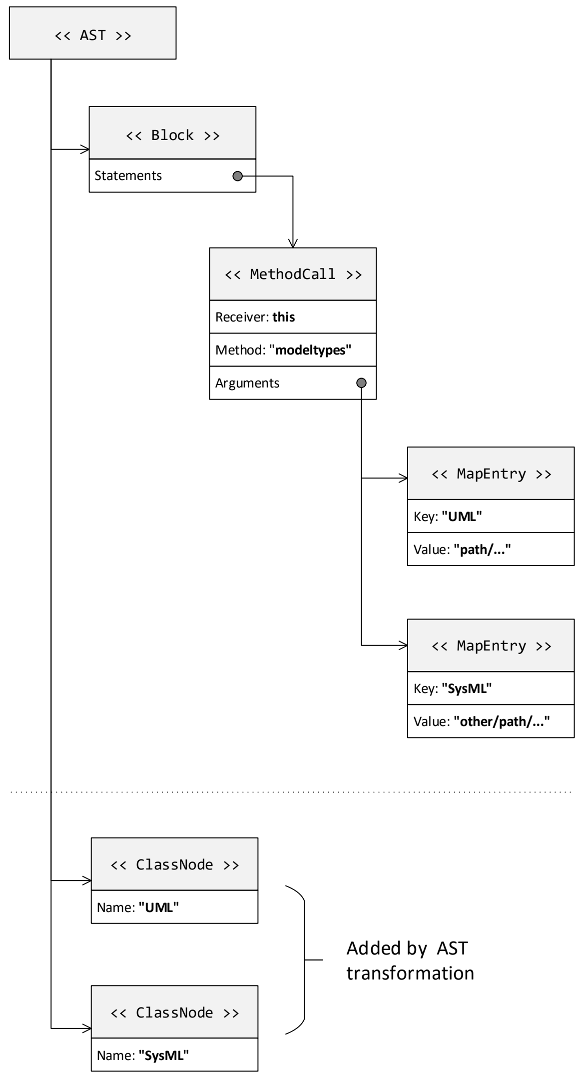

Figure 4.1: AST before and after applying the transformation.

and *path*. The *type* parameter takes a model type (or any other Groovy class) and is used for typing the model. Since the *model* keyword defines exactly one model, a more intuitive notation can be used by pulling the model name in front of the parameter list. This syntax is achieved by an AST transformation. Figure 4.2 depicts the syntax trees for the two alternatives to use the *model* keyword. The left AST is constructed when the *name* parameters is written in front of the parameter list. The parser interprets the model name („domainModel" in our example) as method call and parses an according syntax tree. From the parsers point of view, the *model* method is called with the return value of the *„domainModel"* method call. In comparison, when the *name* is specified within the parameter list, it is also included in the arguments of the *model*

method call.

Although the left AST is valid in terms of Groovy syntax, it cannot be executed since no method of the given name, e.g. „domainModel“, exists. To allow the notation of a prefixed model name, an AST transformation is required. Whenever Moola encounters an AST having a method call as first argument of the *model* keyword, this method call is removed and the name of the method is passed along the other parameters into the *model* keyword. This basically transforms an AST as seen in the left of Figure 4.2 to one as seen in the right of same figure. The code of this transformation can be found in the *ModelNameTransformation*[3] class.

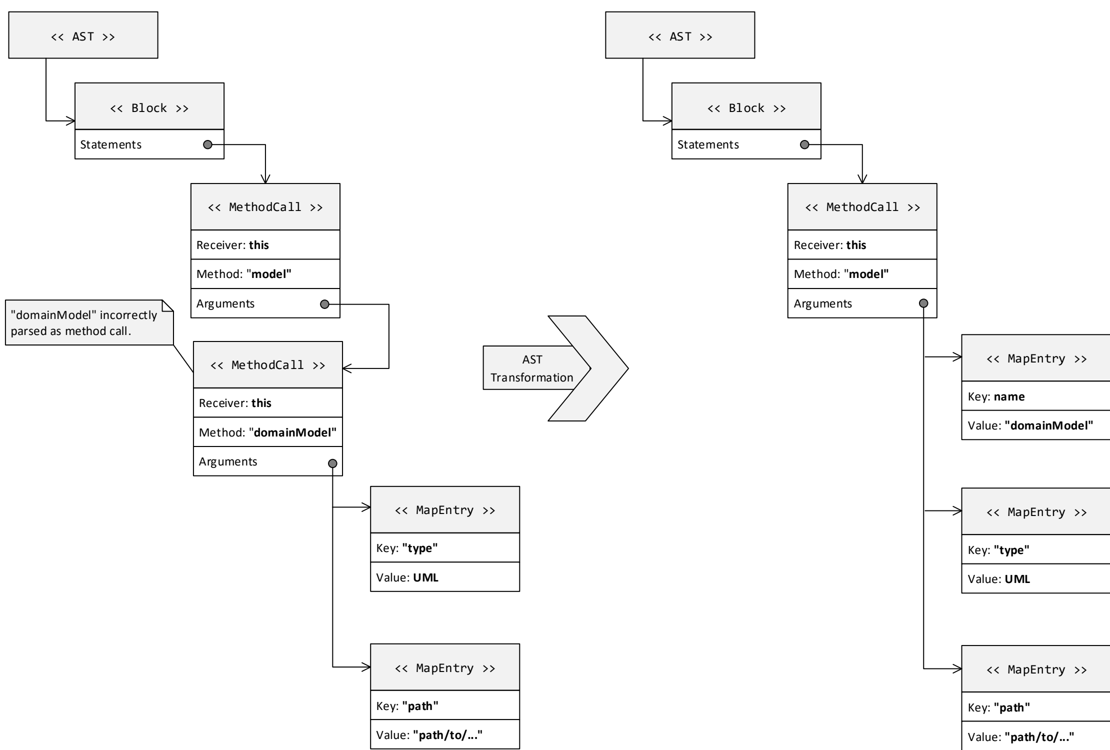

Figure 4.2: *(left)* AST when the name is prefixed. *(right)* AST when the name is included in the parameters. Since the left AST is not executable by Moola, a transformation is used to convert it to the right one.

4.3 Operations

The *operation* keyword can be used to define model operations. Similarly to *modeltypes* and *model*, it is defined as method on the ScriptBase. As with model definitions, an operation definition can hold the name of the operation either directly in the parameter list or in front of it. In

[3] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/dsl/ast/ModelNameTransformation.groovy

the latter case, an AST transformation similar to the one used with the *model* keyword is used to form an AST that Moola can execute. The *OperationNameTransformation*[4] class holds the code for this AST transformation. Listing 4.5 shows two ways of how to use the *operation* keyword. The second call to *operation* not only places the operation name in front of the parameter list, but also pulls the (configuration) closure parameter out of the parameter list. As described in Chapter 2, the last parameter of any method call can be placed after the call if it is a Groovy closure.

Listing 4.5: Defining an operation.

```
// Call with name and closure as parameters.
operation ( name: "COMMIT", type: "Commit", {
  expects input: Uml, message: String
  returns success: Boolean
  timeout = 30000
})

// Call with name in front of parameter list and trailing closure.
operation COMMIT ( type: "Commit" ) {
  expects input: Uml, message: String
  returns success: Boolean
  timeout = 30000
}
```

Two more Moola keywords can be used when defining an operation: *expects* and *returns*. These two keywords define the input and output parameters of the operation respectively. In contrast to previously mentioned keywords, the *expects* and *returns* keywords are defined as methods on the *Operation*[5] class. The closure specified as parameter to the *operation* keyword is not executed on the (topmost) Groovy script scope, but rather on the scope of an operation instance. To put it in another way, the *this* pointer within the closure points to an instance of an operation rather than to an instance of the ScriptBase. By doing so, the two keywords can manipulate the state of the operation they are defined on. A named-parameters list is used to specify the names and types of inputs and outputs. Any Groovy class (and therefore also model types) can be used as value. To safe on boilerplate syntax, the parenthesis around the parameters for the *expects* and *returns* method calls can be omitted.

An important parameter of an operation definition is the *type*. It defines the behavior of the operation when it is executed. When Moola encounters an operation definition, it searches the *OperationRegistry*[6] of the current Moola process for the provided type. Each type is realized by a subclass of the *Operation* class. Moola creates a new instance of the found class and runs the configuration closure in the context of this new instance. In doing so, the closure has direct access to the operation and can manipulate internal values, such as inputs and outputs. Furthermore, it can access any properties or call any methods that are definied on the *Operation* class or the specific subclass of the chosen operation type. In Listing 4.5, this is done to set the *timeout* property. The timeout is specific to the Commit operation and is an integer property on the given subclass of *Operation*.

[4] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/dsl/ast/OperationNameTransformation.gro

[5] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/dsl/core/Operation.groovy

[6] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/dsl/core/impl/OperationRegistry.java

During the execution phase, all defined or imported operations are made available to the orchestration code as methods. To give operations a prominent outline in the orchestration code, the Moola style guide recommends writing operation names in all upper case letters.

4.4 Plug-ins

The Moola core language provides the capabilities to orchestrate operations from any MDE framework. The type of action performed by an operation depends on the operation type specified in the operation definition. These types can be registered to the current Moola process via plug-ins. Listing 4.6 shows how several plug-ins can be added in one statement.

Listing 4.6: Adding plug-ins.

```
plugins "EMF", "File", "SVN"
```

The *plugins* keyword is a method on the ScriptBase that expects a variable number of String parameters. It searches the registered plug-ins for the names provided and adds the corresponding plug-ins to the current Moola process.

4.5 From ... Include ...

A Moola file typically holds four distinct elements: modeltype definitions, model definitions, operation definitions and the orchestration code. Operation definitions create an instance of an operation type by specifying input and output types and settings. To allow reusing operation definitions in various Moola files, the *from* and *include* keywords can be used. These allow importing existing operation definitions from other Moola files to the current one. Listing 4.7 shows how to use the *from ... include ...* keyword pair.

Listing 4.7: Including operations from other files.

```
from "./path/to/file" include SOME_OPERATION, OTHER_OPERATION
```

The included operations can be used in the orchestration code like any locally defined operation. The *from* keyword is implemented as method on the ScriptBase, which returns an instance of *OperationImporter*[7], which defines an *include* method. Listing 4.8 shows the usage of the *from-include* keyword pair if the parenthesis are not omitted.

Listing 4.8: From-Include with parenthesis.

```
from("./path/to/file").include(SOME_OPERATION, OTHER_OPERATION)
```

As last step to realize the *from-include* keyword pair, an AST transformation is required to process the parameters of the *include* method call. When parsing the Groovy scripts of Listing 4.7 or 4.8 (both lead to the same AST), the parameters are understood to be variables. This is correct from a compiler's point of view, however for Moola, the parameters refer to the names of the operation definitions and should be therefore considered as strings. The AST transformation therefore takes each occurance of a variable inside the *include* method call's parameter list and replaces it with a String constant.

[7] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/phases/config/OperationImporter.groovy

Listing 4.9: From-Include from the user's and the compiler's point of view.

```
// Import as defined by the user.
from "./path/to/file" include SOME_OPERATION, OTHER_OPERATION

// Import rewritten with parenthesis and dots.
from("./path/to/file").include(SOME_OPERATION, OTHER_OPERATION)

// Actually executed import after AST transformation.
from("./path/to/file").include("SOME_OPERATION", "OTHER_OPERATION")
```

Listing 4.9 shows the usage of *from-include* as suggested with omitting dots and parenthesis, the call after adding both, and the resulting call after the AST transformation is applied. Figure 4.3 illustrates the AST transformation.

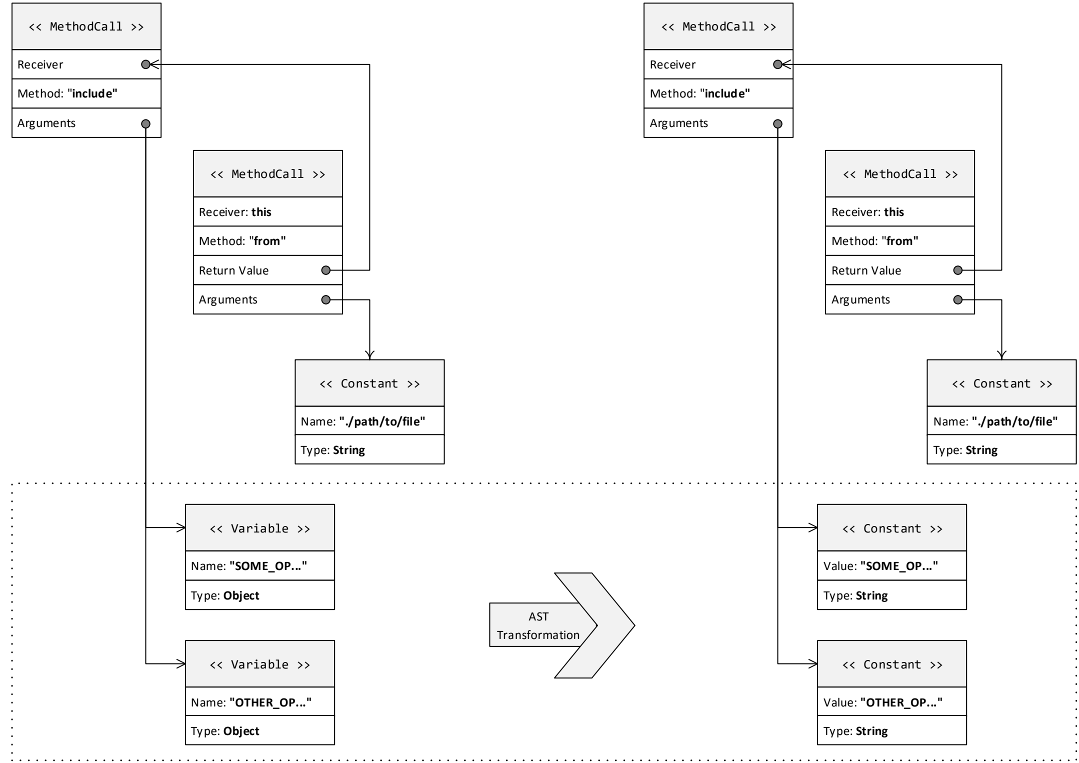

Figure 4.3: AST transformation for the *from-include* keyword pair.

4.6 Orchestration Code

The code describing the orchestration is specified via the *run* keyword. This keyword is implemented as method on the ScriptBase and expects a closure as only argument. Listing 4.10 shows the outline of how to use the keyword.

Listing 4.10: Specifying the orchestration code.

```
run {
  // Orchestration code here.
}
```

The closure passed to the *run* keyword is called *run closure* and can access models and operations as variables and functions respectively. It can use standard Groovy code to orchestrate the operations (which are present as regular Groovy functions). The *run closure* is not executed in the topmost Groovy script scope, but rather in a specific *ExecutionContext*[8]. How this ExecutionContext is constructed and used for type checking is explained later in this chapter.

4.7 Parallel Execution

The *run closure* can use arbitrary Groovy code to orchestrate operations. This especially includes control flow structures such as *if*, *for*, *while*, *switch*, *try ... catch ...*, etc. To achieve concurrent execution of code, Groovy relies on threads. Since parallel execution of operations is a common requirement in operation chains, Moola introduces the keywords *parallel* and *await*. Both are realized as methods on the ExecutionContext (since an instance of this class is the *this* pointer within the *run closure*). Listing 4.11 demonstrates the usage of *parallel* and *await*.

Listing 4.11: Using *parallel* and *await*.

```
run {
  // Start several threads and deferred wait.
  promise = parallel ({
    ...
  }, {
    ...
  })
  await promise

  // Immediately wait on all threads to finish.
  await parallel ({
    ...
  }, {
    ...
  })
}
```

The *parallel* keyword takes a list of Groovy closures and returns a promise [48]. Each closure is executed in a separate thread and has access to the models and operations just as any other place within the *run closure*. The *await* keyword takes the promise returned by a *parallel* invocation and blocks the current thread until all threads started by *parallel* have finished. It can be used to defer waiting for the threads if several *parallel* keywords are used, e.g. in a loop. If the current thread should be blocked directly at the *parallel* call, the call can be prefixed with *await*.

[8] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/phases/exec/ExecutionContext.groovy

Exception Handling

Since each thread has its own execution path, exceptions raised in a thread do not pervade to the main thread. The default behavior of Groovy is to print unhandled thread exceptions to system error and stop execution of the thread. Since the *parallel* keyword starts a new thread for each provided closure, exceptions would not be passed to the main thread. To circumvent this, exception handlers are attached to all threads. If any thread raises an exception, *await* raises the same one. If multiple exceptions occur, *await* raises a *ParallelExecutionException*[9]. Listing 4.12 shows how *await* can be used to handle exceptions that occured during parallel execution.

Listing 4.12: Exception handling with *parallel* and *await*.

```
run {

  promise = parallel ({
    throw new RuntimeException()
  }, {
    ...
  })

  try {
    await promise // Forwards exceptions from within parallel.
  } catch(RuntimeException e) {
    ...
  }
}
```

Concurrency Control and Deadlock Prevention

Since, through *parallel*, several threads may be active at the same point in time and some of them may access the same variables, valid concurrent access to shared values needs to be ensured. In Moola, a pessimistic locking strategy is applied [21]. The locking is done on operation level. Developers of an operation can choose which inputs they want to lock by using the standard Groovy *synchronized* keyword. Similarly, the *synchronized* keyword can be used anywhere in the orchestration code to prevent concurrent access, although this approach is discouraged. Since the manipulation of values should be done within operations, operation-level locking is sufficient.

Although this strategy prevents concurrent access issues, it opens the door to deadlocks [57]. For a deadlock to occur, four conditions identified by Coffman [36] need to be fulfilled:

- **Mutual Exclusion**: The resources cannot be shared between the threads, but are fully claimed by one thread.
- **Hold and Wait**: A thread holding a lock for a resource continues to hold the lock while waiting for additional locks on other resources.
- **No Preemption**: The lock on a resource can only be released by the thread holding the lock. Releasing a lock cannot be initiated by other threads.

[9] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/phases/exec/ParallelExecutionException.g

- **Circular Wait**: A cycle of threads exists, in which each thread waits on resources locked by another thread.

In Moola, a deadlock prevention strategy [16] is used. Deadlock prevention refers to the action of ensuring one of the above conditions to always be false in the system, thus preventing a deadlock from occurring. Moola achieves deadlock prevention by introducing a total order over the resources to always invalidate the *Circular Wait* condition. By introducing an order over the resources, all threads are forced to lock the resources in the same sequence, thus preventing them from forming a cycle of dependent threads.

Moola realizes this total order through the *LockFactory*[10] class. Instead of locking the input values directly, a thread can request the locks for all its resources via the *LockFactory* class. A list of locks is returned, which the thread needs to acquire in the order it receives from the *LockFactory.getLocks* call. The *LockFactory* ensures the same order of locks over all calls independent of which thread executes the call, thus creating a total order of all resources over all threads. If the orchestration code needs to lock certain resources directly within the *run closure*, the *LockFactory* class also needs to be called. Although orchestration code can access the *LockFactory*, the behavior is discouraged since all actions on models and other resources should be performed through operations.

4.8 Type Checking

The first step on how typing is achieved in Moola was already introduced when describing the *modeltypes* keyword. Each model type definition leads to the creation of an inline Groovy class, which can be used to type models and operation in- and outputs. Before type checking works in Moola scripts, a final step needs to be taken: unrolling multi-assignments.

Unrolling Multi-Assignments

In Moola, a model operation can return more than one value. In Groovy on the other hand, each function has either no return value (*void*) or exactly one. When a list is returned from a Groovy function, a multi-assignment syntax can be used to achieve the impression of multiple return values. Listing 4.13 shows a function return a list of values and how a call to this function can use a multi-assignment syntax to assign the list to individual variables.

Listing 4.13: Multi-Assignment syntax in Groovy.

```
def multiReturn() {
    return ["value", 1, 42.0]
}

// Call with untyped variables.
def (a, b, c) = multiReturn()
println a // "value"
println b // 1
println c // 42.0
```

[10] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/dsl/core/LockFactory.groovy

```
// Call with typed variables.
String s; Integer i; Float f

(s, i, f) = multiReturn()
println s // "value"
println i // 1
println f // 42.0
```

When adding type checking to Moola, a return type has to be given to all methods so that Groovy can detect type mismatches on method calls. If no concrete type is specified by using the *def* keyword, Groovy will assume a return value of type *Object*. When returning several values of different types (which may not even share a common base class aside from *Object*), the only allowed return value is *List<Object>*. This, however, leads to typing problems, since a list of objects cannot be multi-assigned to several variables of concrete types. In fact, Groovy completely disallows multi-assignments when type checking is activated and raises a compile-time error when a multi-assignment is found.

Since it is a common trait for model operations to return more than one result, Moola circumvents this issue by adding an AST transformation that replaces all multi-assignments with regular assignments when the right side of the multi-assignment is an operation call. Listing 4.14 shows the call to an operation and how it is translated by the AST transformation.

Listing 4.14: Unrolled multi-assignments in Moola.

```
operation X() {
  returns x: String , y: Integer
}

run {
  String a; Integer b // (1)

  // Call to operation using multi-assignment.
  (a, b) = X()

  // Replaced by AST transformation with:
  OperationResult temp = X()
  a = temp.value(0) as String  // (2)
  b = temp.value(1) as Integer // (2)
}
```

Instead of returning a *List<Object>* from operations, an instance of the *OperationResult* class is returned. This class allows accessing the concrete result values via a method call. The mutli-assignment is then replaced with a separate assignment for each returned value. The two important parts are marked with (1) and (2) in Listing 4.14. At position (1), the orchestration developer specifies the variables (and their types) used to store the return values. At position (2), each return value is cast to the type specified in the orchestration definition. When a type mismatch occurs between operation definition and variable declaration, a type error is raised by the type checker. Listing 4.15 shows such a situation. The operation defines two *String* return values, but the variables used to store the results are defined with type *Integer*.

Listing 4.15: Mismatch in operation definition and variable declaration.

```
operation X() {
  returns x: String , y: String
}
```

```
run {
  Integer a; Integer b

  // Call to operation using multi-assignment.
  (a, b) = X()

  // Replaced by AST transformation with:
  OperationResult temp = X()
  a = temp.value(0) as String // Type Error: Cannot assign value of type String
      ↪to variable of type Integer.
  b = temp.value(1) as String
}
```

The AST transformation unrolls the multi-assignments before the type checker becomes active. Since the types of the return values need to be known to Moola to unroll multi-assignments, only calls to operations with a properly defined return value can be replaced. All other multi-assignments are ignored by the AST transformation and will cause the Groovy process to stop.

4.9 Plug-ins for Moola

Until this point, this chapter described how Moola was implemented as a domain-specific language on top of Groovy. However, to apply Moola to a real-world scenario, it also needs to understand and be able to execute model operations. Since the tools and languages used to implement model operations are manifold, Moola exposes a plug-in mechanism that allows plug-in developers to implement custom *operation types*. Two plug-ins were implemented in the course of this work: the *standard plug-in*[11] defines framework-agnostic operation types that may be useful in any operation chain (e.g. executing command-line tools). The *EMF plug-in*[12] defines operation types for working with MoDisco, Acceleo and ATL.

Listing 4.16: Definition of Moola's standard plug-in.

```
class StandardPlugin extends Plugin {

  String getName(){
    return "Moola␣Standard␣Plugin"
  }

  String getVersion(){
    return "0.0.1"
  }

  void applyTo(Process process){
    process.operationRegistry.register("Exec", ExecOperation)
  }

}
```

[11] https://github.com/We-St/moola/tree/master/org.moola.core/src/main/groovy/org/moola/plugin/standard
[12] https://github.com/We-St/moola/tree/master/org.moola.emf

To create a plug-in, a subclass of the *Plugin*[13] class needs to be defined. Listing 4.16 shows the StandardPlugin[14] class, which registers the ExecOperation[15] under the name **„Exec“** to any Moola process. The *ExecOperation* does no require any input and does not deliver any output, but defines a *command* setting which can be used to specify a command that should be executed. Listing 4.17 shows how it can be used.

Listing 4.17: Using the ExecOperation.

```
// Defining a new ExecOperation.
operation PING_SERVER ( type: "Exec" ) {
  command = "ping␣192.168.0.1"
}

// Calling the operation from within the run closure.
run {
  PING_SERVER()
}
```

The standard plug-in is availble to all Moola scripts by default. Other plug-ins, e.g. the EMF plug-in, need to be registered to the *PluginRegistry*[16]. Since there is no automatic detection of available plug-ins, this needs to be done by the caller of Moola's API (e.g. the Moola Eclipse plug-in in case of the EMF plug-in).

Any class defining a new operation type must itself be derived from the Operation[17] class. The concrete implementations of *ExecOperation* and the operation types for the EMF plug-in are omitted for brevity, but can be found in the respective plug-in source folders on GitHub.

4.10 Moola Eclipse Plug-in

To provide a better developer experience when writing operation chains in Moola, direct integration to target IDEs is beneficial. Since part of this thesis focused on orchestrating tools from the Eclipse Modeling Framework, integration to Eclipse seemed most natural. The two features chosen for the integration are:

- **Syntax highlighting** for Moola scripts provides visual feedback while reading and writing operation chains.
- **Script execution** enables running Moola scripts from within Eclipse. This allows developers to work without loosing the context of the IDE and further enables Moola scripts to run from *within the Eclipse workspace*. Since no new JVM process is started for Moola, operations can access the Eclipse workspace, which is necessary for certain tools (i.e. MoDisco) to run correctly.

[13] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/plugin/Plugin.groovy
[14] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/plugin/standard/StandardPlugin.groovy
[15] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/plugin/standard/ExecOperation.groovy
[16] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/plugin/PluginRegistry.groovy
[17] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/dsl/core/Operation.groovy

Listing 4.18 shows the relevant parts of MoolaKeywordHighlighter[18] class used for adding the keywords to the Eclipse syntax highlighter.

Listing 4.18: Adding syntax highlighting for Moola.

```
public class MoolaKeywordHighlighter implements IHighlightingExtender {

    public List<String> getAdditionalGroovyKeywords() {
        return Arrays.asList(
          "modeltypes", "model", "operation", "expects", "returns",
          "run", "parallel", "await", "plugins", "include", "from"
        );
    }

    ...
}
```

Relying on a host language for creating a DSL does not only allow for a rapid development process, but also allows for reusing host language infrastructure. In the case of Groovy, the *IHighlightingExtender*[19] can be used to add syntax highlighting with minimal effort.

Listing 4.19 shows an excerpt on how an embedded Moola process is started from within Eclipse. The full code can be found in the *RunHandler*[20] class.

Listing 4.19: Start embedded Moola process.

```
public class RunHandler extends AbstractHandler {

  @Override
  public Object execute(ExecutionEvent event) {
    try {
      this.showConsole(event);
      this.clearConsole();

      IFile file = this.getSelectedFile(event);
      String moolaPath = file.getLocation().toPortableString();
      String projectName = file.getProject().getName();
      String projectPath = file.getProject().getLocation().toPortableString();
      OutputStream outStream = console.newMessageStream();

      Runnable runnable = new MoolaRunner(moolaPath, projectPath, projectName,
          ↪outStream);
      Thread thread = new Thread(runnable);
      thread.start();

    } catch (Exception e) {
      throw new RuntimeException(e);
    }
    return null;
  }

  ...
}
```

[18]https://github.com/We-St/moola/blob/master/org.moola.eclipse/src/org/moola/eclipse/highlighting/MoolaKeywordHighlighter.java
[19]org.codehaus.groovy.eclipse.editor.highlighting.IHighlightingExtender.java
[20]https://github.com/We-St/moola/blob/master/org.moola.eclipse/src/org/moola/eclipse/commands/RunHandler.java

The event handler first shows the default Eclipse console so Moola can interact with it. It then uses Moola's API to execute the Moola script in a separate thread to not block the Eclipse user interface.

CHAPTER 5

Evaluation

"I don't want to achieve immortality through my work. I want to achieve it through not dying."
— Woody Allen

The result of this work, Moola, is an executable orchestration language for model operations. Shaw et al. [69] define several ways on how such tools can be properly evaluated. In a first step, by means of what Shaw calls *Evaluation*, Moola's implementation is checked against the feature list defined in Chapter 3. In a second approach, called *Example* by Shaw et al., Moola is applied to real-world scenarios to prove its applicability. The scenarios are taken from the ARTIST project [4]. Furthermore, the term correctness is defined for tools that allow the orchestration of model operations and applied to Moola. Finally, the threats to validity of this thesis are introduced and briefly discussed.

5.1 Feature Completeness

One of the earliest activities in designing and implementing Moola was extracting a list of features from real live scenarios and existing operation orchestration languages. This list of features and explanations for each individual feature can be found in Chapter 3. In this section, the implementation of Moola is evaluated for feature completeness, i.e. which features are present in Moola and how they are realized.

- **Allow Common Control Flows:** Moola uses Groovy keywords to specify the control flow within the orchestration code. These are *if* and *switch* for conditions, *for* and *while* for iterations and *try*, *catch* and *finally* for exception handling. To account for parallel execution, the *await* and *parallel* keywords were introduced in Moola. These keywords allow describing operation chains of all required control flow variants.

- **Support Diverse Operations:** The core of Moola was designed without any specific MDE framework in mind. The only requirement to run Moola is an installation of Groovy and a corresponding JVM. To add MDE frameworks and operation languages, Moola introduces a plug-in system. Any operation language can be included by deriving from the *Operation* class, thereby creating an operation type that can be used in Moola scripts to define operations of that operation language. A proof-of-concept plug-in for EMF-based models and operations was implemented as part of this work.

- **Allow Reusing Operations:** If operations are reused in the same or other projects, the operation definition is likely to remain the same. To only define the interface for an operation once, Moola introduces the *from* and *include* keyword pair, which can be used to import operation definitions from other Moola files.

- **Hide Operation Details, Expose Execution Errors:** The implementation details of an operation are not relevant for the orchestration, thus a black-box view on operations can be applied as orchestration developer. This is supported in Moola by solely relying on operation definitions. These can either be supplied by the developer who implemented the operation, or by the orchestration developer, and describe the interface required to invoke the operation. No other information about the operation is required in order to include it in an operation chain.

  To which extent errors are forwarded to the orchestration code depends on the implementation of the operation types. Concretely, each operation is defined as subclass of *Operation*. When an operation is called from within the orchestration code, the corresponding *execute* method of the *Operation* subclass is called. Any error thrown in this method is forwarded to the orchestration, thus allowing for more or less details depending on the implementation of the operation type. Furthermore, this enables different levels of exception details and behavior in error cases, since the implementation of operation types can consider settings to behave differently in certain scenarios.

- **Provide Early Feedback About Chain Validity:** Chain validation is implemented in Moola via Groovy's static type checking. Groovy code can be statically type checked by adding specific annotations and AST transformations to the Groovy code. To enable static typing, model types are used to type both models and in- and outputs of operations. Model types are transformed into real Groovy classes via an AST transformation. Multi-assignments are supported for operation invocations by applying an AST transformation which unrolls them. This allows Moola to evaluate the operation chain before a single operation was executed.

  Another form of early feedback can be derived from analyzing the call graph of an operation chain. A call graph [28] is a directed graph, which illustrates how indiviual parts of a program interact and can be used to determine unreachable code or gain knowledge on the impact of code changes. Due to limited scope of this work, this opportunity was not further investigated, but can be exploited in a future work.

- **Allow Embedding in Other Processes:** Moola is available as executable JAR file, which allows running Moola script via the command line through the regular *groovy* command. The first parameter indicates the location of the Moola script.

  The JAR file also contains the *Launcher*[1] class, which allows other JVM processes to embed Moola via a simple-to-use API. The Moola Eclipse plug-in illustrates how the API can be used to run Moola in the context of another process, in this case Eclipse.

- *(optional)* **Optimize for Performance:** To speed up the execution of operation chains, Moola allows for *concurrent execution* of operations via the *await* and *parallel* keywords. Furthermore, all models are passed in-memory between operations. To write a model to disc, the orchestration developer needs to explicitly invoke the *save* command. By relying on in-memory data flow, expensive file access can be avoided.

- *(optional)* **Allow User Interaction:** Moola scripts use regular Groovy code and can access Groovy's runtime environment, including standard in- and output. This allows Moola to print feedback to the console as well as retrieve user input from the console. The later was included in the orchestration code via the *ask* command, which allows orchestration developers to print a string and pause execution until the user types a response.

  Although not included in this work, more complex forms of user interactions are imaginable, including rich graphical user interfaces, e.g. via JavaFX[2].

5.2 ARTIST Scenarios

An EU Integrated Project for legacy software migration called ARTIST [77] was jointly developed at the Vienna University of Technology and other research institutions. ARTIST stands for „Advanced software-based seRvice provisioning and migraTIon of legacy SofTware“ and applies model-driven engineering to migrate legacy software. In the following section, three scenarios from the ARTIST project are described and implemented via Moola. The scenarios are illustrated in Figure 5.1.

Java to UML Profile

The first scenario shows a simple operation chain extracting a code model from a Java library and converting it to an UML profile. The two operations are sequentially applied and are implemented with two different underlying tools (*MoDisco* and *ATL*). Listing 5.1 shows a version of the orchestration in Moola. The code first applies the *EMF* plug-in and then defines two model types. Operation definitions for the two operations follow. While the MoDisco operation can be used as-is, the ATL operation receives specific in- and outputs as well as certain settings values, which need to be defined in the operation definition. The orchestration code then calls the operations in sequence.

[1] https://github.com/We-St/moola/blob/master/org.moola.core/src/main/groovy/org/moola/Launcher.java
[2] http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

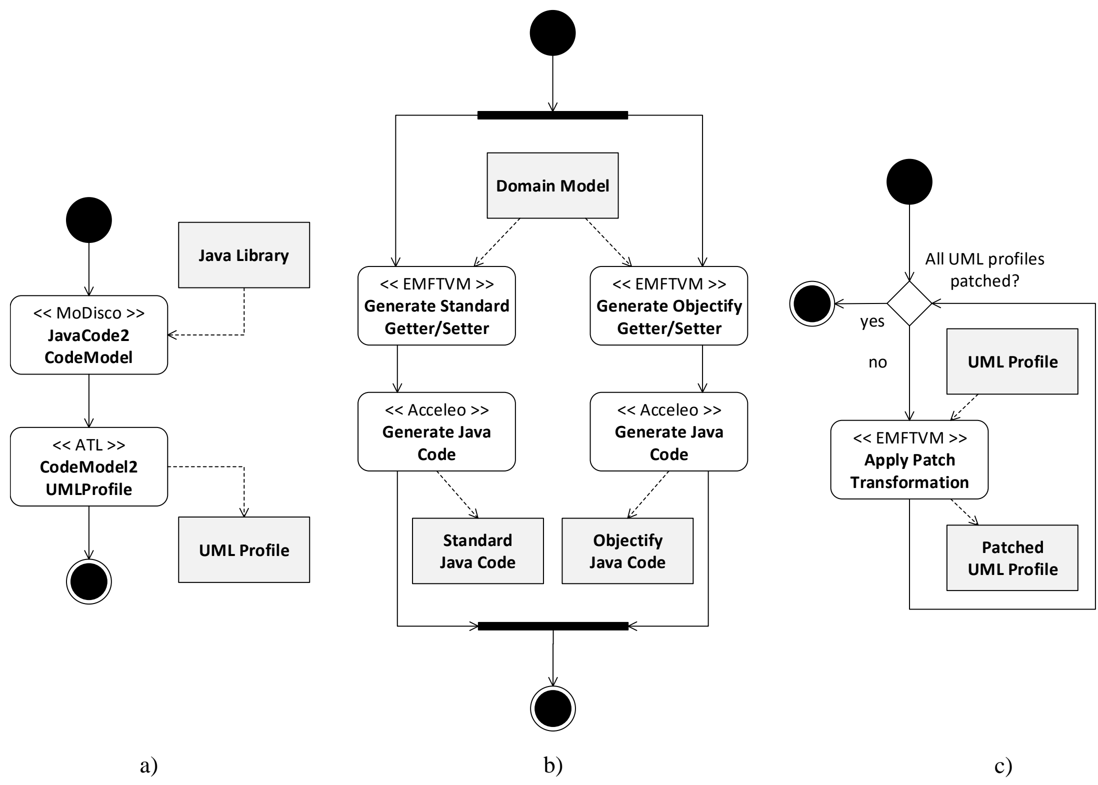

Figure 5.1: a) Java code to UML profile. b) Different implementations are generated in parallel. c) A patch transformation is applied to multiple UML profiles.

The concrete implementation of the Code to UML operation takes several input and produces several output models. These were left out in Listing 5.1 for a better understanding of the core operation chain. The places where models were left out for bevity are indicated with ... in the Moola code.

Listing 5.1: Moola script for ARTIST scenario a).

```
plugins "EMF"

modeltypes (
  Umlmm: "http://www.eclipse.org/uml2/5.0.0/UML",
  Jmm: "http://www.eclipse.org/MoDisco/Java/0.2.incubation/java"
)

// Operations

operation DISCOVER_JAVA ( type: "MoDisco" ) { }

operation CODE_TO_UML ( type: "ATL" ){
  expects cm: Jmm, ...
  returns up: Umlmm, ...

  path "CodeModel2UMLProfile.atl"
```

```
  option "supportUML2Stereotypes", "true"
  option "allowInterModelReferences", "true"
  library "javaLibrary", "javaLibrary.asm"
  library "profileLibrary", "profileLibrary.asm"

}

// Orchestration

run {
  Jmm codeModel = DISCOVER_JAVA( "<<␣Project␣in␣Eclipse␣>>" )
  ( umlProfile, ... ) = CODE_TO_UML( codeModel, ... )
  save umlProfile
}
```

Parallel Implementation Generation

The second ARTIST scenario demonstrates how a single domain model is used to generate two separate implementations of a system. The domain model describes a system which may run on different platforms. The two platforms in question are Google's App Engine, for which the implementation needs to be based on the Objectify Library, and any other JVM platform, in which case the implementation is based on standard Java objects with corresponding getters and setters.

Figure 5.1b) depicts the operation chain. A single domain model is the starting point for both implementations. Before the code can be generated, the domain model is transformed to two different models, which themselves include the relevant information for the code generation. Since the two platforms only share the domain model as origin, all steps leading to the final implementations can be executed in parallel. The domain model is transformed with two different operations, both implemented on *EMFTVM*. The code is then generated using *Acceleo*. The same Acceleo operation can be used on both execution paths to yield the implementations. Listing 5.2 shows the orchestration chain implemented with Moola.

Listing 5.2: Moola script for ARTIST scenario b).

```
from "./emftvm/operations.moola"  include SIMPLE_GETSET, OBJECTIFY_GETSET
from "./acceleo/operations.moola" include GEN_CODE
plugins "EMF"

modeltypes (
  Umlmm: "http://www.eclipse.org/uml2/5.0.0/UML"
)

// Model

model domainModel ( type: Umlmm, path: "./model/domainModel.uml" )

// Orchestration

run {
  await parallel({
    Umlmm simpleModel = SIMPLE_GETSET( domainModel, ... )
    GEN_CODE( simpleModel )
  }, {
    Umlmm objectifyModel = OBJECTIFY_GETSET( domainModel, ... )
    GEN_CODE( objectifyModel )
```

```
  })
}
```

Listing 5.2 shows the main Moola script for ARTIST scenario b). The operation definitions are imported from other Moola scripts. The only model is defined and used in the orchestration code in both execution paths to first generate more detailed models. These, in term, are then used as parameter to the same Acceleo operation to generate the code. The two *EMFTVM* operations take more than one model as input, which are again left out for better readability. Corresponding places are marked with ... in the Moola code.

Patching Multiple UML Profiles

The last ARTIST scenario applies a patch transformation on a list of UML profiles. Listing 5.3 shows the corresponding Moola script.

Listing 5.3: Moola script for the last ARTIST scenario.

```
plugins "EMF"

modeltypes (
  Umlmm: "http://www.eclipse.org/uml2/5.0.0/UML"
)

// Models

model petProfile ( type: Umlmm, path: "./models/petProfile.uml" )
model storeProfile ( type: Umlmm, path: "./models/storeProfile.uml" )
model userProfile ( type: Umlmm, path: "./models/userProfile.uml" )

// Operation

operation PATCH_TRANSFORM ( type: "ATL/EMFTVM" ){
  expects ucd: Umlmm
  returns patchedUCD: Umlmm

  path = "."
  module = "CASE01PatchTransformationAddEnumeration"
}

// Orchestration

run {
  // In case all profiles are defined as models.
  [ petProfile, storeProfile, userProfile ].each { m ->
    patched = PATCH_TRANSFORM( m, ... )
    save patched, m.path
  }

  // In case all files in a system should be processed.
  files( "./models/*.uml" ).each( file ->
    if (! file.canRead() ) {
      return
    }
    Umlmm m = load path: file.path, type: Umlmm
    patched = PATCH_TRANSFORM( m, ... )
    save patched, file.path
  }
}
```

In this example, several models and the patch operation are defined. The orchestration code shows two different ways on how to apply the transformation to all models, depending on whether the models are defined within the Moola script or not. The first approach uses Groovy's built-in list operator to create a list of all known models. A closure is then applied to each element of the list, which executes the transformation and saves the model.

Alternatively, if the models are not known at the time when the orchestration code is written, the *files* command can be used to get a list of files matching a certain path. The *files* command returns a list of *java.io.File*[3] instances and can be used to load the models stored at that location. Additionally, checks can be added to ensure that a corrupt file does not stop the execution.

Both approaches for the orchestration code run the patch process one model at a time. To perform the tasks consecutively, *parallel* can be used within the closures.

5.3 Correctness

The predominant metric for determining Moola's fitness for use in industrial-wide settings is *correctness*. In the context of this thesis, correctness of a tool for model operation orchestration is defined as yielding semantically identical results as to when the same sequence of operations is executed manually step-by-step. This definition of correctness deliberately ignores incorrect implementations of operations, since the purpose of a tool for model operation orchestration is on a higher level of abstraction and independent of implementation details of operations. In contrast, correctness of operations, especially transformations, can be determined on its own [52]. Furthermore, the definition deliberately ignores incorrect modeling of the operation chain within the tool, since the interest does not lie in correctly deriving and implementing an orchestration for particular use cases. As with other workflows, a single operation chain may or may not be correct [79]. A tool for operation orchestration is correct when it adheres to the above definition.

Listing 5.4: Parallel execution may lead to different operation sequences.

```
// Model and operation definitions...

run {
  A()

  await parallel({
      B()
      C()
    }, {
      D()
    })

  E()
}
```

The definition of *correctness* given in this thesis relates the outcome of an invocation of an operation chain via an orchestration tool to the outcome of the sequence of operations if called sequentially by hand. This effectively expresses that an orchestration tool is considered correct if the sequence of operations it executes yields an identical result as to when the operations are

[3] https://docs.oracle.com/javase/8/docs/api/java/io/File.html

executed manually. Since next to all tools for operation orchestration support parallel execution, they can yield different sequences of operations whenever executed. To derive a valid sequence of operations that reflect the concurrent execution of an operation chain, simultaneously executed operations must run in isolation and hence must not influence each other. A similar discussion exists in the domain of database transactions [5].

Listing 5.4 shows an operation chain in Moola that uses *await* and *parallel*. Through the concurrent execution of operations B, C and D, executing this Moola script may yield a different sequence of operations every time it is called. If isolation is guaranteed, any execution of this script will be equivalent to one of the following sequences:

- $A > B > C > D > E$
- $A > B > D > C > E$
- $A > D > B > C > E$

Operations A and E are outside the concurrent execution and will therefore always be first respectively last. Operation B will always be executed prior to C, but operation D might occur anywhere between A and E. To restate the definition of correctness, a tool for operation orchestration is considered to be correct when the result it yields is semantically identical to the manual execution of the operation sequence that yielded that result.

Correctness of Moola

The language core of Moola allows connecting operation implementations independent of any underlying framework. Specific operation languages can be connected to Moola via operation definitions, which in term use the API of the corresponding framework to invoke an operation (see section 4.3 for details). The correctness of Moola is thus depending on the correct behavior of the APIs used to call operations and the in-memory handling of data between operation invocations. If the operation language API behaves the same as when calling the operation manually (e.g. via a CLI, IDE integration, etc.), Moola's correctness can be inferred by looking at side-effect freeness and operation isolation, since Moola guarantees to modify in-memory data only through code explicitly stated in the *run closure*.

Operations in Moola are per convention side-effect free. Side-effect freeness [34] refers to the fact that a function does not modify values outside its scope. Moola operations must only operate on their input values (which they are allowed to change) to yield output values, any changes to the environment must not occur. Moola further guarantees isolation of model operations by using a pessimistic locking strategy (see section 4.7 for details). Operations using at least one common input cannot run in parallel, since Moola requires an exclusive lock for all input values before starting an operation. Operations which do not share any common input are, by definition, isolated from each other due to Moola's side-effect free execution of operations. Executing such operations in any order is bound to yield the same result.

Through isolation, any execution of a Moola script can be represented as sequence of operations. Since the APIs used to call the individual operations are expected to behave the same as

when called manually, Moola adheres to the correctness definition given in the beginning of this section.

Correctness of Moola's EMF plug-in

The correctness of Moola depends on the correct implementation of operation types, especially in conjunction with calling the API of the underlying framework to invoke operations and reading and writing models. Along with the core implementation of Moola, a EMF plug-in was provided that allows calling EMF operations such as MoDisco, ATL and Acceleo.

Moola's EMF plug-in holds models by using EMF's Resource class [4]. The API for reading and writing these models, alongside the API for invoking individual operations was empirically tested for correctness on all ARTIST scenarios and the case study. Since implementing checks for semantical identity for models is out of scope of this thesis, the resulting files were compared for byte-wise equality. Since EMF resources are completely described by the serialized content of a corresponding resource file, two files with identical content contain identical models. This yielded the confirmation that for all examined operation chains, the execution via Moola's EMF plug-in and the manual execution of the corresponding sequence of operations yielded files with identical content, thus providing empirical evidence that Moola's EMF plug-in adheres to the definition of correctness stated before.

5.4 Threats to Validity

The approach used to derive Moola's requirements and yield its implementation was driven by several real-life scenarios. The process was influenced by several assumptions, which are bound to follow when using a limited scope of observation. In this section, three assumptions are presented and their impact on Moola and the validity of this work are discussed.

- **Assumption 1: The examined use cases form a representational set within MDE.**
  The bulk of requirements for Moola was derived from a small set of real-life use cases taken from the ARTIST project and established over various discussions with different developers. Further requirements were derived by looking at the ATL Transformation Zoo[5] and examples used to justify requirements of other tools in the domain of model operation orchestration. The examined use cases and scenarios are just a small subset of possibly operation chains and may not constitute a representational picture of the MDE world in general. This threat was mitigated as best as possible by drawing use cases from different works [45, 59, 77] and by incorporating extensibility as core feature of Moola (see section 4.4 describing Moola's plug-in system for details).

- **Assumption 2: Operations can be called without side-effects.**
  A convention for implementing Moola operations is their side-effect freeness. This property is required to guarantee that operations do not influence each other during parallel

[4] org.eclipse.emf.ecore.resource.Resource

[5] https://www.eclipse.org/atl/atlTransformations/

execution. Side-effect freeness was stated as convention, because a comprehensive implementation of such a mechanism for object-oriented programming languages is a research topic on its own [64] and was deemed out of scope for this work. Especially when dealing with modeling frameworks and their APIs, no guarantees can be given on how operations are executed internally and if side-effects occur. In the case of EMF, the isolation of operations was empirically tested and therefore might change with future releases of EMF. If certain operations cannot be invoked without side-effects, these operations need to be manually locked or are not allowed to occur in concurrent executions. Moola provides means for orchestration developers to describe orchestrations containing such operations, although side-effect-related issues may only surface sporadically and might be hard to debug.

- **Assumption 3: Tools for describing operation chains are necessary.**
  MDE has a stable community of developers, as can be seen by looking at popular implementations such as EMF and the hundreds of thousands of downloads[6] it has. The manifold suggestions of tools for operation orchestration combined with the reluctant adoption of these tools in practice might indicate no need for orchestration tools at all. A core assumption of this work was that developers can benefit from describing their operation chains via specific tools. This assumption is presented without proof, and might justify investigations into this phenomenon as part of a future work.

[6] http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/keplersr2

CHAPTER 6

Related Work

„What day is it?“
„It’s today,“ squeaked Piglet.
„My favorite day“, said Pooh.

— A. A. Milne

The orchestration of model operations has been a field of extensive research in past years and has led to a number of tools and orchestration languages. While some of them were created only for academic purpose and were not published for immediate use, others have been in development and maintenance for several years. In this chapter, several orchestration languages are introduced and compared to Moola. The main criteria for the comparison is based on the list of features introduced in Chapter 3. Additional information such as last stable release, target platform, etc. are added if available.

Table 6.1 shows the result of the comparison. Since the chosen orchestration languages may have been built for a different audience than Moola or with a different set of features and methodologies in mind, the comparison favors Moola in the sense that the mentioned criteria formed the starting point for Moola’s design and implementation. A comparison of orchestration languages based on different criteria can be found in [50]. The remainder of this chapter discusses the different orchestration languages in greater detail and how they compare to Moola. Later in this chapter, interesting approaches from outside the immediate domain of operation orchestration are introduced.

Unified Transformation Representation (UTR)

A model-based approach to graphically describe operation chains is the Unified Transformation Representation (UTR) [82]. It specifically targets transformations and introduces three aspects of them: *implementation*, *specification* and *execution*. While the *implementation* describes the underlying source code, the *specification* describes the high-level behavior of a transformation including expectations on input and guarantees on output models. Finally, the *execution* represents a transformation that is applied to concrete models during the execution of the operation

Criterion	**Moola**	**UTR**	**Wires***	**MCC**	**Epsilon**	**Gradle**
Control Flow	Explicit	Explicit	Explicit	Explicit	Explicit	Implicit
Sequences	Yes	Yes	Yes	Yes	Yes	Yes
Conditions	Yes	No	Yes	Yes	Yes	Yes
Iterations	Yes	No	Yes	No	Via Plug-in	No
Parallel Execution	Yes	Yes	No	Yes	Yes	Yes
Diverse Operations	Plug-in System	Extensions, Transformations only	ATL only	Plug-in System, Transformations only	ANT Extensions	Plug-in System
Chain Validation	Static Typing	Static Typing	No	Static Typing	No	No
Embeddable	CLI and API	N/A	API	No	CLI and API	CLI and API
In-memory Model Passing	Yes	No	Yes	N/A	Yes	Yes
User Interaction	Yes	No	No	No	Yes	Yes
Implementation Details	**Moola**	**UTR**	**Wires***	**MCC**	**Epsilon**	**Gradle**
Concrete Syntax	Textual	Graphical	Graphical	Textual	Textual	Textual
Latest Release	08/2016	N/A	03/2011	N/A	11/2016	08/2017
Target Platform	Eclipse / JVM	Eclipse / JVM	Eclipse	Eclipse	Eclipse / JVM	Eclipse / JVM
License	MIT	N/A	EPL 1.0	N/A	EPL 1.0	ASL 2.0

Table 6.1: Comparison of Orchestration Languages.

chain. The approach is based on the assumption that a transformation can be seen as black box when the specification is formulated precisely. Hence, the *specification* is an implementation-independent description of the transformation. The approach allows modeling data flow between transformations when the input and output model types match. The UTR supports different roles participating in the MDD process such as the *Transformation Developer*, the *Transformation Specifier* and the *Transformation Assembler*. While the *Transformation Specifier* describes the behavior of a transformation on a high level, the *Transformation Developer* creates the concrete implementation conforming to the specification. The developer can thereby choose any transformation tool. The *Transformation Assembler* finally chooses specifications to fulfill the requirements and orchestrates them to an operation chain. UTR further supports the definition of model types. Instead of just relying on the meta-model to describe the input and output models of a transformation, UTR can specify local transformation requirements via constraints. This allows to further restrict the number of valid input and output models of a transformation.

While UTR is a convenient way to orchestrate model transformations, it differs from Moola in several ways. While it supports sequences and parallel execution, it does not support condition execution and iterations. It further restricts the available operations to be transformation. The data flow in UTR is limited to models and does not allow for other types of values, e.g. strings or numeric values. UTR saves all intermediate models to the file system in order for the developer to inspect them if necessary. While this helps during development and for troubleshooting, it also slows down execution. In Moola, developers can choose which intermediate models to save. Finally, UTR exists as Eclipse Plug-in and cannot be called outside the Eclipse context. Moola offers both a CLI and an API and can be called from within Eclipse as well as stand-alone process. No source code or plug-in of UTR could be found.

Wires*

A language for orchestrating ATL transformations is Wires* [59]. Wires* provides a graphical editor to describe the orchestration as model and can directly execute the orchestration by utilizing an execution engine. The graphical notation of Wires* allows describing the flow of models between transformations and explicitly supports the composition of transformations to reusable modules. Wires* supports sequences, conditions and iterations. Although parallel execution can be modeled, the execution engine executes transformations one after the other. Furthermore, Wires* supports persisting intermediate models if the user wants to use them outside of the orchestration or temporary models if they are only passed between transformations. The Wires* markup consists of model and transformation nodes and explicitly adds control flow structures, e.g. for conditional routing. Introducing dedicated structures for the control flow also has the side effect of needing so-called *Identity Transformations*. These special transformations do not alter the input models but rather forward them directly as outputs. Since a condition (called *Decision Node* in the markup) in Wires* can only have exactly two branches (a true-branch and a false-branch), *Identity Transformations* can be used to model empty else-branches. Another kind of special transformations are so-called *Generic Transformations*, which allow for higher-order transformations (HOT). This enables Wires* to orchestrate transformations that themselves were outputted by other transformations during the execution of the operation chain. Wires* can be

called from within Eclipse as well as via an API and can therefore be embedded in other JVM processes.

Differences between Wires* and Moola include: Wires* only offers support for the orchestration of ATL transformation, which also excludes user interactions from being part of the orchestration chain. Moola is designed to integrate with different orchestration languages and also allows for adding other types of operations to the orchestration. While Wires* relies on a graphical markup and adds control flow structures explicitly, Moola mainly relies on regular Groovy code to describe the orchestration. Wires* does not statically check (i.e. at compile time) if the inputs and outputs of successive transformation match in terms of meta-models. However, this is mentioned as goal for future releases.

All intermediate results (i.e. models produced by a transformation and consumed by another one) during an execution of Wires* are kept in-memory by default and are therefore temporary. To persist such artifacts, a corresponding action has to be modeled in the markup. This interesting approach was adopted in Moola, where all intermediate models are temporary and need to be persisted explicitly by the orchestration developer.

MCC

The MDA Control Center (MCC) [41] is an environment to orchestrate model operations, especially transformations. The basis forms a taxonomy of transformations, in which seven types of transformations are determined. MCC then introduces executable units that can be used to account for the various types of transformations defined in the taxonomy and a scripting language to orchestrate the executable units. The authors account for sequential, parallel and conditional execution in the scripting language. The approach was implemented as *Eclipse Plug-in*. The concrete implementations of the transformations can be realized using any transformation language and can then be plugged into the transformation environment using Eclipse extension points.

While MCC relies on a custom scripting language, Moola is implemented as Groovy DSL, allowing users to include any Groovy code if it is beneficial to the orchestration. As with Moola, MCC uses type checking to determine valid orchestration based on the transformation interfaces. Since the scripting language does not include loops, no iterations can be described with MCC. Furthermore, MCC relies on Eclipse's plug-in system to specify the implementations for the transformations. This requires MCC to run within the Eclipse context. All transformations need to exist as Eclipse plug-in in order to be integrated to an operation chain.

Epsilon

An advanced approach for describing operation chains can be found in the Epsilon language family [45]. This project contains domain-specific languages for describing various model operations. Examples are the Epsilon Transformation Language(ETL) to describe model transformations, the Epsilon Validation Language (EVL) to describe model validation constraints etc. Furthermore, the Epsilon project introduces an ANT-based workflow language to describe operation chains. By utilizing ANT, many powerful features are available to orchestration developers. These include calling Java code from within the operation chain, executing external

commands (e.g. command line tools), include ANT tasks of third-party vendors, use ANT's API or CLI to execute operation chains, etc. Especially the active ANT community with its many open source and freely available libraries and extensions is a valuable asset. Extra precautions have been taken by Epsilon developers to allow in-memory data flow between tasks. Special transaction tasks allow for atomic operations on models. Accompanied by a comprehensive on- and offline documentation, Epsilon offers a viable solution for describing operation chains.

Although ANT adds considerable power to the approach taken on by the Epsilon team, it has a severe impact on the syntax of the work flow markup. The XML-based syntax is directly exposed to the orchestration developer and verbose especially when dealing with longer orchestration chains. While iterations are not supported natively, they can be added via third-party ANT plug-ins. Moola supports defining model types on metamodel level. This allows for type checking before any operation is executed. The Epsilon approach does not include any static validation of the operation chain beyond pure syntax checks.

Gradle

A Groovy-based DSL for build automation is Gradle [3], which allows developers to specify build tasks and dependencies between them. Since Gradle is built as Groovy DSL, it allows for Java and Groovy code within the build script, including calls to library functions. A powerful plug-in system allows customizing and extending Gradle in various ways. Combined with an active community, Gradle is a popular choice for build automation on the JVM.

The driving data structure behind Gradle is the so-called dependency graph. When a new Gradle process is started, it operates in two phases. The first phase, the configuration phase, scans the Gradle script and builds the dependency graph based on the task definitions. The dependency graph is a directed, acyclic graph (DAG), in which tasks are represented as vertices and dependencies between tasks as directed edges. The dependency graph is subsequently used in the execution phase to derive the order of task execution. Tasks are executed in sequence if they directly or indirectly depend on each other. Tasks without any dependency relation can be executed in parallel. Each task definition can supply further information impacting the order of execution, e.g. tasks can be skipped if certain criteria apply or certain files did not change since the last run.

Since Gradle heavily relies on the dependency graph during execution and the dependency graph cannot contain any cycles, iterations in the build chain cannot be implemented. While conditions for single tasks are possible, excluding larger parts of the build chain is only possible by repeatedly excluding single tasks. If a task fails during execution, subsequent tasks depending on the failed task are not executed (except *Finalizer* tasks). However, Gradle will not stop the build process when a failed task is encountered, but will run as many tasks as possible and print an error summary to facilitate troubleshooting. While data flow between tasks is possible, task definitions do not contain an interface description of the expected inputs and outputs, therefore not allowing Gradle to apply Groovy's static type checking to validate the build before executing it.

Although Gradle cannot be used for describing orchestration chains for the aforementioned limitations, it was a major source of inspiration for Moola. The two-phase approach of Gradle was adopted in Moola to first parse Moola scripts and build an internal representation of the

orchestration chain. After validation of the chain, the execution follows. Moola's syntax for defining operations was strongly influenced by Gradle's task definitions. The pre- and post-actions possible on Gradle tasks inspired the same behavior on Moola operations.

Other Approaches

The Transformation Composition Modeling Framework [56] allows the orchestration of transformations based on UML2 activity diagrams. Remarkable on this approach is the explicit consideration of transformations requiring user interaction. The execution of the orchestration pauses while feedback of a user is required and resumes once the feedback was collected. Since Moola is set on top of Groovy, custom Java and Groovy code can be integrated to the operation chain. This especially encompasses code to interact with the user, e.g. showing the progress of the execution and asking the user for input.

An interesting approach for defining model transformations is RubyTL [12]. Although not meant for describing operation orchestrations, the approach is interesting from an implementation point of view. The authors used the features built into Ruby to create a domain-specific language that allows defining transformations in a short and concise way. RubyTL is a rule-based language allowing for declarative and imperative style definitions of rules. The declarative syntax allows users to specify what a rule should be doing without specifying how it should be done. This allows for short markup in standard use cases. In more complex scenarios, users can define rules by using ordinary Ruby code and specify exactly how a transformation should be executed. During runtime, RubyTL can automatically derive the order in which rules have to be executed. Since RubyTL is set on top of Ruby, it allows users to include transformation-wide services, e.g. for tracing. Finally, RubyTL is built with extensibility in mind. It uses a rich plug-in system to allow users to modify any part of RubyTL.

The domain of business process modeling comes with a rich set of workflow tools. Especially BPEL [87] and its related graphical notation BPMN [9] are worth noting. BPEL is an XML-based language for describing business processes. Building blocks in BPEL are tasks, that are implemented by standalone web services. It introduces control flow constructs for conditional and parallel execution, iterations and exception handling. BPEL aims to describe complex business tasks by orchestrating standalone services. Since these services may not be under immediate control of the orchestration developer, but may be provided by a third party, BPEL focuses on a strict black box view on services. Implementation details are neither required nor desired in the orchestration. To aid the developer in creating BPEL scripts, BPMN was introduced. BPMN provides a graphical notation for work flows and can be converted to either BPEL or any other workflow language (e.g. XPDL).

CHAPTER 7

Conclusion & Future Work

"This book was written using 100% recycled words."
— Terry Pratchett, Wyrd Sisters

Conclusion

In this work, a new orchestration language for model operations, **Moola**, was proposed. Based on a set of features drawn from real-world use case scenarios and existing orchestration languages, syntax and semantics for Moola were derived. A natural mapping between the domain of model operation orchestration and Groovy's built-in capabilities as scripting languages was developed and later implemented as Groovy DSL. Aside from the core orchestration language, which is independent of any concrete MDE framework, a Moola plug-in was developed to allow the orchestration of EMF-based models. The implementation was completed by an Eclipse plug-in featuring syntax highlighting for Moola scripts and the execution of Moola scripts from within Eclipse. The implementation was explained in great detail, focusing on how keywords of Moola were realized with Groovy's DSL features, including the abstract syntax tree transformations allowing Moola to bend Groovy's syntax limitations to an extend not possible without manipulation the AST. Moola was then evaluated for feature-completeness and compared to alternative orchestration languages. The source code of Moola was released under the MIT license and is available for download and modification under `https://github.com/We-St/moola`.

Future Work

While Moola is fully functional when it comes to the previously defined features, several open topics for future work can be conceived. In Chapter 2, the relationship between model and metamodel was introduced as resembling data and its type [78]. Moola uses this relation to type models as well as define interfaces for operations. The interface definition of an operation consists of the expected input and guaranteed output types of that operation and are implemented

in Moola by means of an AST transformation. The transformation creates real Groovy classes for each metamodel used. This allows Moola to use Groovy's type checking system to ensure the validity of an orchestration chain before its execution. Using metamodels directly as model types has the disadvantage of being too unspecific in certain scenarios. If an operation sets certain requirements on its input, a more concrete model type could be derived by defining constraints, effectively allowing only a subset of models conforming to the metamodel. Such constraints could be defined in addition to the metamodel on model type level. Alternatively or additionally, constraints could be added to the operation definition to implement design-by-contract [54], thereby reducing the number of allowed input and output models.

Another extension of the type checking mechanism is accounting for relationships between metamodels. In the current implementation of Moola, each metamodel leads to the definition of a model type Groovy class. If a metamodel is a specification or generalization of another metamodel, this should lead to a hierarchy of these Groovy classes, thus allowing for substitution of models in operation invocations. Furthermore, two metamodels might overlap in a way that a model can conform to both simultaneously. In this case, Groovy's implicit cast capabilities could be used to seamlessly transform the model from one model type to another, thus enabling the model to be used as input to operations which are defined on different, but compatible, model types. These possible relations between metamodels are not reflected in the current version of Moola.

Moola is capable of including higher-order transformations (HOT) [76] in an orchestration. When an operation produces another operation, i.e. a transformation, as output and writes it to the file system, later parts of a Moola script can reference these files as implementation of a new operation. Since this feature is a side-effect of the current implementation and was not included in the feature list, thorough tests are still needed to determine which conditions need to apply for Moola to support HOTs.

A more practical line of work is the extension of Moola for other MDE frameworks and languages. The core of Moola was implemented without the need for a specific environment other than the JVM and without a particular MDE framework in mind. A proof-of-concept plug-in for working with EMF-based models was part of this work, including operation types for MoDisco, Acceleo and ATL. More languages and other MDE frameworks can be included using Moola's plug-in system.

APPENDIX A

Moola Style Guide

To enable a consistent use of Moola, some style recommendations are expressed here that were derived after using Moola to describe several operation chains. This guide uses *must*, *should*, their negated forms *must not* and *should not* and *may* as defined in RFC 2119[1]. This guide only covers Moola-specific style recommendations. More extensive and general code guidelines for Groovy and Groovy DSLs can be found in the official Groovy style guide[2].

Naming

- **Files** holding Moola code should be marked by using the file extension *.moola*. If better editor support follows from revealing Moola's Groovy nature, files may use *.moola.groovy*. Moola files holding operation definitions should be located as close to the operation implementation as possible, preferably in the same directory, having the same name (aside from the file extension) as the operation implementation's main file, if such a file exists. The Moola file holding the orchestration code should be located in the project's main directory or a sub directory specifically allocated for holding the orchestration artifacts.

- **Model Types** relate to Groovy classes and should thus be named with upper camel case. Abbreviations, such as UML, should be converted to lower case with an upper case first character, e.g. Uml.

- **Models** are present as variables in the orchestration code and should therefore be named with lower camel case. Since most variables in the orchestration code are expected to be models of some sorts, name pre- or suffixes should not be used. Abbreviations should be handled similarly to naming *model types*.

[1] https://tools.ietf.org/html/rfc2119
[2] http://groovy-lang.org/style-guide.html

- **Operations** are the central point of reference and should thus be made clearly visible when observing the orchestration code. Their names should be written in CONST_CASE, with underscores to separate name parts.

Listing A.1: Do.

```
modeltypes (SomeType: "...")
modeltypes (Uml: "...")

model someName (...)

operation SOME_NAME (...)
```

Listing A.2: Don't.

```
modeltypes (someType: "...")
modeltypes (UML: "...")

model SomeName (...)
model SOME_NAME (...)
model intSomeName (...)

operation someName (...)
operation SomeName (...)
```

Control flow

Moola uses Groovy's control flow keywords (*if*, *switch*, *while*, *for*). Additionally, *await* and *parallel* were introduced. There should be exactly one space before and after the condition of these statements. Control flow keywords may be used in the beginning of one-line statements. Multi-line statements must use curly braces to indicate the scope of the statement. Code belonging to a *parallel* statement may be indented by at most one level to show its connection to *parallel*.

Listing A.3: Do.

```
if (cond) F()

while (cond) {
  F()
}

parallel ({
    F()
  }, {
    G()
  })
```

Listing A.4: Don't.

```
if (cond)
  F()

while(cond){
  F()
}

parallel ({
        F()
      }, {
        G()
      })
```

Calling operations

Calling a model operation corresponds to calling a function in Moola's *run closure*. Operation definitions do not support optional parameters or a variable number of parameters. Corresponding Groovy features therefore must not be used in conjunction with model operations.

Model operations in Moola can return zero, one or more return values. If a return value is not required, the value should be assigned to "_" (underscore), to indicate no further usage. Orchestration code must not use "_" (underscore) as parameter to a function call or on the right side of an assignment. It is strictly meant for write-only purposes.

Listing A.5: Do.

```
res, _ = OP(param1, param2)
NEXT_OP(res)
```

Listing A.6: Don't.

```
res, unused = OP(param1, param2)
NEXT_OP(res)

res, _ = OP(param1, param2)
NEXT_OP(_)

next = _
NEXT_OP(next)
```

Groovy has powerful built-in functional programming support. Closures were extensively used during the development of Moola and to implement operation chains. These functional aspects may be used in the orchestration code as partly shown in Listing 5.3, which uses *each* to apply certain operations to a list of models. More advanced functional aspects, such as currying, are supported and may be used by developers to improve readability of Moola scripts.

Listing A.7: Using currying for recurring params.

```
model const1 ( type: T, path: "..." )
model const2 ( type: T, path: "..." )

// OP takes three parameters, the latter two will always stay the same.
operation OP ( type: "..." ){
  expects dynamic: String , const1: T, const2: T
  returns result: String
}

run {
  // Bind the last two parameters of OP to always be const1 and const2.
  BOUND_OP = { d: String -> OP(d) }.rcurry(const1, const2)

  // BOUND_OP can now be invoked with only one param.
  BOUND_OP("dynamic␣value␣1")
  BOUND_OP("dynamic␣value␣2")
}
```

Listing A.7 shows how currying can be applied to operations. Other concepts, such as memoization, functional composition, etc. may be used when writing orchestration code. The primary reason for using any such concept should be to increase readability or performance.

Bibliography

[1] Don S. Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Trans. Softw. Eng. Methodol.*, 11(2):191–214, 2002.

[2] Alex E. Bell. Death by UML fever. *ACM Queue*, 2(1):72–80, 2004.

[3] Tim Berglund and Matthew McCullough. *Building and Testing with Gradle*. O'Reilly Media, Inc., 2011.

[4] Alexander Bergmayr, Hugo Brunelière, Javier Luis Cánovas Izquierdo, Jesús Gorroñogoitia, George Kousiouris, Dimosthenis Kyriazis, Philip Langer, Andreas Menychtas, Leire Orue-Echevarria Arrieta, Clara Pezuela, and Manuel Wimmer. Migrating legacy software to the cloud with ARTIST. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, pages 465–468, 2013.

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] Jean Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal*, 5(2):21–24, 2004.

[7] A. K. Bhattacharjee and R. K. Shyamasundar. Activity diagrams: A formal framework to model business processes and code generation. *Journal of Object Technology*, 8(1):189–220, 2009.

[8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.

[9] Michele Chinosi and Alberto Trombetta. BPMN: an introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012.

[10] Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. Workflow management in condor. *Workflows for e-Science*, pages 357–375, 2007.

[11] Brad J. Cox and Andrew J. Novobilski. *Object-oriented programming - an evolutionary approach (2. ed.)*. Addison-Wesley, 1991.

[12] Jesús Sánchez Cuadrado, Jesús García Molina, and Marcos Menárguez Tortosa. RubyTL: A practical, extensible transformation language. In *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, pages 158–172, 2006.

[13] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, 10(1):7–29, 2005.

[14] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in metaocaml, template haskell, and C++. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, pages 51–72, 2003.

[15] Fergal Dearle. *Groovy for domain-specific languages*. Packt Publishing Ltd, 2010.

[16] Harvey M. Deitel. *An introduction to operating systems (2. ed.)*. Addison-Wesley, 1990.

[17] Marlon Dumas and Arthur H. M. ter Hofstede. UML activity diagrams as a workflow specification language. In *«UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, pages 76–90, 2001.

[18] Clarence A. Ellis and Gary J. Nutt. Modeling and enactment of workflow systems. In *Application and Theory of Petri Nets 1993, 14th International Conference, Chicago, Illinois, USA, June 21-25, 1993, Proceedings*, pages 1–16, 1993.

[19] Rik Eshuis and Roel Wieringa. Comparing petri net and activity diagram variants for workflow modelling - A quest for reactive petri nets. In *Petri Net Technology for Communication-Based Systems - Advances in Petri Nets*, pages 321–351, 2003.

[20] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering, WiSME*, pages 262–271. Citeseer, 2004.

[21] Pascal Felber and Michael K. Reiter. Advanced concurrency control in java. *Concurrency and Computation: Practice and Experience*, 14(4):261–285, 2002.

[22] Andrew Forward and Timothy C. Lethbridge. Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals. In *International Workshop on Modeling in Software Engineering, MiSE 2008, Leipzig, Germany, May 10-11, 2008*, pages 27–32, 2008.

[23] Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley, 2011.

[24] Martin Fowler and Kendall Scott. *UML distilled - a brief guide to the Standard Object Modeling Language (2. ed.).* notThenot Addison-Wesley object technology series. Addison-Wesley-Longman, 2000.

[25] Etienne M. Gagnon and Laurie J. Hendren. Sablecc, an object-oriented compiler framework. In *TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems, 3-7 August 1998, Santa Barbara, CA, USA*, pages 140–154, 1998.

[26] Debasish Ghosh. *DSLs in action.* Manning Publications Co., 2010.

[27] Debasish Ghosh. DSL for the uninitiated. *Commun. ACM*, 54(7):44–50, 2011.

[28] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. pages 108–124, 1997.

[29] Stefan Hanenberg. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 22–35, 2010.

[30] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[31] Florian Heidenreich, Jan Kopcsek, and Uwe Aßmann. Safe composition of transformations. In *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*, pages 108–122, 2010.

[32] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Domain-specific languages in practice: A user study on the success factors. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*, pages 423–437, 2009.

[33] Paul Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3:39–60, 1997.

[34] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[35] John Edward Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 471–480, 2011.

[36] Edward G. Coffman Jr., M. J. Elphick, and Arie Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[37] Robert M. Herndon Jr. and Valdis Berzins. The realizable benefits of a language prototyping language. *IEEE Trans. Software Eng.*, 14(6):803–809, 1988.

[38] Stuart Kent. Model driven engineering. In *Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15-18, 2002, Proceedings*, pages 286–298, 2002.

[39] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, pages 220–242, 1997.

[40] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. Do static type systems improve the maintainability of software systems? an empirical study. In *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, pages 153–162, 2012.

[41] Anneke Kleppe. MCC: A model transformation environment. In *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, pages 173–187, 2006.

[42] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison Wesley object technology series. Addison-Wesley, 2003.

[43] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison Wesley object technology series. Addison-Wesley, 2003.

[44] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in action*, volume 1. Manning, 2007.

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

[46] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The epsilon object language (EOL). In *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, pages 128–142, 2006.

[47] Tomaz Kosar, Pablo E. Martínez López, Pablo Andrés Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information & Software Technology*, 50(5):390–405, 2008.

[48] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 260–267, 1988.

[49] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[50] Levi Lucio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss. FTG+PM: an integrated framework for investigating model transformation chains. In *SDL 2013: Model-Driven Dependability Engineering - 16th International SDL Forum, Montreal, Canada, June 26-28, 2013. Proceedings*, pages 182–202, 2013.

[51] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.

[52] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[53] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[54] Bertrand Meyer. Applying 'design by contract'. *IEEE Computer*, 25(10):40–51, 1992.

[55] Mohamed Mussa, Samir Ouchani, Waseem Al Sammane, and Abdelwahab Hamou-Lhadj. A survey of model-driven testing techniques. In *Proceedings of the Ninth International Conference on Quality Software, QSIC 2009, Jeju, Korea, August 24-25, 2009*, pages 167–172, 2009.

[56] Jon Oldevik. Transformation composition modelling framework. In *Distributed Applications and Interoperable Systems, 5th IFIP WG 6.1 International Conference, DAIS 2005, Athens, Greece, June 15-17, 2005, Proceedings*, pages 108–114, 2005.

[57] David A. Padua, editor. *Encyclopedia of Parallel Computing*. Springer, 2011.

[58] David J. Pearce and Paul H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *ACM Journal of Experimental Algorithmics*, 11, 2006.

[59] José E Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating ATL model transformations. *Proc. of MtATL*, 9:34–46, 2009.

[60] José Raúl Romero, José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. Formal and tool support for model driven engineering with maude. *Journal of Object Technology*, 6(9):187–207, 2007.

[61] Spencer Rugaber and Kurt Stirewalt. Model-driven reverse engineering. *IEEE Software*, 21(4):45–53, 2004.

[62] James E. Rumbaugh, Ivar Jacobson, and Grady Booch. *The unified modeling language reference manuel - covers UML 2.0, Second Edition*. Addison Wesley object technology series. Addison-Wesley, 2005.

[63] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In *Conceptual Modelling 2006, Third Asia-Pacific Conference on Conceptual Modelling (APCCM 2005), Hobart, Tasmania, Australia, January 16-19 2006*, pages 95–104, 2006.

[64] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for java programs. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, pages 199–215, 2005.

[65] Ken Schwaber. Agile project management. In *Extreme Programming and Agile Processes in Software Engineering, 6th International Conference, XP 2005, Sheffield, UK, June 18-23, 2005, Proceedings*, page 277, 2005.

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

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

[68] Shane Sendall and Jochen Küster. Taming model round-trip engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, page 1, 2004.

[69] Mary Shaw. What makes good research in software engineering? *STTT*, 4(1):1–7, 2002.

[70] Peter Smith. *Software build systems: principles and experience*. Addison-Wesley Professional, 2011.

[71] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

[72] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development - technology, engineering, management*. Pitman, 2006.

[73] Jim Steel and Jean-Marc Jézéquel. On model typing. *Software and System Modeling*, 6(4):401–413, 2007.

[74] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[75] Jeff Sutherland. Agile development: Lessons learned from the first scrum. *Cutter Agile Project Management Advisory Service: Executive Update*, 5(20):1–4, 2004.

[76] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*, pages 18–33, 2009.

[77] Javier Troya, Hugo Brunelière, Martin Fleck, Manuel Wimmer, Leire Orue-Echevarria, and Jesús Gorroñogoitia. ARTIST: model-based stairway to the cloud. In *Proceedings of the Projects Showcase, part of the Software Technologies: Applications and Foundations 2015 federation of conferences (STAF 2015), L'Aquila, Italy, July 22, 2015.*, pages 1–8, 2015.

[78] Antonio Vallecillo and Martin Gogolla. Typing model transformations using tracts. In *Theory and Practice of Model Transformations - 5th International Conference, ICMT 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings*, pages 56–71, 2012.

[79] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[80] Arie van Deursen and Paul Klint. Little languages: little maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998.

[81] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[82] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Uniti: A unified transformation infrastructure. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, pages 31–45, 2007.

[83] Bert Vanhooff, Stefan Van Baelen, Aram Hovsepyan, Wouter Joosen, and Yolande Berbers. Towards a transformation chain modeling language. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 6th International Workshop, SAMOS 2006, Samos, Greece, July 17-20, 2006, Proceedings*, pages 39–48, 2006.

[84] Mirko Viroli. Towards a formal foundation to orchestration languages. *Electr. Notes Theor. Comput. Sci.*, 105:51–71, 2004.

[85] Eelco Visser. Webdsl: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 291–373, 2007.

[86] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14, 1992.

[87] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F Ferguson. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005.