Generation of Domain-Specific Language-to-Language Transformation Languages

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Dawid Kopetzki

Dortmund

2019

Tag der mündlichen Prüfung:
16.09.2019

Dekan:
Prof. Dr.-Ing. Gernot A. Fink

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Sven Jörges

Abstract

The increasing complexity of software systems entailed by the imposed requirements and involved stakeholders creates new challenges towards software development and turns it into a complex task. Nowadays, sophisticated development approaches and tools are needed to handle this complexity. Model-Driven Engineering (MDE) provides means to abstract from the details of a software system during the development phase by using models. Domain-Specific Modeling (DSM), a branch of MDE, tackles the complexity by proposing to use modeling languages which are restricted towards the solution space of the targeted problem domain. These *Domain-Specific Visual Languages* (DSVLs) are used in the DSM approach to create models in the restricted design space making the generation of modeled solutions feasible and providing a basis for the communication between various stakeholders. Since for each of the targeted domains a DSVL is needed, language workbenches emerged which support the development of DSVLs. During the development of a DSVL the semantics of the language has to be defined and, if the DSVL changes, existing models created using the DSVL have to be migrated. Furthermore, models are represented in a specific format hindering the application of, e.g., mature verification methods and tools. To solve these tasks, model transformations are promoted to transform models into different representations conforming to other DSVL.

This thesis presents a new kind of model transformation languages, which can be used to handle the arising tasks during the development of DSVLs. These transformation languages are tailored towards the domain of "computational model transformations between DSVLs". The presented transformation languages are based on graph-transformation approaches and simplify the specification of computations by utilizing Plotkin's Strucural Operation Semantics (SOS), and thereby facilitate the definition of computation steps in a declarative way. This approach suffers from the versatility in the scope of DSVLs and thereby requires techniques to reduce the development costs of the transformation languages for different source and target languages.

The key to reduce the development costs is the application of the Domain-specific, Full-generation, Service orientation (DFS) approach for the domain of model transformation languages. The application of domain-specifc concept results in graph-based, domain-specific two-level transformation languages. The essence of those languages is captured in a pattern describing possible two-level transformation languages. This pattern is used as the basis for the definition of a generator for those kind of transformation languages making full-generation feasible. The semantics of pattern matching and rewriting rules in the context of graph-based transformations are defined by the utilization of existing graph-transformation tools.

Attached Publications

I S. Naujokat, M. Lybecait, D. Kopetzki & B. Steffen. **CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools.** In. Int. J. on Software Tools for Technology Transfer (STTT), 2017.

The presented concepts were discussed among all authors. Stefan Naujokat was main author of all sections. The Cinco implementation has been primarily done by Michael Lybecait and myself.

II M. Lybecait, D. Kopetzki, P. Zweihoff, A. Fuhge, S. Naujokat & B. Steffen. **A Tutorial Introduction to Graphical Modeling and Metamodeling with CINCO.** In: Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018), volume 11244 of LNCS, pages 519–538. Springer, 2018.

The presented concepts and technologies were discussed among all authors. I co-authored section 4. Implementation of the GCS tool was primarily done by Annika Fuhge under my supervision and implementation of Pyro was done by Philip Zweihoff, both building on concepts and technologies developed by Stefan Naujokat, Michael Lybecait and myself.

III S. Bosselmann, M. Frohme, D. Kopetzki, M. Lybe- cait, S. Naujokat, J. Neubauer, D. Wirkner, P. Zweihoff & B. Steffen. **DIME: A Programming-Less Model-ing Environment for Web Applications.** In Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part II (ISoLA 2016), volume 9953 of LNCS, pages 809–832. Springer, 2016.

The presented concepts and technologies were discussed among all authors. Imple- mentations of Dime have been done primarily by Steve Boßelmann, Markus Frohme, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner and Philip Zweihoff. En- hancements in Cinco required for the implementation of Dime have been done primarily by Michael Lybecait and myself.

IV M. Lybecait, D. Kopetzki & B. Steffen. **Design for 'X' through Model Transformation.** In: Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018), volume 11244 of LNCS, pages 381–398. Springer, 2018.

The presented concepts were discussed among all authors. I co-authored all sections and I am main author of section 5.

V D. Kopetzki, M. Lybecait, S. Naujokat & B. Steffen. **Towards Language-to-Language Transformation.** In. Int. J. on Software Tools for Technology Transfer (STTT), 2020

The presented concepts were discussed among all authors. I co-authored all sections and I am main author of sections 3, 4 and 5

Contents

1 Introduction **1**
- 1.1 My Contribution . . . 3
- 1.2 Related Work . . . 12
- 1.3 Organization of this Thesis . . . 13
- 1.4 Context of Attached Publications . . . 14

2 Background **17**
- 2.1 Cinco Meta Tooling Suite . . . 17
- 2.2 The WebStory Language . . . 19
- 2.3 Model Transformations . . . 20
  - 2.3.1 Categorization of Model Transformations . . . 21
  - 2.3.2 Graph Transformations . . . 21

3 Towards Graph-Based Model Transformations in the Cinco Framework **25**
- 3.1 Model Transformations in Cinco . . . 27
  - 3.1.1 Model Semantics . . . 27
  - 3.1.2 Meta Model and Model Co-Evolution . . . 31
  - 3.1.3 Validation and Verification . . . 34
- 3.2 The Need for Domain-Specific Transformation Tools . . . 34
  - 3.2.1 Code Generation in Cinco . . . 35
  - 3.2.2 Model-to-Model Transformations . . . 35

4 Generation of Language-to-Language Transformation Languages **43**
- 4.1 Two-Level Transformation Languages . . . 43
- 4.2 Generation of Two-Level Transformation Languages . . . 45
  - 4.2.1 Source Language Pattern . . . 47
  - 4.2.2 Target Language Pattern . . . 48
  - 4.2.3 The Auxiliary Rule Language . . . 48
  - 4.2.4 Generator Configuration . . . 50

4.2.5 The WebStory to Kripke Transition System Transformation Language 52

5 Future Work **55**
5.1 Application 55
5.2 Higher-Order Transformations 55
5.3 Auxiliary Language 56

6 Conclusion **57**

1 Introduction

The increasing complexity of software systems imposes large problems on their development. To handle this complexity, one has to abstract from technical details and focus on the solution design for the given problem. Model-based development approaches reinforce the abstraction process by providing means to capture the main concepts of a solution in a formal model, abstracting from implementation details. In Model-Driven Engineering (MDE) [Sch06, HRW11] models are the main artifacts during the development process from which code is generated (semi-)automatically. The Object Management Group (OMG) [10] introduced the Model-Driven Architecture (MDA) [BCT05, Poo01], a set of standards to facilitate the creation of models and transformations between them down to the actual system's implementation. Those models are usually created using general purpose-modeling languages.

Despite the increase of productivity resulting from model-driven development techniques, one intrinsic problem still persists: The semantic gap between technical experts and business experts, i.e., what the business experts expect and what the technical experts understand and realize. In [MS09], Margaria and Steffen propose eXtreme Model-Driven Design (XMDD) as a development paradigm to reduce this gap by including the business experts in the development process of the software system. XMDD is a combination of *service orientation* and *model-driven design*. In this approach, a business expert models his own solution using predefined building blocks, which are based on services implementing basic functionalities. The building blocks can in turn be composed to new ones, providing more powerful functionalities. The model is the main development artifact, from which the whole system is generated.

Domain-Specific Modeling (DSM) [KT08] follows the same goals, but problem solutions are created using Domain-Specific Languages (DSL) [FP11] and tools, tailored specifically for the problem domain. These tools reduce the possible design space by limiting language constructs towards the target domain. Kelly and Tolvanen report in [KT08] on the advantages of DSM, e.g., a vast increase in the productivity and a better quality of applications, achieved by a restricted design space and code generation. Furthermore, the domain-specialization of the language

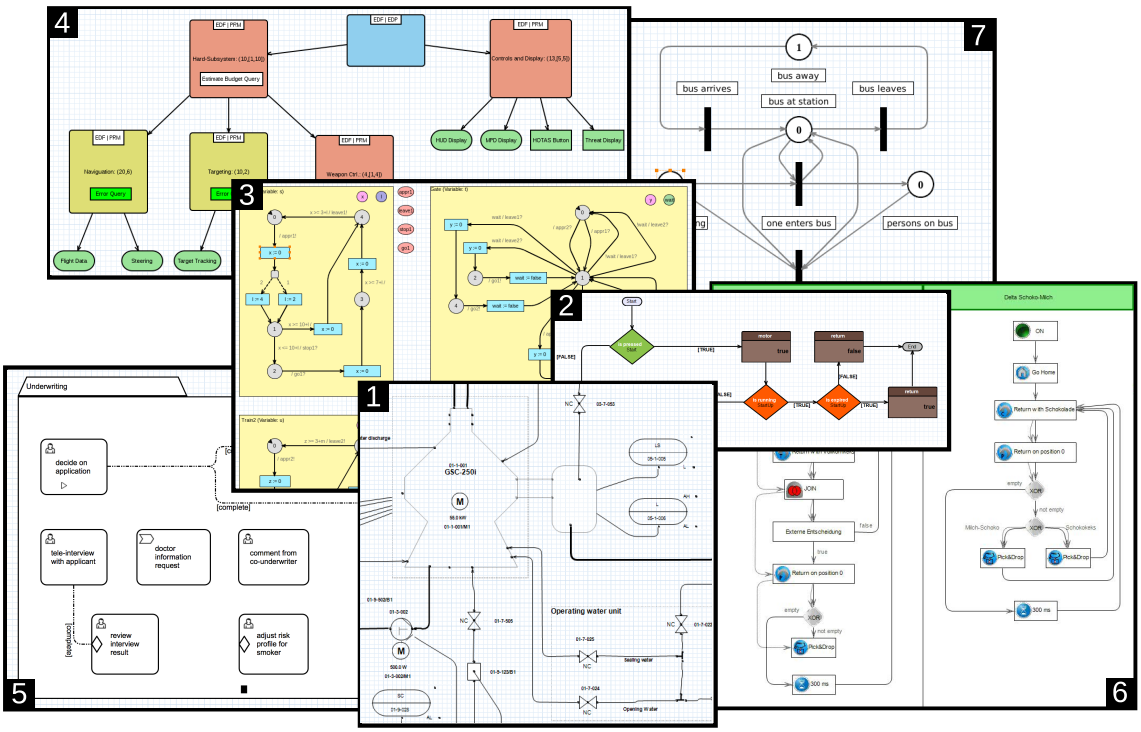


Figure 1.1: Overview of domain-specific visual languages developed using Cinco. (1) Piping & Instrumentation diagram [WMN16] (2) Flow Graph [WMN16] (3) Probabilistic Times Automata [NTI+14] (4) Hierarchical Scheduling Systems [CKL+17] (5) OMG's Case Management Notation (CMMN) [Wec16] (6) Easy Delta Pick&Place DSL [BDG+15] (7) Petri net [NLKS17]. (reprinted from [SGNM19])

enhances the comprehensibility of created solutions for domain experts and even enables them to create their own solutions.

Models and their meta models require subsequent processing steps introduced by typical activities in model-driven engineering, i.a., refactoring, migration, verification, and simulation [Men13]. These activities are realized by model transformations, which, e.g., change the models' structure to meet specific structural constraints (refactoring), transform them into an existing semantic domain, or translating them to formats suitable for, e.g., verification tools.

In the last seven years, we developed the Cinco meta-tooling suite [NLKS17], a language workbench [Fow05] easing the development of graph-based, domain-specific visual languages (DSVL). Cinco is used in many industrial and academic projects yielding several DSVLs. Figure 1.1 shows exemplary models of DSVLs created using Cinco. These DSVLs are developed, with simplicity in mind, to provide business experts the possibility to model their solutions. Of course, these DSVLs and the associated models are subject to model transformations targeting different goals. Ideally, these transformations should be defined using transformation languages following the same concepts as the underlying DSVLs, i.e., in a graph-based visual

way, utilizing the abstract and concrete syntax of the involved DSVLs. But due to the large number of DSVLs, the manual development of dedicated transformation tools for those languages is not feasible. Furthermore, one transformation language is not necessarily appropriate to specify model transformations for different purposes. This thesis presents the results of my efforts towards specialized graph-based model transformation languages for DSVLs, which should provide an easy way for the specification of transformations.

1.1 My Contribution

Stefan Naujokat introduced in his Ph.D. thesis [Nau17] the Domain-specific, Full-generation, and Service orientation (DFS) approach for modeling languages and then applied this approach reflexively to the domain "development of graphical domain-specific visual languages" resulting in the CINCO meta-tooling suite. CINCO is used to create graph-based DSVLs by specifying their abstract and concrete syntax. But the development of DSVLs does not end at this point. I consider three major activities arising during the development of DSVLs, each targeting a different purpose.

1. *Semantics definition*: Given the abstract syntax of a DSVL, one has to specify the meaning of models expressed in this language, i.e., define the language's semantics. Usually, this can be done by generating executable code in a high-level programming language (e.g., Java), or by providing translational semantics mapping the concepts of the source language to a semantic domain.

2. *Model Migration and Model Co-Evolution*: Models expressed in a DSVL conform to the language's meta model (abstract syntax). Small changes in the meta model can invalidate the conformance of existing models, thus the conformance has to be "repaired".

3. *Validation and Verification*: Even if models conform to the meta model of a DSVL, it is not guaranteed that the models represent reasonable solutions or satisfy desired properties posed to a solution. The design space of a DSVL can be further restricted towards reasonable solutions by validating the static semantics of a language. To verify desired properties of a solution, one can use existing formal verification methods and thereby benefit from various advantages resulting from the research in that area. Those verification methods require their input models to conform to a specific format (meta model). Generally, models expressed in DSVLs do not satisfy this requirement and have to be transformed into a suitable format for a verification tool.

All three activities can be performed by transforming the models into models (1.) expressed in a language with existing semantics, (2.) conforming to a modified meta model, or (3.) conforming to the expected format of a verification tool. Thus, model transformations provide excellent means to perform these tasks.

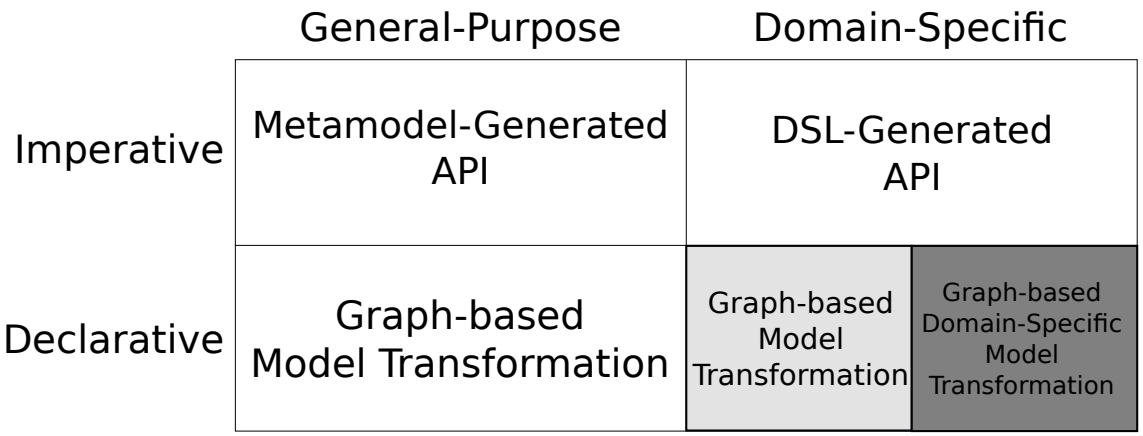


Figure 1.2: Categorization of model-to-model transformation approaches studied in the context of this thesis.

The majority of existing approaches for graph-based transformations usually consider transformations where the required information is represented statically in the model. In [KLNS20], we present model transformations, concerning the migration and verification activity, in which the required information is not statically represented in the model but has to be computed. Although the aspect of computation is a mature topic in computer science, graph-based approaches do not address it specifically in their rule definition languages. In this thesis, I present a transformation language supporting the specification of computations in a declarative way. This language is inspired by Plotkin's Structural Operation Semantics [Plo81], which facilitates the definition of a programming language's semantics in a very elegant way.

I studied how the concepts of the DFS approach can be applied to the domain of model transformation languages, and thereby ease the development of model transformations for DSVLs. The resulting transformation languages hide the complexity of the definition of model transformations and consider the involved DSVLs in their rule definition language. Thereby, the languages provide a less error-prone development process. These languages are used to implement the described activities in the scope of DSVLs. Figure 1.2 compares common approaches for model transformation languages with the ones I studied. The DFS approach is reflected as follows:

- *Domain-specific*: Transformation languages should respect the abstract and concrete syntax of the DSVLs involved in a transformation (Approaches in the `Domain-Specific` column of Fig. 1.2). Additionally, the purpose of a transformation is considered as a domain-specific aspect for the transformation language (cf. `Graph-based Domain-Specific Model Transformation` in Fig. 1.2).

- *Full-generation*: Due to the amount of possible transformation languages resulting from the application of the *domain-specific* aspect of DFS, these languages should be generated to fully executable transformation languages.

- *Service orientation*: The `Declarative/Domain-Specific` approaches facilitate the specification of transformations using graph patterns. Finding matches of these patterns in the input model is not trivial, thus existing matching algorithms are used.

The first application of domain-specificity for transformation languages resulted in a CINCO-generated domain-specific API for each DSVL (cf. Fig. 1.2, `DSL-Generated API`). This allowed for a more convenient implementation of model transformations in an imperative manner, i.e., a developer specified *how* to transform models of the source language to target language models. Using this generated API, the three described activities can be defined by:

1. Translating models expressed in a new DSVL into a DSVL with an existing semantics definition or to executable code in a high-level programming language.

2. Transforming models which do not conform to a modified meta model to a model conforming to it.

3. Translating models expressed in a DSVL to models suitable for verification tools.

The disadvantages of this approach are that the logic of the transformation engine is implemented in each transformation anew and the transformation rules for DSVLs are implemented using an imperative textual programming language.

To allow for a declarative specification of *what* the transformation should achieve, i.e., the relation between source and target language types, I employed graph-based model transformation languages for DSVLs (cf. Fig. 1.2 `Graph-based Model Transformation`). These transformation languages allow for the specification of the relation between structures of the source language and structures of the target language using the concrete syntax of the corresponding DSVLs. The transformation logic is part of the language and does not have to be specified by the transformation developer.

Using the specialized graph-based transformation languages allows for a more convenient definition of transformations to realize the introduced activities. However, some of these transformations are more intricate and result in complex transformation rules, even if the graph-based transformation languages tailored towards the DSVLs are used. For instance, model migration or verification may require the aggregation of information, leading to the specification of rules that simulate a computation. A solution for this problem is to consider the purpose of the transformation as domain-specificity, leading to `Graph-based Domain-Specific Model Transformation` languages. These languages provide an additional auxiliary language tailored towards the domain of computation.

The highlighted parts of Fig. 1.2 represent two of my contributions. The light gray part represents a technical contribution, i.e., the realization of existing approaches

in our meta-tooling suite CINCO. The dark-gray part represents my contribution towards a new kind of model transformation languages. My third contribution consists of the generator for graph-based domain-specific visual languages.

1. Graph-based Model Transformation In graph-based model transformations, rules are specified using graph patterns which relate to structures in the source and target language, meaning that the target pattern is created if the source pattern is present in the source model (cf. Sect. 2.3.2). The transformation language facilitates the specification of patterns using the concrete syntax of the involved DSVLs, leading to more comprehensible rules than, e.g., in an imperative textual approach. Users who are familiar with the languages involved in the transformation, recognize the elements used for pattern definition at first glance. For instance, the transformations depicted in Fig. 2.3 and Fig. 3.8 specify similar rules, but the rules in Fig. 3.8 use the concrete syntax of the underlying DSVL (cf. Fig. 2.2).

The derivation of pattern languages for domain-specific languages is described in, e.g., [KMS+10, HRW15]. Thus, my first contribution is a technical one, which paved the way towards graph-based transformation languages for DSVLs in CINCO. Given a DSVL specification, a graph pattern language is generated which satisfies the requirements described above. Furthermore, to execute a transformation, the semantics of pattern matching and graph rewriting were defined by means of translational semantics. Pattern matching consists of finding sub-structures in a graph on which rewrite rules can be applied. Thus, pattern matching relates to the graph isomorphism problem and constitutes a complex task. To prevent the implementation of pattern matching algorithms for DSVLs, I utilized the general-purpose graph transformation tool *Groove*. Groove facilitates the specification of transformation rules for attributed typed graphs. To use Groove's capabilities, the DSVL, its models, and the defined rules are transformed to Groove graphs.

- The meta model of a DSVL is transformed to a type graph in Groove.
- The models expressed in a DSVL are transformed to host graphs in Groove, which are typed by the corresponding type graph.
- The rules are transformed to rule graphs in Groove, which are typed by the corresponding type graph.

I supervised the master thesis of Marius Feltmann [Fel19], in which he implemented a generator for graph-based transformation languages for DSVLs.

A drawback of transformation languages typed towards DSVLs is their inflexibility. Each concept that does not belong to source or target language has to be explicitly added as a new type in the transformation language.

For the computation required for the transformations described in [KLNS20], several additional concepts are needed to model the components of an interpreter.

These concepts have to be represented by means of node and edge types. Additionally, the *execution* of the interpreter has to be specified using rules that only facilitate for graph rewriting. Thus, a transformation developer is left alone with the specification of the interpreter's execution by means of *finding*, *deleting*, and *creating* model elements. Using these basic functionalities, the following aspects have to be specified which introduce the complexity in the corresponding transformation rules.

- Explicit modeling of runtime information: The computed data is represented by special data nodes and the current position in the input model is represented by, e.g., a special node type which refers to the current node using a dedicated edge type.

- Defining a computation step by finding the special edge representing the current position in the graph, deleting and creating it at the next position.

- Extending the transformation rules with new types that can be used to specify conditions determining when a computation should start or when it has finished. This allows to realize a scheduling for the application of the actual transformation rules and thereby create elements in the target language at the correct moment.

- Definition of rules determining if a target element representing the current state of the computation was already created.

- Modeling of a global termination condition for the transformation.

I modeled the transformation incorporating these components using a general-purpose graph transformation language. Some of the resulting rules are represented in Fig. 1.5 and Fig. 1.6. The complexity of the transformation rules introduced by the computation is tremendous, although the needed concepts are well-known and mastered in computer science.

2. Graph-based Domain-Specific Model Transformation My second contribution is the definition of languages, that are used to define computational transformations. To overcome the problems introduced by modeling computation concepts by means of node and edge structures, I considered the required concepts as domain-specific aspects themselves and extracted the relevant parts in form of the following design decisions.

- *SOS-based Auxiliary Rules*: To handle needed computations for a transformation, I introduce an auxiliary language based on Plotkin's Structural Operation Semantics (SOS) [Plo81]. It encapsulates the specification of individual computation steps allowing to separate the computation rules from the rules that are responsible for the creation of the target model.

- *Additive Transformation System*: Constraining the source and target pattern languages by preventing the deletion of elements in the target model, and preventing the deletion and creation of elements in the source model, results in an additive transformation system. With a proper identification of target elements, the transformation can be executed until a fix-point is reached. Thereby, a transformation developer does not have to explicitly specify a global termination condition. Defining such a condition would require to enhance transformation rules by bookkeeping newly created elements and checking for their existence before the application of subsequent rules.

- *Separation of Source Language and Target Language*: To focus on the constituents of a transformation and reduce the complexity implied by explicitly defining the relations between source, target, and computation concepts in one representation the transformation language separates the definition of source and target language pattern. Additional to the separation of the computation concepts, this results in a modular language.

These design decisions are manifested in a two-level transformation language pattern, which is tailored to support computations. The resulting languages provide specialized means for handling the computation aspects as follows.

- A two-level transformation language providing two types of rules, distinguishing between computation and transformation rules.

- The first rule type defining the relation between source and target language elements using pattern languages, restricted as described in the *Additive Transformation System* design decision.

- The second rule type defining individual computation steps in the source model using SOS. Individual steps are specified by transitions between states that represent configurations of a computation.

Those languages allow for the specification of the actual transformation rules in a declarative way, by relating source and target language elements. Even the specification of computation steps is realized in a declarative way by relating source language elements with SOS configurations. Fig. 1.3 and Fig. 1.4 exemplify the transformation rules for the WebStory to KTS transformation [KLNS20]. The relation between source and target elements is described by the rule in Fig. 1.4, where a source language pattern is defined in the upper part on the left-hand side, a target language pattern in the lower part. The upper part on the right-hand side represents the start and end configuration of a computation. This allows for the definition of a start configuration for the computation. The end configuration serves as basis for the derivation of attribute values for target language elements. The computation steps are defined by the rules in Fig. 1.4. Source language patterns are defined in the upper part, a transition between two SOS configurations is defined in the lower part. Relations between source language elements and configurations

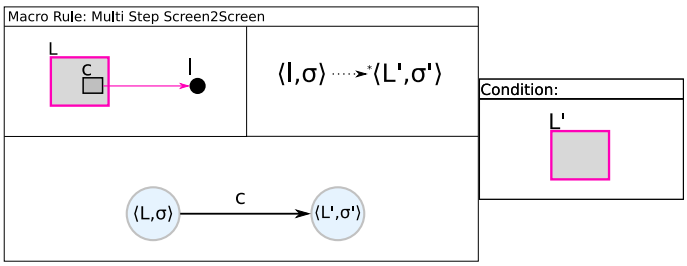


Figure 1.3: First-level rule defining the transformation of WebStory elements into states of a KTS.

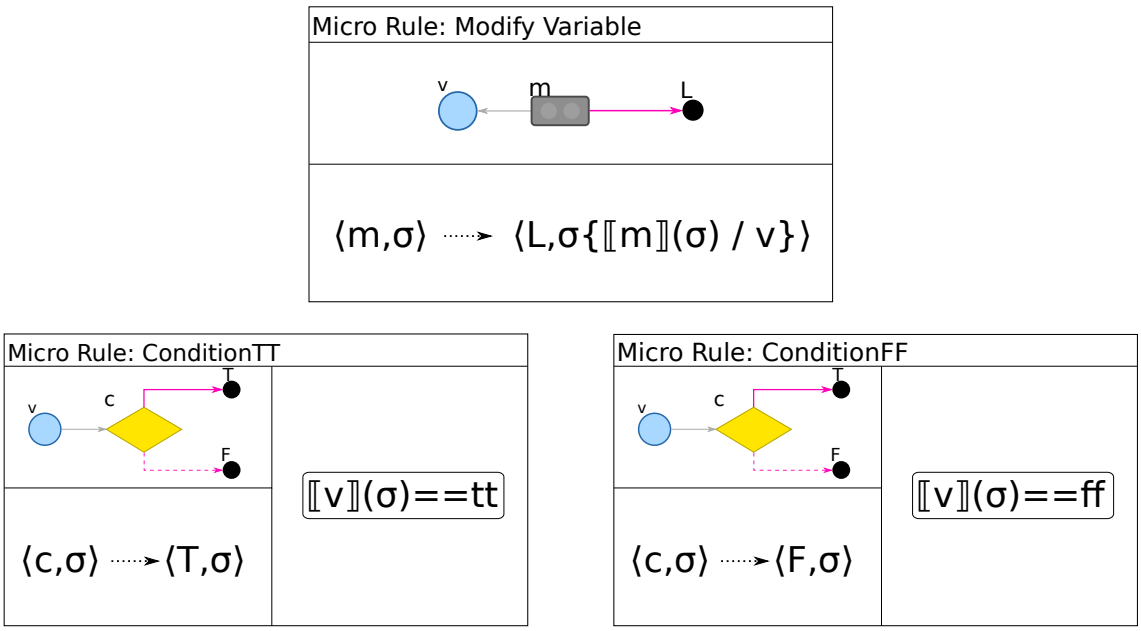


Figure 1.4: Second-level rules defining the aggregation of information in a WebStory.

are defined using identifiers. The meaning of those rules is detailed in Sect. 4, but here I want to emphasize that none of these rules require the explicit modeling of element deletion or creation to simulate the computation aspects.

In contrast, Figure 1.5 shows the rules of Fig. 1.4 encoded in the general-purpose graph-transformation tool *Groove*. Furthermore, Fig. 1.6 explicitly models the comparison of two KTS states. This rule requires for advanced transformation concepts, such as the matching of sub-graphs using quantified rules. The comparison mechanism is a domain-specific aspect of the two-level transformation language and has not to be addressed explicitly by a transformation developer.

The rules represented in Fig. 1.3 and Fig. 1.4 are specialized towards the involved source and target languages and the purpose of the transformation. Thus, for each combination of source and target language, and the transformation's purpose, a dedicated two-level transformation language is required. This leads to my third contribution.

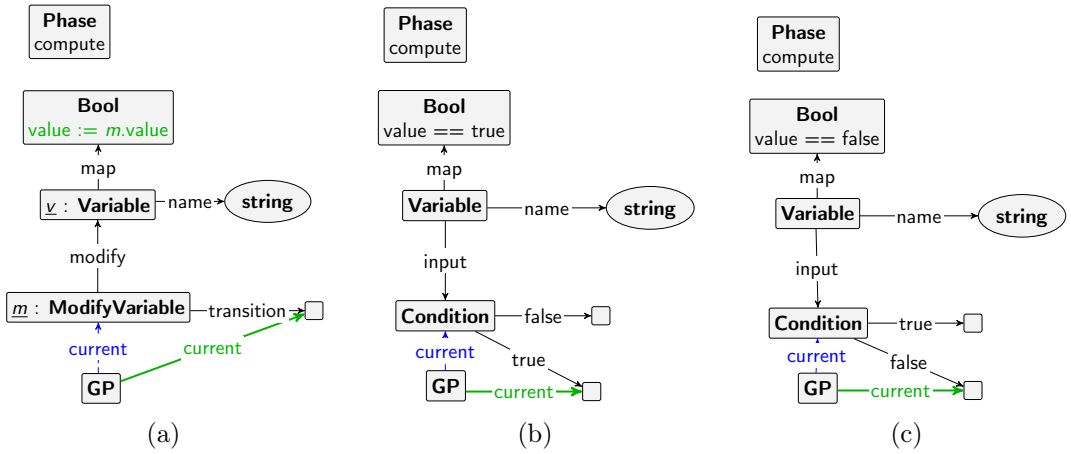


Figure 1.5: Computational rules from the transformation shown in Fig. 1.4 encoded in Groove: (a) `Modify Variable`, (b) `ConditionTT`, and (c) `ConditionFF`. In the rules green elements will be created after rule application, blue elements are matched and deleted after rule application.

3. Generation of Graph-Based Domain-Specific Model Transformation Languages My third contribution is the definition of a generator for the described two-level SOS-based transformation languages. It is implemented under my supervision by Phillip Goldap in the context of his master thesis [Gol19].

The main task during the extraction of a generator for two-level transformation languages facilitating the specification of rules as shown in Fig. 1.3 and Fig. 1.4, is the generalization of concepts constituting the transformation language. Of course, this generalization should not cause a loss of the domain-specific aspects described above. In this process, I studied the constituents of the transformation and the interrelations between them. Having identified the interrelations, I extracted the information which entailed the interrelations and provided means to define this information on a higher specification level, making the generation of two-level transformation languages feasible.

Clearly, the constituents for a computational two-level transformation language are the source language (L_S), target language (L_T), and the SOS-based computation language (L_C). Fig. 1.7 shows the interrelations between them.

- *Memory Initialization*: The auxiliary language facilitates the computation of information. The data that serves as the basis for the computation is defined in the source language. Therefore, a mapping from the source language to the data component of the computation language is required to initialize the memory for the actual computation.

- *Semantics*: The required data for the transformation is not necessarily represented locally in a source model and has to be retrieved by traversing structures of the model. In combination with the memory initialization, the SOS-

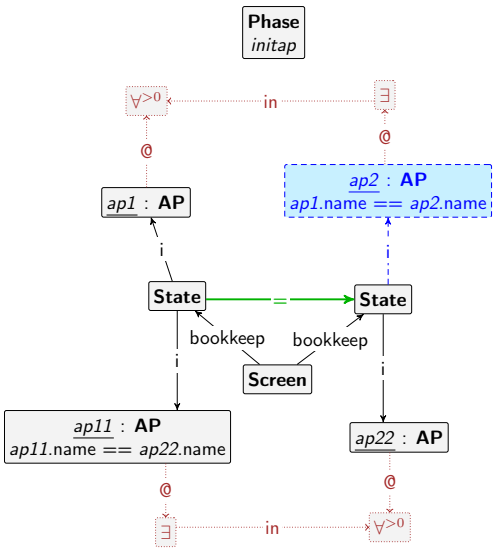


Figure 1.6: Rule for the identification and merging of equivalent KTS states.

language has to be coupled with the source language to allow for the definition of the source language's semantics.

- *Data derivation*: Attribute values of target language elements can depend on information represented in the source model and the computed information provided by the auxiliary language. Therefore, a mapping from source and auxiliary language to the target language is required.

Analyzing these interrelations resulted in the observation that the major parts of the transformation language can be generated, i.e., without the knowledge of the actual source and target language.

The two-level structure of the transformation language is static. The first rule type encapsulates the `Data derivation` between L_S, L_C and L_T, the second one the `Semantics` dependency between L_S and L_C. Furthermore, the structure of the individual rule types is also static. The interrelations between the languages and the design decision to separate the source, target, and auxiliary language imply the segmentation of the rule types. The first rule type is divided into three parts: Source language part, target language part, and computation part. The computation part is used to represent the start configuration of a computation and its result (cf. Sect. 4.1). The second rule type is divided into two parts. One allowing to define a traversal step using the computation language and one for the definition of source language patterns.

The next observations consider the actual causes for the interrelations between the languages. The means to define the `Semantics` interrelation between source and computation language can be generated independent of the actual source language. It is realized by using identifiers in both languages to relate corresponding elements. The extension of source language types by an attribute representing this identifier is

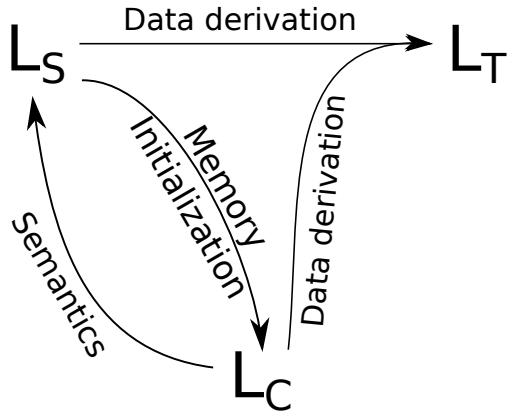


Figure 1.7: Interrelation between the languages involved in a computational transformation.

independent of the actual source language. Consequently, this feature can be added to the specification of an arbitrary source language. The computation language's purpose is to ease the traversal of structures, hence we assume the identification concept as part of the computation language.

The `Memory Initialization` depends on the purpose of the transformation, and thereby on the actual source language. Consequently, the transformation developer has to define which data of the source model should be considered for the transformation before the transformation language can be generated.

The `Data Derivation` depends on the source and auxiliary language (especially on its memory component). Consequently, to derive required data for target elements, the memory initialization has to be given.

To support the developer in the specification of these two interrelations, I provide specification languages tailored towards the concepts of source, target and computation language. I.e, these languages ease the access to the concepts defined in the corresponding languages and support the specification of their relations.

1.2 Related Work

The generation of transformation languages is researched for textual and visual DSL. In [Wei12, HRW15], the authors consider textual domain-specific languages. They describe how transformation languages which re-use the concrete syntax of the base language can be automatically generated. In this approach only one DSL is considered, yielding an endogenous transformation language (cf. Sect. 2.3.1). For visual DSL, the authors in [KMS+10, SGV13] describe the (semi-)automatic generation of transformation languages, considering the abstract and static syntax of the involved DSLs. The pattern languages are derived from the DSLs by relaxation, augmentation, and modification. The relaxation, e.g., decreases lower bounds for references defined by the meta model of the DSL to 0, and changes abstract types

to concrete ones, to prevent the enumeration of all possible concrete sub types in a transformation rule. The augmentation adds attributes to types allowing for their identification and adds concepts which do not belong to the DSLs in form of "generic nodes and links". The modification changes all attribute types to "constraints" and "actions" to specify constraints on pattern elements or derive element values.

The need for model transformations for specific purposes was already recognized by several researchers. Agrawal et al. describe in [AKN+06] a possible design of model transformation languages. They use the concrete syntax of UML class diagrams to specify rule patterns. Cuadraro proposes in [Cua12] to realize transformations by families of transformation languages. Thereby, the individual languages should be designed to provide relevant features to realize a specific aspect or task of the desired transformation. The main focus of this research is the composition and interoperability of the involved languages. The composition of transformation languages creates a new transformation language, the interoperability defines how to process transformation results gained by applying individual transformations of the transformation family. This approach resembles Language-Driven Engineering (LDE) [SGNM19], in which several DSLs are employed allowing to involve stakeholders from different domains in the system's development process. The interplay between the DSLs is realized in a service-oriented fashion. Changing the purpose from "systems" to "transformation languages" and providing DSLs capturing different tasks of transformation language development facilitates their development in a similar fashion as described by Cuadraro.

In [SGV13] and [SVL15], Syriani et al. present the Transformation Core (*T-Core*) framework. In [SV10], Syriani and Vangheluwe identified a set of primitive constructs needed to develop model transformation languages and encapsulated them in the T-Core framework. For instance, *Matcher* and *Rewriter* provide interfaces to define preconditions and rewrite rules, which can be combined to transformation rules. Using T-Core, a transformation language can be realized by implementing the primitive constructs. Each construct defines how a matching is computed, in which order a set of matchings is processed, or how conflicts between rewrite rules should be resolved.

To the best of my knowledge, the existing graph-based transformation approaches hold on to the traditional rule definition style of left-hand side and right-hand side pattern specification. All further aspects are handled specifically by, e.g., rule scheduling definitions, or by falling back to imperative textual specifications (hybrid approaches).

1.3 Organization of this Thesis

Section 1.4 describes the context of the attached publications. In Chapter 2, I describe the technical framework in which the transformation languages were implemented (Sect. 2.1), the DSVL used to exemplify the generation of a two-level transformation language (Sect. 2.2), and briefly explain the concepts of model and graph

transformations and exemplify graph transformations by means of a transformation for the WebStory language (Sect. 2.3). Chapter 3 describes the need for graph-based domain-specific transformation languages by listing several projects in which the semantics definition, model migration and model co-evolution, and validation and verification are realized using model transformations (Sect. 3.1). The improvements towards domain-specific transformation languages are described in Sect. 3.2 concluding with graph-based transformation languages for DSVL. Chapter 4 describes my second and third contribution, i.e., the two-level transformation language pattern and the generation of two-level transformation languages. Thereby, Sect. 4.1 recapitulates the pattern for two-level transformation languages and explains the link to Plotkin's SOS. The generation of two-level transformation languages is described in Sect. 4.2.

1.4 Context of Attached Publications

This section gives an overview of the publications belonging to this cumulative dissertation. They exemplify the application of domain-specific approaches in several areas. At a meta-level the domain-specific approach is applied to the domain "development of domain-specific visual languages" [NLKS17] yielding the CINCO language workbench. CINCO is used to develop domain-specific modeling languages (CINCO products) [BFK+16, LKZ+18] and specifically tailored transformation languages [KLNS20, LKS18].

CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools

This paper introduces the CINCO meta tooling suite. CINCO constitutes the foundation of this thesis by providing a fully functional generation of domain-specific graphical modeling tools from abstract specifications described using a text-based DSL. It provides a technical framework to realize all ideas presented in this thesis in the context of DSVL. The motivation for the development of CINCO – the desire for domain-specific, full-generation, and service-orientation – quickly expanded on the accompanying tasks such as the realization of code generators, definition of specific DSVL functionalities which can not be expressed using CINCO's specification languages, and model transformations for the considered DSVL. This desire has driven my research towards the concepts and tools easing the definition of solutions to accomplish the mentioned tasks and resulted in the transformation languages presented in this thesis.

A Tutorial Introduction to Graphical Modeling and Metamodeling with CINCO

This paper shows the capabilities of CINCO using a simple educational example language called WebStory. The specification and extension of the WebStory language is described using the *Graphical* CINCO *Specification* language (GCS) [Fuh18], which is a CINCO product (a DSVL) facilitating a WYSIWYG-driven definition of CINCO product specifications. Using the GCS language for the development of CINCO products enables the application of the concepts presented in this thesis, i.e., graph-based domain-specific model transformation languages, on a higher meta-level, i.e., for CINCO product *specifications*. Consequently, for instance, required preprocessing steps on CINCO product specifications for CINCO's code generator can be defined in an declarative way.

DIME: A Programming-Less Modeling Environment for Web Applications

The *DyWA Integrated Modeling Environment* (DIME) is the largest CINCO product developed thus far. It is used for modeling of single-page web-applications providing three languages for *data*, *user interface*, and *business process* modeling. During the development of DIME the need for mechanisms supporting activities such as model migration, model co-evolution, and model-checking arose, which motivates the research towards model transformations as a possible solution.

The development of DIME establishes an additional way for the semantics definition of new DSVLs. The concepts presented in this thesis can be used to define languages' semantics by means of translational semantics, i.e., transformations between new DSVLs and DIME.

Design for 'X' Through Model Transformation

The first ideas of the two-level transformation language are introduced in this paper. It motivates transformations as main tool to achieve *X-by-Construction*, a generalization of *Correctness-by-Construction*. Properties, 'X', are learnability, performance, and model-checkability.

Towards Language-to-Language Transformation

This paper is the extension of [LKS18]. It presents the the connection of CINCO models to the algebraic graph transformation approach. It describes the model transformation needed for verification and compares the two-level transformation language with the traditional graph transformation approaches.

2 Background

This dissertation targets the practical scope of software development, i.e., model-driven engineering and model transformations. The ideas presented in this thesis are independent from a concrete technical framework, but of course are also implemented. In this chapter, I present the technologies used as the basis for the realization and introduce the relevant languages and formalism to get a basic understanding for the topics addressed in this thesis.

2.1 Cinco Meta Tooling Suite

In DSM, for every problem domain a DSVL is required. Instead of implementing the DSVL manually, language workbenches [Fow05] are used. These are specialized tools supporting the construction of DS(V)Ls, and thereby help to reduce their development costs.

The Cinco meta-tooling suite is the result of applying the Domain-specificity, Full code generation, and Service-orientation (DFS) [Nau17] approach to the domain "development of graphical domain-specific visual languages". Figure 2.1 shows an abstract overview on the artifacts incorporated in the development of Cinco and DSVLs developed using Cinco (called Cinco products). Cinco is build on the basis of *Eclipse* [1] technologies, mainly the *Eclipse Modeling Framework* (EMF) [SBPM08] and *Xtext* [6]. EMF is an implementation of the *Meta Object Facility* (MOF) [11], a standard defined by the *Object Management Group* (OMG) [10] for the realization of modeling languages. EMF facilitates the creation of (meta) models using *Ecore*. On the basis of Ecore (meta) models, textual languages are developed using Xtext. The abstract and concrete syntax of a DSL is represented by Ecore meta models and Xtext grammars, respectively.

Cinco provides the *Meta Graph Language* (MGL) and *Meta Style Language* (MSL) for abstract and concrete syntax specification of graph-based visual modeling languages. Both languages are realized using Ecore and Xtext and thereby conform to Ecore's meta model (cf. Fig 2.1 `MGL.ecore` and `Style.ecore` conform

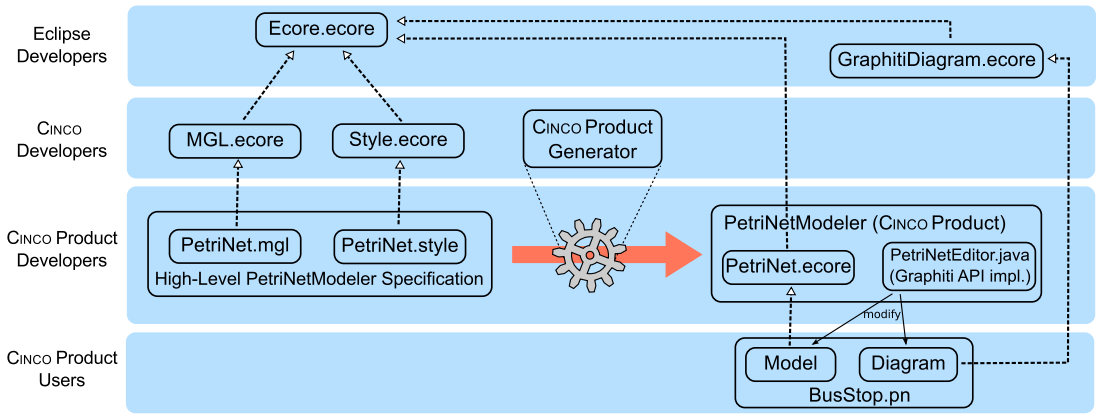


Figure 2.1: High-level view on Cinco [NLKS17].

to `Ecore.ecore`). In Figure 2.1 the Cinco *product Developers* level exemplifies the realization of a DSVL on the basis of Petri nets [Mur89]. The `PetriNet.mgl` and `PetriNet.style` define the abstract and concrete syntax for Petri nets. The specifications conform to `MGL.ecore` and `Style.ecore` and serve as input for the Cinco *product generator*, which generates a fully functional modeling tool on the basis of Ecore and *Graphiti* [4]. Models created by the generated modeling tool (cf. `BusStop.pn` on the Cinco *Product Users* level) conform to the Petri net meta model (`PetriNet.ecore`).

The main specification constructs in MGL are *node* and *edge* types, which are defined in a dedicated *graph model* type. Additionally, Cinco provides the notion of *container* types which can in turn contain node and container types. Furthermore, one can specify *user defined* types and *enumerations*. Types can contain primitive attributes, e.g., Integer, Boolean, String or complex attributes, e.g. a user defined types (compositions of primitive attributes and types), enumeration, or other node, edge, and container types. In Cinco, service orientation is realized by *prime references*. A prime reference points to an instance of a previously defined type.[1] To define the concrete syntax (visual appearance of elements specified in an MGL model) of a DSVL, the MSL is used. The main elements are *nodeStyle*s and *edgeStyle*s. The former are used to define a (hierarchical) representation using geometric figures, e.g., *ellipse*, (*rounded*) *rectangle*, *polygon*, *text* and *image* elements. The latter specify the appearance of edges, i.e. attributes like line style, line width etc. Edges can be decorated by specific elements to allow for, e.g. the labeling of edges. For detailed information on Cinco we refer the reader to [NLKS17, LKZ+18, Lyb19].

[1] Additional to types defined with Cinco, prime references can refer to arbitrary types conforming to Ecore's meta model.

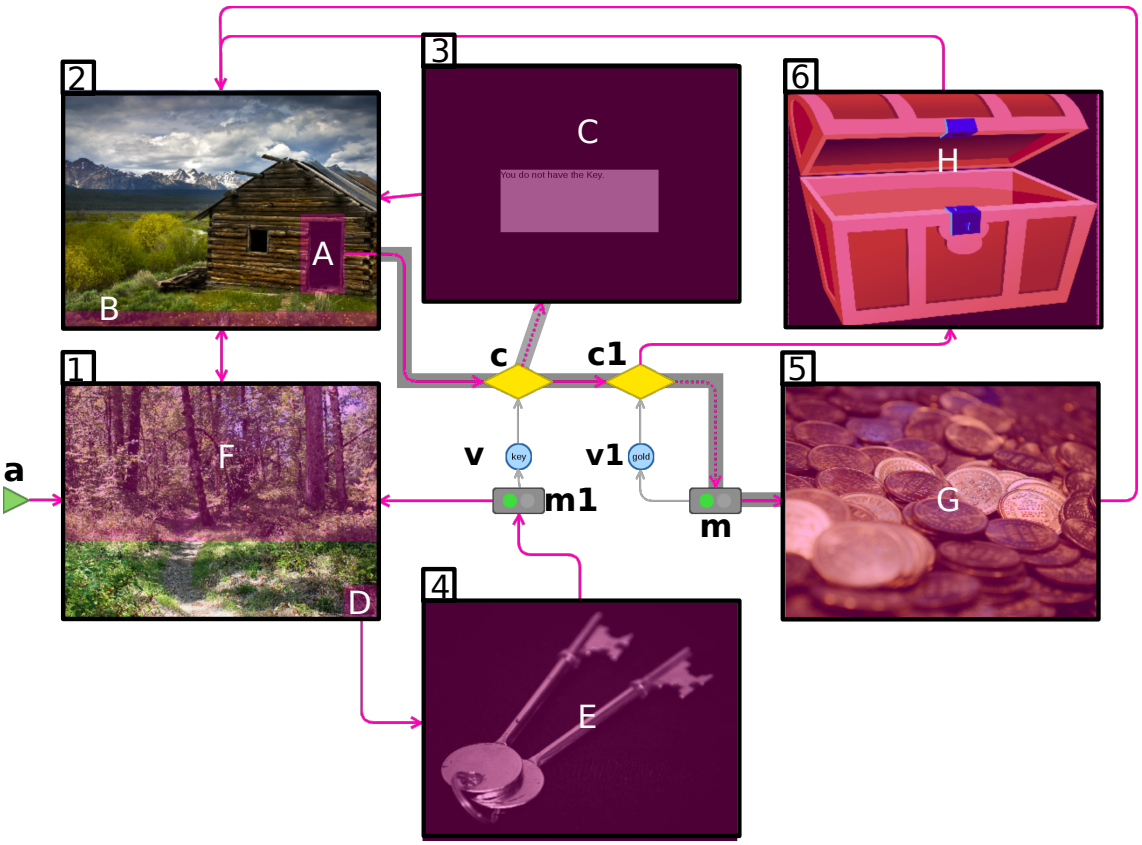


Figure 2.2: A WebStory language model, representing a small gold-hunt adventure from [KLNS20].

2.2 The WebStory Language

To present the capabilities of CINCO we developed the WebStory language. In workshops we want to teach students the advantages of model-driven development and domain-specific languages, wherefore the WebStory language is easy to understand and intentionally uses only a small set of different types. It facilitates the creation of web-based point&click adventure games providing modeling elements such as *Screens*, *Click Areas*, *Text Areas*, *Condition*, and Boolean *Variables* which can be modified using *Modify Variable* nodes. Fig. 2.2 shows a model of an adventure in which the player has to find a key to get hold of a treasure. The typing of nodes (and containers) is given as follows:

- `1-6` are *Screens*,
- `A-H` are *Click Areas*,
- `c, c1` are *Conditions*,
- `m, m1` are *Modify Variables*,

- `v, v1` are *Variable*s,
- `a` is a *Start Marker* and
- *Screen* 3 contains a *Text Area*

Furthermore, the edges are typed as follows:

- Outgoing edges of *Click Area* nodes are *Transition*s,
- outgoing solid edges of *Condition* nodes are *True Transition*s, outgoing dashed edges are *False Transition*s, and
- gray edges, e.g. `m`→`v1` (writing) and `v1`→`c1` (reading), are *Data* edges.

2.3 Model Transformations

Kleppe et al. provide in [KWB03] the following definition of model transformations:

"A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."

With the introduction of model-driven engineering approaches a need for model transformations has arisen. For instance, in MDA model refinement is applied to transform Platform Independent Models (PIMs) to Platform Specific Models (PSMs) eventually resulting in a full-fledged implementation. Tom Mens justifies the need of model transformations listing eight activities in MDE which require "languages, formalisms, techniques, processes, tools and standards that support model transformations" [Men13]. "Model migration and co-evolution" and "Model checking, verification and validation" are two of these activities which we use as motivation in [LKS18, KLNS20]. Regarding the definition of Kleppe et al. it is not surprising that various approaches emerged to support model transformations. In [MVG06, CH06], the authors propose several categories to classify existing model transformation approaches. In [KBRC+18], Kahani et al. identified 60 model transformation tools.[2] They classified them regarding, i.a., the type of transformation rule specification. Among the 60 tools, 15 are graph-based. The categories into which the tools are classified have a large overlap with the categories identified in [CH06, MVG06, Men13].

In this section, I will give a brief overview of the categorization, that has driven my research in this area. Afterwards, I briefly recapture graph transformations, to provide a basic understanding for the ideas presented in this dissertation.

[2]The authors consider Model-to-Model and Model-to-Text transformation tools.

2.3.1 Categorization of Model Transformations

Endogenous vs Exogenous A model transformation is categorized depending on the source and target language(s). An *endogenous* transformation transforms models within the same language. A typical example for an endogenous transformation is refactoring: A model is transformed to meet syntactical properties without changing its behavior. A transformation between different languages is *exogenous*. The transformations given in [KLNS20], from the WebStory language to the language of Kripke Transition Systems (KTS) [MSS99] is exogenous.

In-place vs Out-place An *in-place* transformation is executed within the same model, whereas an *out-place* transformation creates a new target model. A typical example of in-place transformations is refactoring, where the model is usually updated in-place to meet particular syntactic conditions. The compilation process is a typical out-place transformation. The source code of a high-level programming language is transformed to assembler. Thereby, the source model is not altered, but a new model is created.

Rule Definition Transformation rules describe how constructs in the source language are transformed to constructs in the target language. They can be specified in different ways. For instance, a model can be represented internally and *directly manipulated* using an API. This way, transformation rules are implemented in a high-level programming language and executed in the context of an imperative transformation implementation. I.e., the transformation developer describes *how* to transform models and not *what* should be transformed.

More specialized transformation languages provide means to specify rules focusing on the relationship between elements of the source and target model. The *Atlas Transformation Language* (ATL) [JABK08] is a textual DSL for model transformations. The language supports declarative rule specification with the possibility to include imperative parts.

Rules in model-to-text transformations are usually specified using a dedicated template-based language, instead of programming string concatenations directly in e.g. Java. Templates consist of static and dynamic parts. The static parts are independent from the source model and describe, e.g., the constant text in a class definition. The dynamic parts define the traversal and value evaluation of the source model.

For model-to-model transformations in MDE, a natural way to specify transformation rules is to use a graph-based approach. Graph transformations are detailed in the following section.

2.3.2 Graph Transformations

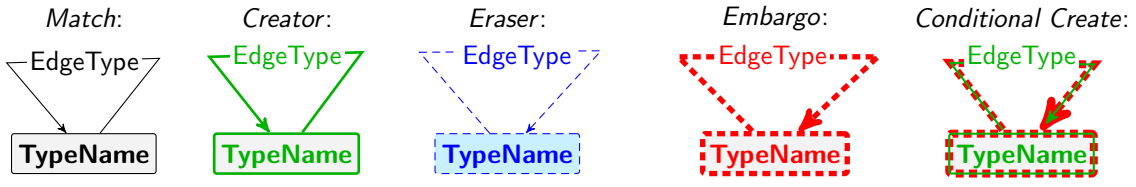
In model-driven engineering, graph transformations are suggested as the natural choice for the specification of model transformations. Ehrig et al. provide in [EEGH15]

a mapping for notions between meta modeling and the algebraic graph terminology, implying a strong interrelationship between both areas. A graph transformation consists of a set of transformation rules, where a rule is defined by a *Left-Hand Side* (LHS) and a *Right-Hand Side* (RHS) graph. The LHS and RHS of a rule represent graph *patterns*. Given an input graph, a rule is applied by searching the pattern defined in the LHS of the rule in the input graph and replacing the LHS by the RHS. Finding the LHS in the input graph is called *matching*. The occurrence of nodes in the LHS and/or the RHS defines the following meaning. A node is

- created, if it is not in the LHS, but RHS of the rule,
- deleted, if it is in the LHS, but not in the RHS of the rule,
- matched, if it is in LHS and RHS of the rule.

In practice, a more accurate control of rule application is desired. Therefore, rule application conditions are used, e.g., constraints for type attributes or *Negative Application Condition* (NAC). A rule is applicable, if the LHS is matched in the input graph, the defined conditions are satisfied by the match and the structure described in the NAC is not matched in the input graph.

I exemplify graph transformations specifying two transformation rules based on the WebStory language. The transformation's purpose is to make a story "traceable". After a *Screen* node is visited more than once, a *Text* element showing the String "Visited" should be displayed. Figure 2.3 shows the transformation rules modeled in Groove [Ren04]. Please note, that Groove is a general-purpose graph transformation tool. Hence, the types in the rules for the WebStory transformation are not defined using the concrete syntax of the WebStory language, but using a generic one. Node types are displayed in bold font inside the nodes. In addition, Groove does not comprise the concept of *Container* types. Consequently, containment is expressed by a "contains"-labeled edge, pointing from the container to the contained element. In Groove, rules are not specified explicitly in LHS, RHS, and NAC, but in one representation using a color encoding to express the membership of elements to LHS, RHS, and NAC. Let e be a node or an edge type, then the affiliation of e to LHS, RHS, an NAC is represented as follows.



$e \in LHS \wedge e \in RHS \wedge e \notin NAC$ | $e \notin LHS \wedge e \in RHS \wedge e \notin NAC$ | $e \in LHS \wedge e \notin RHS \wedge e \notin NAC$ | $e \notin LHS \wedge e \notin RHS \wedge e \in NAC$ | $e \notin LHS \wedge e \in RHS \wedge e \in NAC$

The names used in Groove for the rule types are depicted above their graphical representation (*Creator, Eraser, Embargo, Conditional Create*). The rule depicted

in Fig 2.3(a) creates a *Screen* node if no *Screen* with the same `backgroundImage` exists, and adds a *Text Area* element to the newly created *Screen* node. The *Text Area* element's value is set to "Visited". Additionally, the rule creates a *Variable*, *Modify Variable*, and *Condition* node and connects them in the following way. If the *Screen* node was not visited yet (the *Variable* node's value equals `false`) the *False Transition* edge defines the next node in the control flow. Thus, the *Variable* node's value is set to `true` by the *Modify Variable* node and in the next step the original *Screen* node is displayed. Reaching the created *Condition* a second time will result in displaying the *Screen* node containing the *Text Area* element. The second rule (cf. Fig. 2.3(b)) redirects an incoming *Transition* edge of the original *Screen* node to the *Condition* node, by deleting the original *Transition* edge (dashed blue) and creating a new *Transition* edge connecting the source node of the original transition and the *Condition* node. Furthermore, for *Click Area* nodes in the original *Screen* node, a *Click Area* in the new *Screen* node is created and connected to the same target.

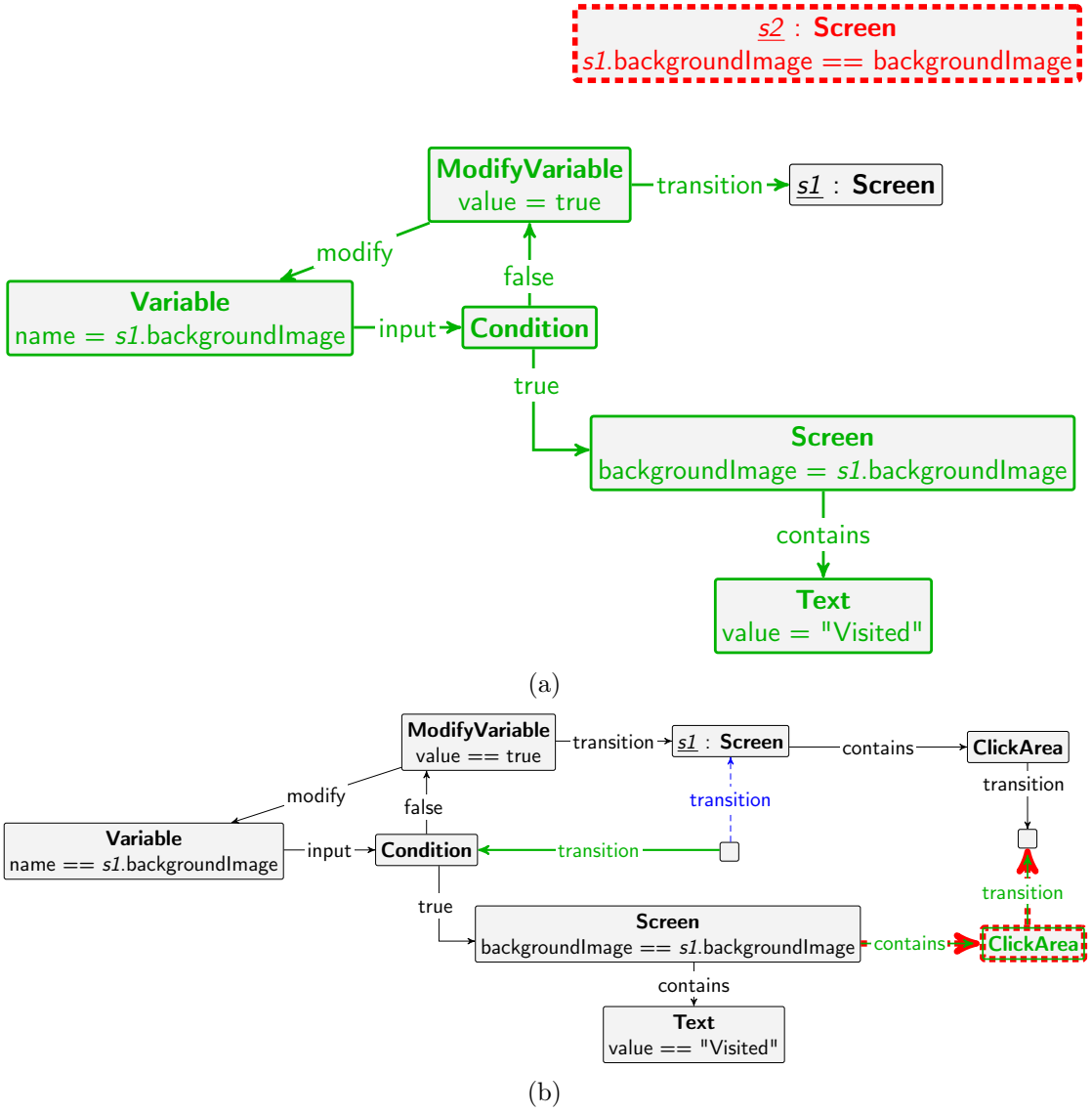


Figure 2.3: "Traceability" transformation of a WebStory model specified in two rules. Rule (a) creates the required logic structure, Rule (b) updates the control flow.

3

Towards Graph-Based Model Transformations in the Cinco Framework

As described by Mens in [Men13], model transformations are used for several activities in MDE. These activities comprise *semantics definition*, *model migration and model co-evolution* [HBJ09, CREP08], and *validation and verification*. They arise during the development of Cinco and Cinco products, therefore they are the main motivation for the development of transformation languages in this context. The individual activities are characterized as follows.

Semantics Definition In programming language theory, the semantics of a programming language defines the meaning of programs written in the corresponding language. Plotkin described the semantics of the *While* language by rules defined in Structural Operational Semantics [Plo81]. The meaning of a *While*-program is its effect on an initial memory configuration.

Model Migration and Model Co-Evolution The conformance relation between models and their meta model [Bé05] can be destroyed by changes made to the meta model [Fav05]. It is repaired by migrating the models to conform to the modified meta model. The required model changes can be specified by model transformations. For some meta model changes the corresponding models can be co-evolved, i.e., the transformations repairing the conformance can be automatically generated.

Validation and Verification Often, the conformance of a model to its meta model does not imply the model's plausibility. Providing model validations for a DS(V)L supports the development of meaningful models. For instance, undesired loops in the control flow of a model expressed in a DSVL can be identified by defining a pattern representing the loop. The transformation engine can be used to find the pattern in the models. Formal verification methods like *Model Checking* [CGL94] are used to check desired model properties. Given a (formal) model M of a system and a property (formula) φ defined in *Linear Time Logic* (LTL) or *Computation Tree*

Logic (CTL), a model checker verifies if the model satisfies the property, written as $M \models \varphi$. Usually, the formal model M is represented by a finite state transition system which serves as input for a model checker.

The model transformations for the described tasks can be realized in different ways (cf. Sect. 2.3). However, following the simplicity approach, CINCO should support an easy way to define those transformations without the need of knowing technical details of the technologies on which CINCO is based. For DSVL, graph-based model transformations are perfectly suited for these needs [CH06, KMS+10, AKS03].

On the CINCO *Product Developers* level, the mentioned tasks reflect themselves as follows.

- *Semantics definition*: Given the abstract and concrete syntax of a CINCO product specified in MGL and MSL, respectively, the developer has to define the semantics of the language. For this purpose, model-to-text and model-to-model transformations are applied (cf. Sect. 3.1.1).

- *Model Migration and Model Co-Evolution*: Changing a CINCO product's specification leads to the problem of model migration. This problem was targeted by Till Schallau in his master thesis [Sch19]. He (semi-) automatically generates transformation rules to repair the conformance relations between models and the DSVL (cf. Sect. 3.1.2).

- *Verification*: The application of model checking on models expressed in a DSVL is not possible without previously transforming them into an appropriate input format for a model checker (cf. Sect. 3.1.3).

The area of application of model transformations is not limited to the development of CINCO products, but also to CINCO itself, since CINCO is a domain-specific tool. One major task is the semantics definition for MGL and MSL. Similar to the semantics definition of CINCO products, this is realized using model-to-model and model-to-text transformations. Additionally, model migration also plays an important role in the development process of CINCO. Changing the meta model of the MGL or MSL could invalidate existing CINCO product specifications. Currently, these tasks are solved by hand (model migration) or by imperatively implemented model transformations (semantics definition). To allow for graph transformations on the CINCO Developers level, a graph-based representation of MGL and MSL is needed. The first step towards this goal was realized by Annika Fuhge in her bachelor thesis [Fuh18] developing the *Graphical* CINCO *Specification* language (GCS), a CINCO product for modeling of MGL and MSL in a graphical model (cf. Sect 3.1.1).

In the following, I will present several projects, bachelor, and master theses in which model transformations are used to handle the described task (cf. Sect. 3.1). In Sect. 3.2, I describe the improvements to support the specification of model-to-text and model-to-model transformations in CINCO.

3.1 Model Transformations in Cinco

This section gives an overview of projects and theses realized in the recent years, which used model transformations to solve a specific task.

3.1.1 Model Semantics

The definition of semantics is crucial in the development of CINCO and its products. They can be defined by means of code generators translating models into executable programs written in, e.g., a high-level programming language, which is a typical model-to-text transformation. Their realization can be supported by model-to-model transformations, e.g., through the refactoring of the source model. For instance, flattening the inheritance hierarchy reduces the complexity of the generation logic, by previously moving the attributes of super-types to their sub-types.

Alternative to code generators, the semantics can be specified utilizing an existing DSVL. Thereby, models expressed in the new DSVL are translated to models of a DSVL with existing semantics.

During the recent years, we had several bachelor and master theses, as well as *Projektgruppen*[1] in which both approaches were applied to develop new DSVLs.

CINCO and Pyro Stefan Naujokat described in his Ph.D. thesis the *Domain-specific, Full generation, and Service orientation* (DFS) approach "for tools to facilitate simplicity-driven model-based software development" [Nau17]. Furthermore, he argued to "... apply the very same DFS concepts to the domain of modeling tool development itself". The result of this application is CINCO. The domain-specificity is reflected in the languages provided by CINCO which facilitate the development of modeling tools for models that can be represented as graph structures. Thus, the *semantics* of a specification written in CINCO is given by a code generator yielding a fully functional modeling tool. In Fig. 2.1, this generation is depicted on the CINCO *Product Developers* level. An MGL model (`PetriNet.mgl`) and an MSL model (`PetriNet.msl`) are transformed into a meta model for the CINCO product (`PetriNet.ecore`) conforming to Ecore's meta-meta model (`Ecore.ecore`) and a *Graphiti* API implementation. The generator is implemented on the CINCO Developer level and realizes the former transformation as a model-to-model transformation, since an Ecore meta model is created using the EMF API. The latter is a model-to-text transformation, realized as a code generator whose output is Java code, implementing the Graphiti API. Using Ecore as target language allowed us to build on the existing Ecore code generators and take advantage of the numerous features provided by EMF, such as the persistence mechanisms of EMF and the notification system using the EMF-generated Java classes. Furthermore, using the Graphiti API as framework for the modeling editor eased the generation pro-

[1] *Projektgruppen* are courses at the TU Dortmund University, where up to 12 students develop a more ambitious project during one year.

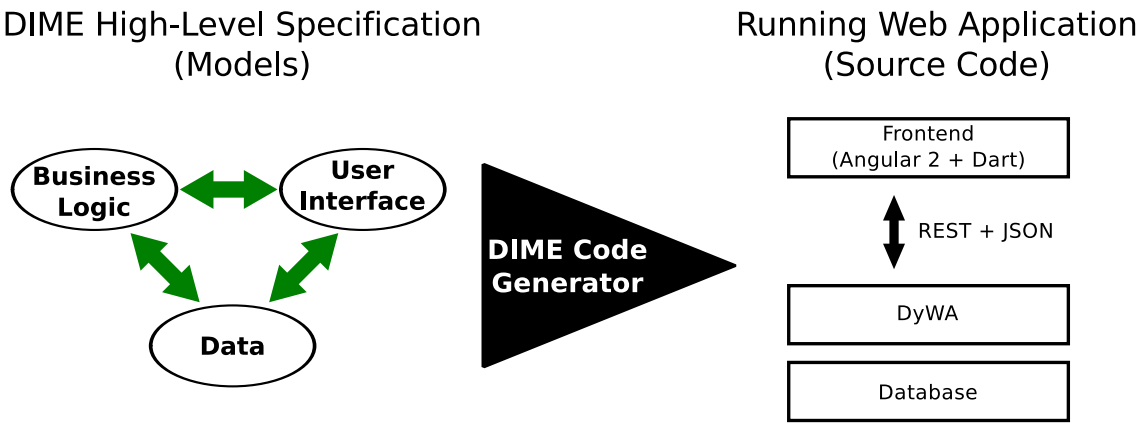


Figure 3.1: High-level view on the DIME code generation from [BFK+16].

cess, since the API provides an abstraction from the *Graphical Editing Framework* (GEF) [2] and *Draw2D* [3], hiding the technical details behind a plain Java API.

With *Pyro* [Zwe15, ZNS19], Philip Zweihoff implemented in his master thesis a generator, which, given a CINCO product specification, generates a corresponding web-based modeling tool. This corresponds to the exchange of the generation target in Fig. 2.1. Instead of generating EMF models and a Graphiti implementation, he generated a meta-schema for the *Dynamic WebApplication* (DyWA) [NFSM14, Fro13] and a model editor based on the *JointJS* framework [9].

DIME DIME [BFK+16] is the most complex CINCO product developed at our chair. Its purpose is the development of single-page web-applications. It consists of languages for *data, business logic*, and *user interface* modeling, and employs service-oriented development using *service libraries*. Fig. 3.1 shows a high-level view on the generation process in DIME. The target of the generation is a web application employing several technologies. DyWA forms the back-end of the generated application. A meta data schema is generated from DIME *data* models and constitutes the persistence layer of the web-application. The front-end is built on *Angular 2* and *Dart* and is mainly generated from the *user interface* models. As the name says, the *business logic* defines the application logic, i.e., what data should be displayed/collected on which pages and how to process it.

WebStory The semantics of the WebStory language (cf. Sect. 2.2) is defined by a code generator producing the adjacency matrix of nodes in a WebStory model. A static framework evaluates the generated matrix to determine the next *Screen* node in the model. Therefore, it implements the semantics of the types defined in the WebStory language, e.g., *Condition* types have to evaluate their connected *Variable* node to determine the next element in the control flow.

In [Kue18], Dennis Kühn describes an approach to transform DSVLs implemented in CINCO to DIME models, enabling their execution in a web-based environment.

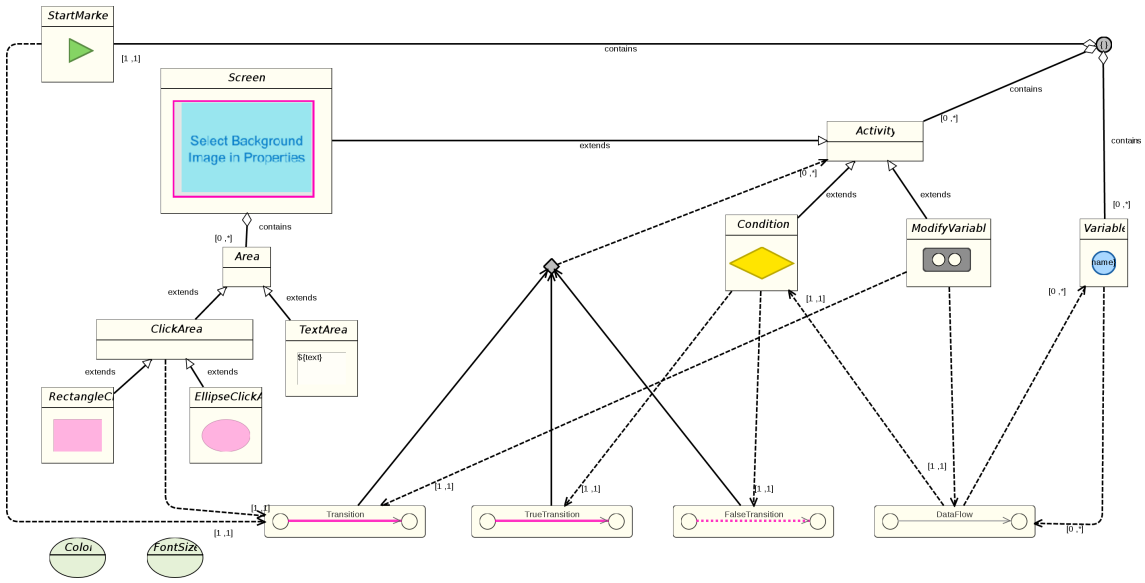


Figure 3.2: Specification of the WebStory language using the *Graphcial* Cinco *Specification* tool.

He uses the WebStory language as an example for this transformation.

Graphical Cinco Specification (GCS) In her bachelor thesis [Fuh18], Annika Fuhge developed under my supervision a modeling tool, which enables the graphical specification of MGL and MSL models in a WYSIWYG manner. Figure 3.2 shows the WebStory language specification created in GCS. The abstract and concrete syntax of the WebStory is defined in one model. Node and container types are depicted as boxes defining the type name at the top of the box and including their concrete syntax in the bottom part. Edge types are specified by a box containing two circles connected by an edge that represents the edge type's concrete syntax (cf. Fig 3.3). The name of the edge type is specified above its concrete syntax representation.

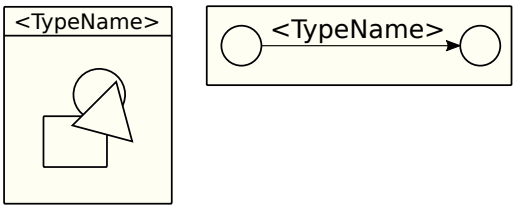


Figure 3.3: Node type (left) and edge type (right) definition in the GCS.

Figure 3.4 illustrates the development process of the GCS tool, and the generation process for Cinco products defined in GCS. The modeling tool itself was developed using Cinco, i.e., it is a Cinco product which can be used to graphically specify Cinco products. The top part of Fig. 3.4 illustrates the development of the GCS

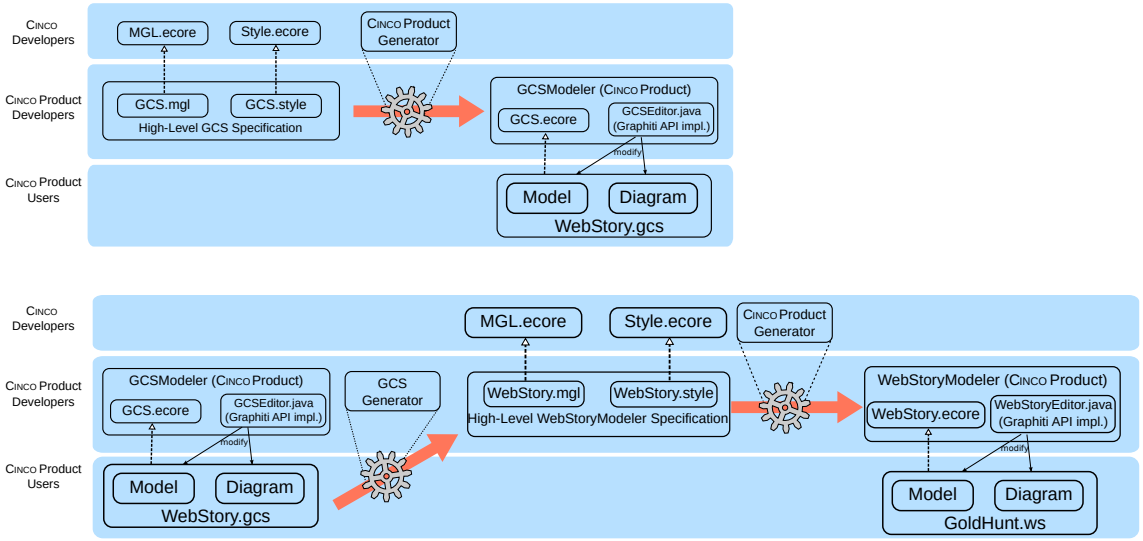


Figure 3.4: Development of the *Graphical* CINCO *Specification* tool (top) and specification of the WebStory modeling tool using GCS (bottom) [Fuh18].

tool, consisting of `GCS.mgl` and `GCS.style` which are generated to a fully functional modeling tool (cf. Sect. 2.1).[2] The generation process of GCS models consists of two steps. The bottom part of Figure 3.4 exemplifies those steps for the WebStory language. In the first step, the GCS specification of the WebStory language (`WebStory.gcs`) is transformed to the CINCO specifications for a CINCO product, namely `WebStory.mgl` and `WebStory.style`. Afterwards, the existing CINCO generators are used to generate the resulting WebStory modeling tool. Fuhge realized the transformation from GCS models to CINCO specifications as a model-to-text transformation, generating the corresponding textual representations of MGL and MSL models. Of course, this transformation can be realized as a model-to-model transformation between GCS models and an in-memory representation of MGL and MSL models.

CMMN In his master thesis, Jan Weckwerth developed a *Case Management and Modeling Notation* (CMMN) [7] tool on the basis of CINCO. He specified the semantics through a transformation from CMMN models to XML files, which are interpreted by the *Camunda* [12] engine. In a subsequent thesis, Todor Nikolov developed a model transformation from CMMN to DIME, enabling the execution of CMMN models in a web-application context.

[2]The Eclipse Developer level is omitted in the figure.

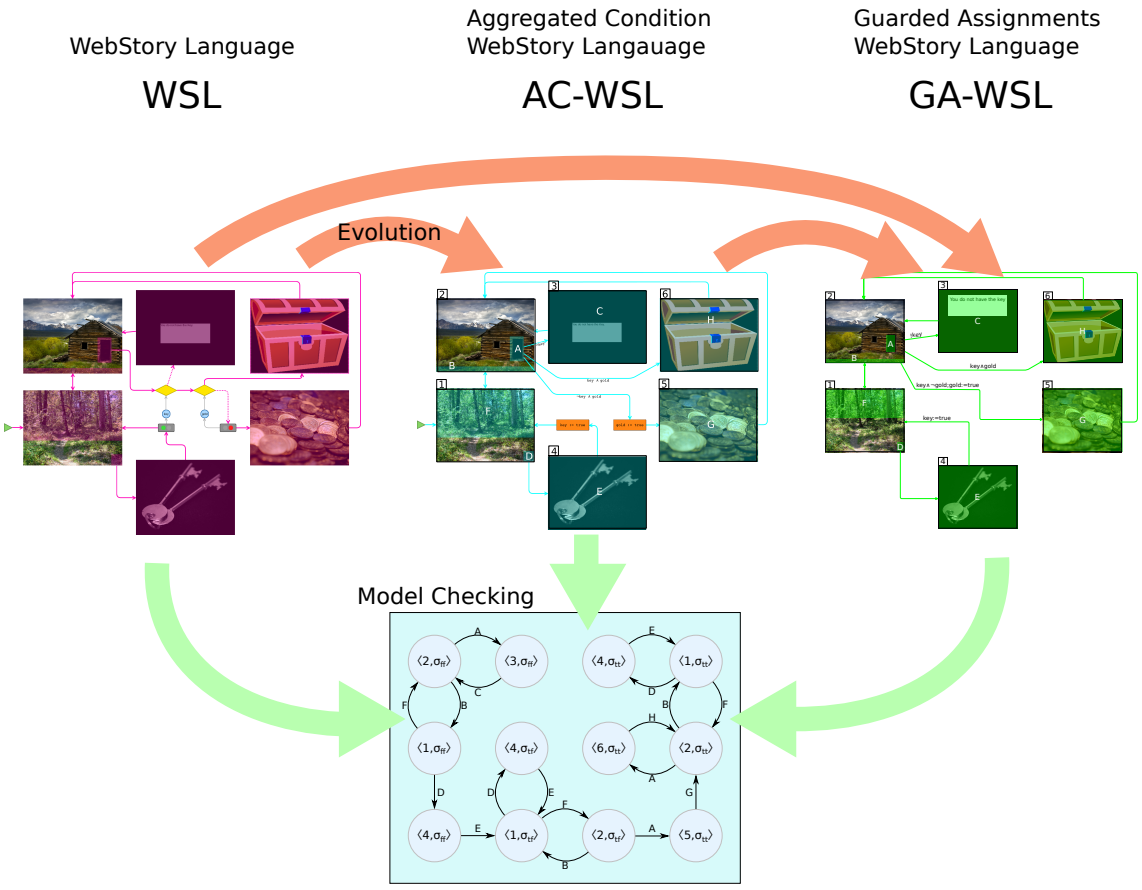


Figure 3.5: Overview of language evolution and the required transformations to facilitate formal verification for the evolved languages.

3.1.2 Meta Model and Model Co-Evolution

A major challenge during model-driven development using DSLs is that not only models conforming to a language evolve, but the language itself is a target to changes [Fav05]. Some of these changes may break the model conformance of existing models to their meta models. For instance, deleting a type from the language results in the invalidation of models containing an instance of this type. A central challenge is to restore the conformance between models and their meta models [MWD+05].

Figure 3.5 shows evolution steps of the WebStory language (orange arrows), and the transformations to Kripke Transition Systems (KTS) represented by green arrows. The `Evolution` from the *WebStory Language* (WSL) to the *Aggregated Condition WebStory Language* (AC-WSL) invalidates the conformance of models expressed in WSL, by removing the *Condition*, *Variable* and *Modify Variable* types. Conditions modeled in WSL are represented by Boolean expression labels on an augmented *Transition* edge type in AC-WSL (blue edges in AC-WSL model). Variable modifications are expressed using a new *Assignment* type (orange nodes in AC-WSL model).

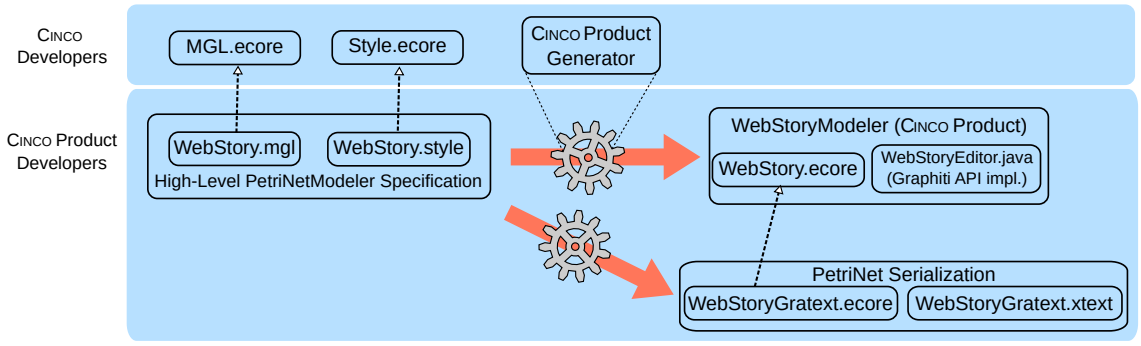


Figure 3.6: Generation process of the GraText serialization format for CINCO products.

In the early days of CINCO, opening models in the generated model editor, which did not conform to the updated specification, raised an exception. This was caused by the persistence mechanism of EMF: Given an Ecore meta model, EMF generates an XML Schema Definition describing this meta model. Trying to open a model which does not conform to the editor's supported schema definition leads to an exception. Consequently, the invalidated models had to be migrated to repair the conformance relation. We encountered this problem in the development of several CINCO products. It had the largest impact during the development of DIME, since in parallel, DIME applications were also modeled. The first migrations of DIME models were performed by manually repairing their XML representations, which was a very tedious task. To ease the migration process, Steve Boßelmann developed a textual representation for CINCO product models (graphs), called *GraText*. Figure 3.6 shows the generation of the serialization format. Given a CINCO specification, an additional meta model (`WebStoryGratext.ecore`) and grammar specification (`WebStoryGratext.xtext`) forming the basis for an Xtext-generated textual editor are generated. Using this serialization format allows for a more comfortable manual migration process.

Of course, the migration for the `Evolution` of WSL to AC-WSL (cf. Fig 3.5) can be described by a model transformation. Instantiating the transformation language presented in this thesis using WSL and AC-WSL as source and target language of the transformation, results in a language facilitating the definition of rules as shown in Fig. 3.7. The two rules at the bottom of the figure represent the required computation for the migration: they compute the Boolean expression (b_{exp}) by constructing a conjunction of *Variable* nodes which are evaluated by *Condition* nodes. The four rules at the top of Fig. 3.7 represent the relation between WSL and AC-WSL elements.

Till Schallau implemented in his master thesis [Sch19] a DSL to describe changes in an MGL model. The changes can be *non-breaking*, *breaking and resolvable*, and *breaking and non-resolvable* [GKP07]. Non-breaking changes in the meta model do not require model migration. Using his DSL, the rules to repair breaking and resolvable changes can be derived automatically. Breaking and non-resolvable changes

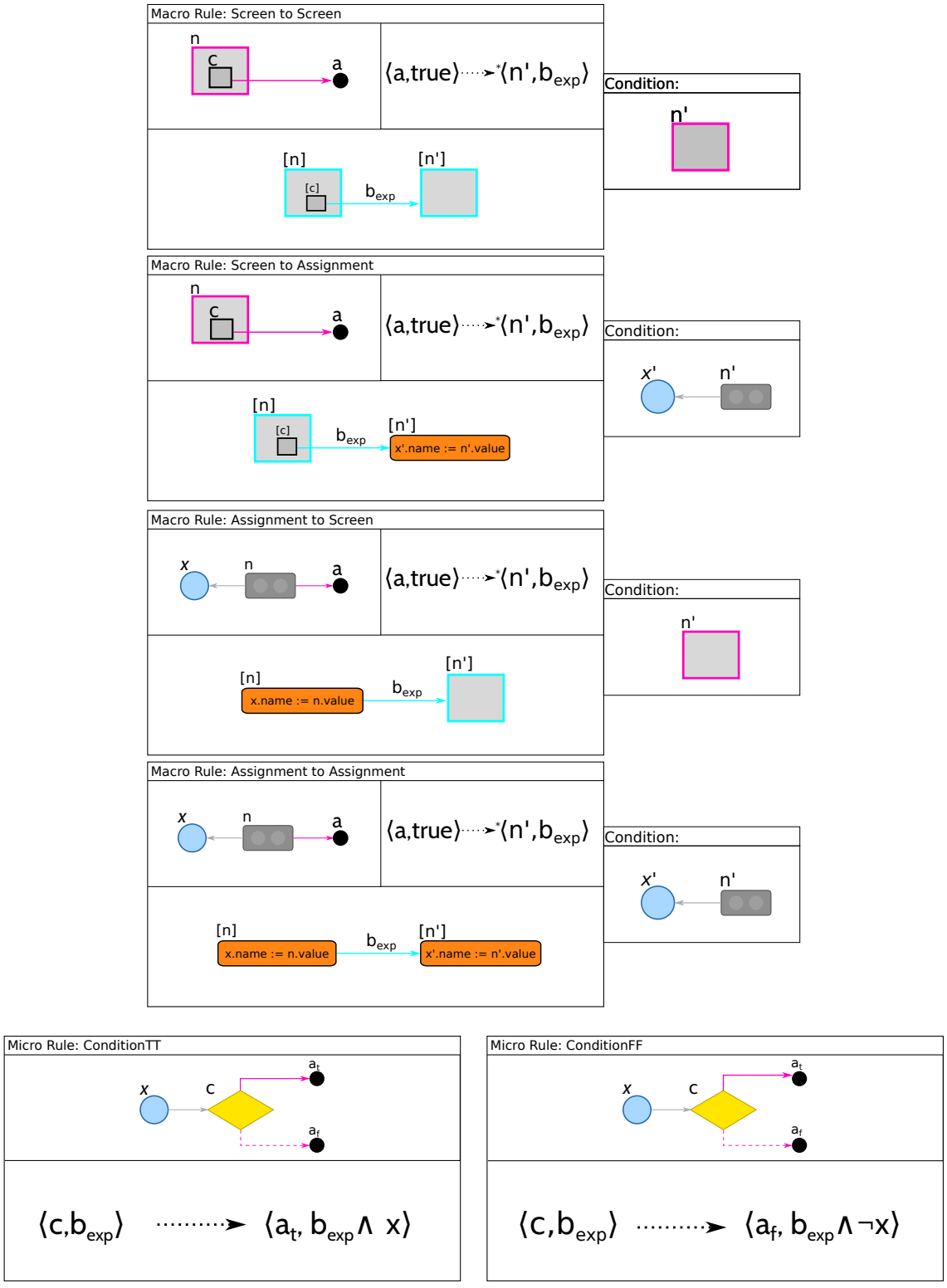


Figure 3.7: Two-level transformation for the migration of WSL models to the modified AC-WSL.

have to be manually repaired. In his thesis, he describes how the two-level transformation language could be used to derive breaking and resolvable rules and the migration for breaking and non-resolvable changes can be conveniently modeled using the two-level transformation language.

Furthermore, the green arrows in Fig. 3.5 represent the transformations to Kripke Transition Systems, which facilitate the application of model checking. After the language evolution represented in Fig. 3.5, these transformation are also invalid. Consequently, these transformations have to be provided to allow for model checking of models expressed in the evolved language. This can be achieved by modeling the transformation anew or transforming the existing transformation rules (cf. Sect 5.2).

3.1.3 Validation and Verification

An important part of language workbenches is the capability to implement validations for the static semantics of the DSVL. For instance, the Xtext [6] framework provides interfaces and annotations that ease the development of *Checks* for a textual DSL. In CINCO, Dominic Wirkner developed the *Model Compare and Merge* framework (MCaM) [Wir15], originally to enable merging of graphical models using *Git* [Spi12]. One of MCaMs capabilities is the annotation of MGL types by a class implementing validations for that type, which are automatically executed in the generated CINCO product.

An alternative to the implementation of validations is the usage of formal verification methods, like *Model Checking* [CGL94]. Usually, the formal model M is represented by a finite state transition system which serves as input for a model checker. Modeling languages containing a notion of control flow already imply a kind of transition system, but their representation is not appropriate for a model-checker. Therefore, transforming the models into a representation that can be interpreted by a model-checker is an obvious solution [LKS18, KLNS20].

3.2 The Need for Domain-Specific Transformation Tools

In [SVL15, Cua12] the authors argue that for different kinds of model transformations, different tools are suited to specify the transformations.

As illustrated in the previous section, model transformations are required in several areas during the development of CINCO and CINCO products. In this section, I explain how we improved CINCO to support CINCO product developers in the definition of model transformations. Our studies and efforts led to the realization, that domain-specific model transformation languages are desirable for different scopes of application of model transformations.

In the following, I distinguish between the ideas to support the development of code generators (Sect. 3.2.1) and model-to-model transformations (Sect. 3.2.2) with a focus on the latter.

3.2.1 Code Generation in Cinco

The first version of the code generator yielding the Graphiti API implementation, was modeled by me during my masters thesis [Kop14] using the *Java Application Building Center* (jABC) [MS09] and *GeneSys* [Jö11] in a process-oriented manner. I.e., the models reflected the generation logic [Jör13, CE00] and the output description was created using special building blocks, which allowed for the definition of code templates. In spite of the advantages of this approach, legacy libraries forced us to rewrite the generators. We decided to use *Xtend* [5], due to its integrated template DSL, the support for lambda expressions and compatibility to Java.

In several projects, we experienced that the code for the management of Eclipse projects was duplicated and slightly changed. This motivated Steve Boßelmann to create a DSL, abstracting the technical details of the Eclipse project creation process, and thereby providing a standard for the development of code generators for Cinco and Cinco products. The DSL was implemented as an *internal DSL* [Fow05] in *Xtend*. With the growing complexity of Cinco products (i.e. the number of involved MGL models and the models created using Cinco products) the time consumed by code generators increased to a point, at which the development process was hindered. The problem resulted from the full generation approach: Since models are the main development artifacts the common development strategy is to modify a model and generate the whole application anew. In Cinco, this occurred during the development of Dime. Michael Lybecait reports in his Ph.D. thesis [Lyb19] about an optimization to reduce the generation time of Cinco products by analyzing updated models and only generate the subset of changed models.

The described features could be provided by a DSL supporting the development of efficient code generators. First steps in this direction were made by the *Projektgruppe NextGen* [BCK+19] and the bachelor thesis of Joel Tagoukeng Dongmo [Don19].

3.2.2 Model-to-Model Transformations

To ease the development of model-to-model transformations, we enhanced Cinco to improve the support for their specification. Figure 1.2 shows an overview of the approaches for model transformations employed in Cinco, categorized in *imperative/declarative* and *general-purpose/domain-specific* approaches.

In the declarative, domain-specific category, I distinguish between *Graph-based Model Transformations* and *Graph-based Domain-Specific Model Transformations*. With the former I relate to languages facilitating the definition of transformations which incorporate the domain-specificity of the underlying DSLs. This approach is realized in, i.a., *AtomPM* [SGV13, SVM+13] for domain-specific visual languages. Weisemöller described in [Wei12, HRW15] the generation of a transformation language for textual DSLs defined by a grammar. With *Graph-based Domain-Specific Model Transformations*, I want to emphasize the purpose of a transformation, which I consider as a domain-specific aspect. Consequently, these transformation languages are specializations of *Graph-based Model Transformations*.

Ideally, developing a model-to-model transformation between DSLs should only presume the familiarity with the involved languages. With each Ecore meta model specification, EMF generates an API to manipulate instances conforming to the meta model. This API is already a specialization towards the involved languages in the domain of Ecore models (cf. Fig. 1.2, `Metamodel-Generated API`). Since CINCO is developed using EMF and the meta models of CINCO products are expressed in Ecore, the EMF-generated APIs can be used to implement model transformations. Unfortunately, the APIs do not consider the domain-specific concepts introduced by CINCO. The definition of node, container, and edge types provide more detailed information towards the structure of meta models expressed in CINCO. This information can be used to prevent the creation of syntactically incorrect models.

Consequently, as a first specialization towards a domain-specific transformation language we generate domain-specific APIs along with each specification of a CINCO product (cf. Fig. 1.2, `DSL-Generated API`). The generator was implemented by Michael Lybecait and me. The generated domain-specific APIs have two mayor advantages:

1. The generated APIs prevent the creation of models which do not conform to the meta model of the DSVL or do not represent a valid graph structure.

2. Hiding the complexity of the EMF-generated APIs and thereby decreasing the lines of code needed to implement specific actions.

These advantages are exemplified by two listings. Listing 3.1 shows an implementation of the rule depicted in Fig. 2.3(a) using the API provided by EMF. Lines 7-10 represent the search for a *Screen* node with the same `backgroundImage` as the considered *Screen* (given as method parameter in Line 2). If such an element does not exist (cf. Line 12), the new elements are created (cf. Lines 14-25). In Lines 27-40 the connections between those elements are established. Afterwards, the new elements' attribute values are assigned (cf. Lines 43-46) and the containment structure is defined (cf. Lines 49-54).

In contrast, Lst. 3.2 shows the creation of the same structure using the CINCO-generated domain-specific API. Since it is generated on the basis of the WebStory.mgl and the knowledge that models are graph-based, it provides, e.g., specific create methods of the form

```
<ContainerType>.new<ContainedType>()
```

exploiting the knowledge about the containable elements (cf. Lines 13-17). Consequently, the new nodes are created directly in the corresponding valid containers. Additionally, the API provides edge create methods of the form

```
<SourceType>.new<EdgeType>(<TargetType>)
```

according to the MGL specification. I.e., only methods representing valid combinations of source, target, and edge types are generated. This domain-specification

reduces the amount of code and the error-proneness in the transformation development. In Lines 28-40 or Lines 49-54 of Lst. 3.1 the developer can accidentally create models that do not conform to the MGL specification, and even invalid graph-structures by creating, e.g., dangling edges.

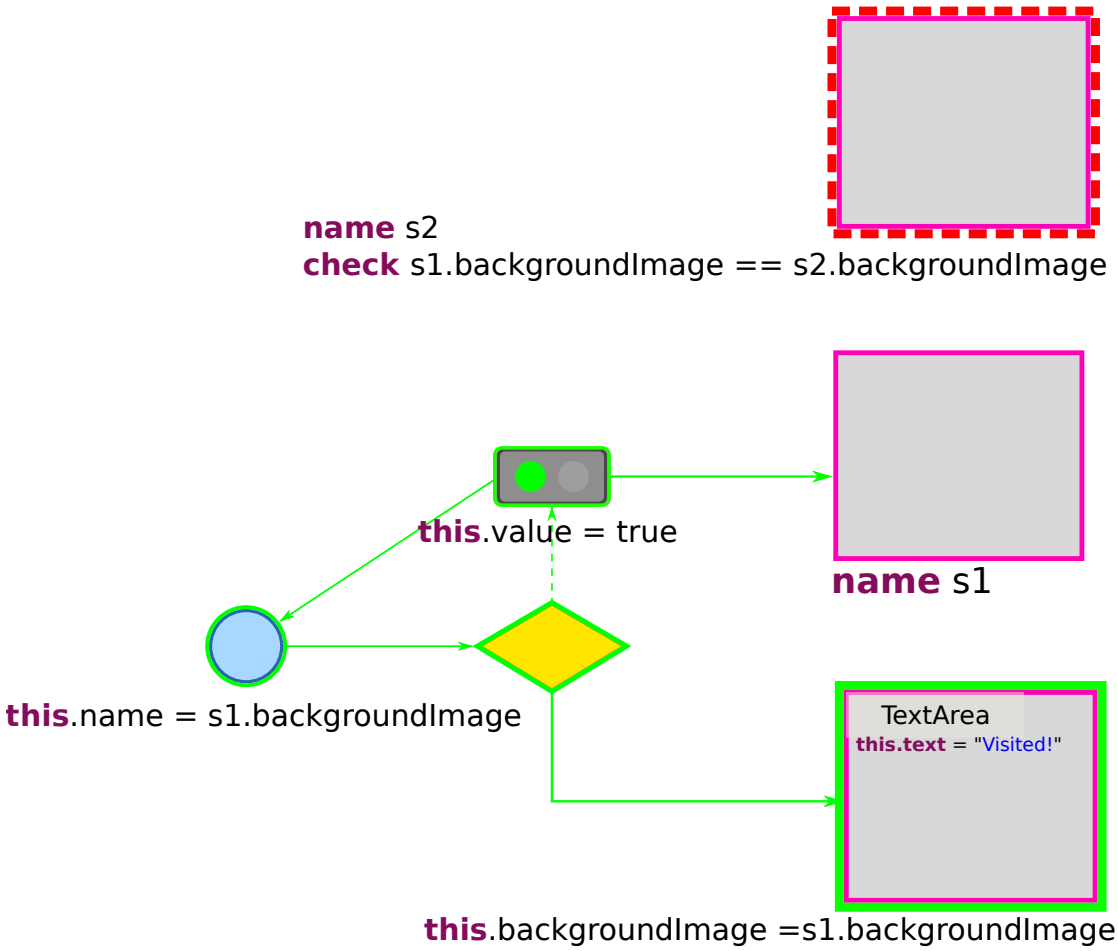
Using the generated domain-specific APIs, a transformation developer is supported in the creation of model transformations, but still requires to have programming skills. The next specialization enables the definition of model transformations for people that do not have the required technical background. In his master thesis [Fel19], Marius Feltmann developed a generator, which, given a CINCO product specification, generates a graph-based transformation language, providing a similar rule definition language as in *Groove* (cf. Sect 2.3.2). Thereby, the concrete syntax defined in the MSL is re-used in the transformation language. The generated transformation language can be used to define endogenous in-place model transformations. Fig 3.8 depicts the transformation rules from the example transformation introduced in Sect. 2.3.2, where Fig. 3.8(a) models the rule implemented in Lst. 3.1 and Lst. 3.2.

```java
@Override
public void execute(Screen sc) {
  WebstoryFactory factory = WebstoryFactory.eINSTANCE;
  WebStory ws = sc.getContainer();

  // Find existing screen node
  Stream<ModelElement> screens = ws.getModelElements().stream().filter(e -> {
    return (e instanceof Screen ) &&
        e != sc && (sc.getBackgroundImage().equals( ((Screen)e).getBackgroundImage
            ()) );
  });

  if (screens.count() == 0) {
    // Create node and container types
    Screen newScreen = factory.createScreen();
    Variable variable = factory.createVariable();
    Condition condition = factory.createCondition();
    ModifyVariable modifyVar = factory.createModifyVariable();
    TextArea text = factory.createTextArea();

    // Create edge types
    DataFlow modify = factory.createDataFlow();
    DataFlow input = factory.createDataFlow();
    Transition transition = factory.createTransition();
    TrueTransition trueTransition = factory.createTrueTransition();
    FalseTransition falseTransition = factory.createFalseTransition();

    // Connect elements
    input.setSourceElement(variable);
    input.setTargetElement(condition);

    modify.setSourceElement(modifyVar);
    modify.setTargetElement(condition);

    falseTransition.setSourceElement(condition);
    falseTransition.setTargetElement(modifyVar);
    trueTransition.setSourceElement(condition);
    trueTransition.setTargetElement(newScreen);

    transition.setSourceElement(modifyVar);
    transition.setTargetElement(sc);

    // Set attribute values
    newScreen.setBackgroundImage(sc.getBackgroundImage());
    variable.setName(sc.getBackgroundImage());
    modifyVar.setValue(true);
    text.setText("Visited");

    // Create containment hierarchy
    newScreen.getModelElements().add(text);
    ws.getModelElements().add(newScreen);
    ws.getModelElements().add(variable);
    ws.getModelElements().add(condition);
    ws.getModelElements().add(modifyVar);
    ws.getModelElements().add(text);
  }
}
```

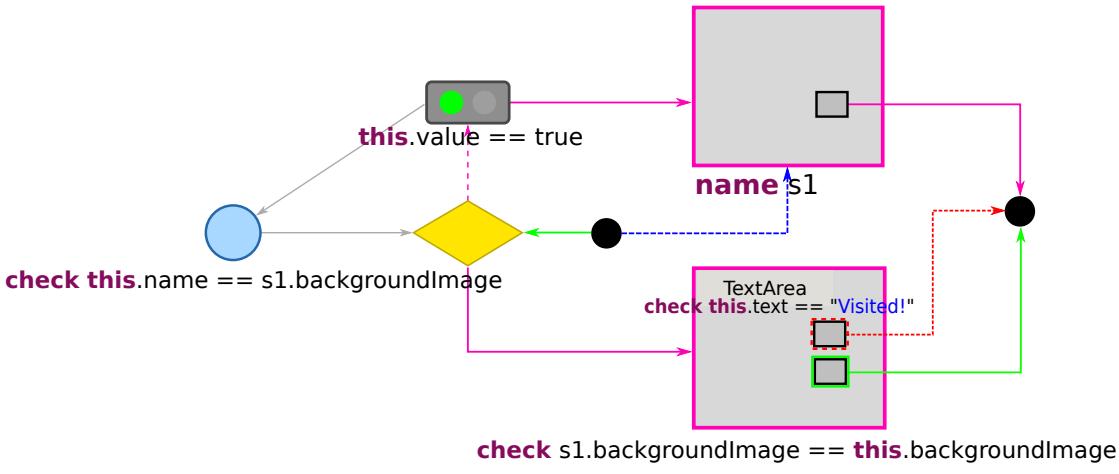
Listing 3.1: Rule from Fig. 2.3(a) implemented using the EMF-generated API.

```
@Override
public void execute(Screen sc) {
  WebStory ws = sc.getContainer();

  // Find existing screen node
  Stream<Screen> screens = ws.getScreens().stream().filter(s -> {
    return (s != sc) && (s.getBackgroundImage().equals( sc.getBackgroundImage()));
  });


  if (screens.count() == 0) {
    // Create node and container types
    Screen newScreen = ws.newScreen(100,100);
    TextArea text = newScreen.newTextArea(0, 0);
    Condition condition = ws.newCondition(0, 0);
    ModifyVariable modifyVariable = ws.newModifyVariable(0, 0);
    Variable variable = ws.newVariable(0, 0);

    // Connect elements
    variable.newDataFlow(condition);
    modifyVariable.newDataFlow(variable);

    modifyVariable.newTransition(newScreen);
    condition.newFalseTransition(modifyVariable);
    condition.newTrueTransition(newScreen);

    // Set attribute values
    newScreen.setBackgroundImage(sc.getBackgroundImage());
    variable.setName(sc.getBackgroundImage());
    modifyVariable.setValue(true);
    text.setText("Visited");

  }
}
```

Listing 3.2: Rule from Fig. 2.3(a) implemented using the CINCO-generated API.



(a)



(b)

Figure 3.8: Transformation rules from Fig. 2.3 modeled using the concrete syntax of the WebStory language.

The syntax is comparable to the example shown in Fig. 2.3, but using the concrete syntax of the WebStory language. As in Groove, the rule types are encoded by the elements' border colors.

In [LKS18, KLNS20], one motivating example is the transformation from the WebStory language to the language of KTSs, which requires the computation of information. This kind of transformation is usually realized using a hybrid approach: The relation between source and target model elements are described using a declarative specification, the computation is implemented using an imperative language [CH06]. This again would demand programming skills of the transformation developer. Since we want to prevent this, we strive for fully model-driven solutions using graph transformations. We described in [KLNS20] how such a transformation could be modeled using *Groove*. Schematically, this solution is shown in Fig. 3.9, where the `Computation` model is added as third part of the transformation.

In general-purpose graph transformation languages, the elements required to model the computation can be easily added to the rule definition. This approach is very powerful, but suffers from the same problems as general-purpose languages compared to DSLs: Due to the lack of specialized computation concepts in general-purpose transformation languages, the modeler has to take care of *how* to specify the transformation (e.g. stepping function, memory allocation, memory update etc.) instead of focusing on *what* should be accomplished by the transformation. Consequently, we propose in [LKS18, KLNS20] specialized transformation languages in the context of DSVLs. Transformations involving a computation represent one of the domains for transformation languages. We consider the computation as a domain-specificity and provide transformation languages for these kind of transformations. This allows for a declarative specification of the computation process by introducing a second rule level which focuses on the computation definition.

Summarizing, model transformations are required to solve different problems in model-driven engineering. One possibility is to develop model transformations in a general-purpose transformation language. In the context of DSVL, we suggest to apply the DFS approach on the domain of model transformations and generate tailored languages, facilitating the modeling of specialized transformation rules. In the next chapter, I describe a pattern for the generation of domain-specific transformation languages in the context of DSVL. The computation part is specified in this language using an SOS-like [Plo81] approach. Individual computation steps are modeled by a transition between states, representing the current configuration of the computation.

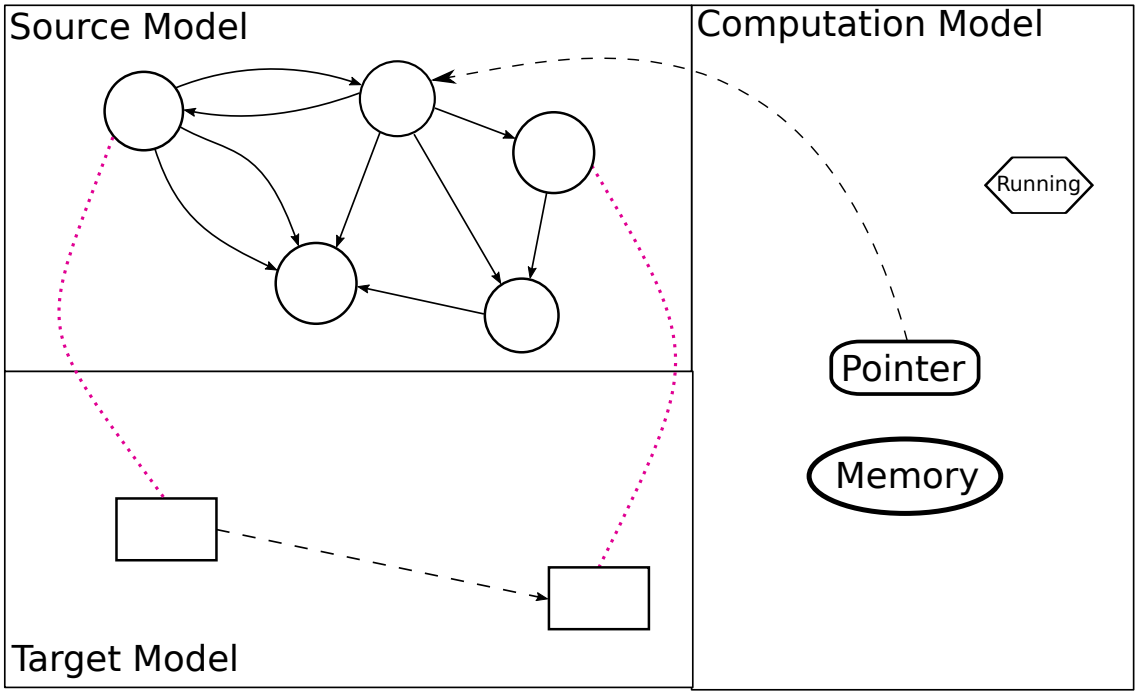


Figure 3.9: Schematic representation of the relevant components for the computation during a graph transformation [KLNS20].

4 Generation of Language-to-Language Transformation Languages

In [KLNS20], we presented applications for model transformation languages utilizing an SOS-like [Plo81] auxiliary rule system to aggregate information. Transformation languages transforming between models of domain-specific visual languages suffer from the same problems as DSVL themselves: The manual realization of modeling tools for the domain-specific transformation language comprises tedious and repetitive activities [Nau17, Lyb19, NLKS17]. For each combination of source and target language a transformation language has to be developed. Thus, the ability to generate the transformation languages is of major importance, since the manual implementation of these languages would exceed their value.

This chapter represents the main contribution of this dissertation by detailing on the generation of those transformation languages. In Sect. 4.1, I will shortly recapitulate the pattern of the two-level transformation language presented in [KLNS20] and afterwards describe in Sect. 4.2 how the transformation language can be generated utilizing a previous configuration. Thereby, I focus on the single parts constituting the transformation process.

4.1 Two-Level Transformation Languages

One of the main ideas presented in [KLNS20] is the explicit separation of the constituents of a model transformation. Since the considered transformations involve the aggregation of information, the transformation language consists of three parts: The source, target, and auxiliary language. Computation steps are modeled separately using specialized rules. This prevents the pollution of the transformation rules describing the relation between elements of source and target language, and results in a two-level transformation language. First-level rules conform to the following

schematic representation.

$$\frac{\boxed{\textit{Source Langauge}\ \text{Pattern}},\ \boxed{\text{Auxiliary}\ (\dashrightarrow^*)}}{\boxed{\textit{Target Language}\ \text{Pattern}}}\ \text{Condition}$$

They facilitate the specification of patterns in the source and the target language, and thereby define the relation between elements of the languages. The optional auxiliary part indicates a computation consisting of an arbitrary number of steps ($\dashrightarrow^*$). The rules specifying individual computation steps are schematically represented as follows.

$$\frac{\boxed{\textit{Source Language}\ \text{Pattern}}}{\boxed{\textit{Auxiliary Language Step}\ \text{Pattern}\ (\dashrightarrow)}}\ \text{Condition}$$

These rules allow for the definition of the relation between source language elements and auxiliary language elements which describe a computation step.

In the next section, I use a Structural Operational Semantics (SOS)-based auxiliary language to describe the generation process for the presented transformation language. Therefore, I will shortly describe the main concepts of Plotkin's SOS [Plo81] and how this approach is used in second-level rules to provide a basic understanding of the transformation language. The SOS is used to specify semantics for programming languages. The semantics of a program written in a programming language is defined by the program's effect on the processed data. The effect is defined for statements of the programming language using rules of the following form.

$$\frac{Premise}{Conclusion}\ \text{Condition}$$

These rules describe how the residual program and the data are derived executing a statement. Applying the specified rules on a program results in a transition system, which represents the program's execution. A state of the transition system, also called *configuration*, consists of

1. the residual program and
2. the current data valuation.

Thus, one transition in the transition system represents the execution of one statement in the program, defining the effect on the data. In the remainder of this thesis, I will refer to the structure managing the data as *store*.

In the two-level transformation language, SOS is utilized in the *Auxiliary Language Step* pattern in second-level rules, which is used to model a transition between two SOS configurations. The *source language pattern* in second-level rules corresponds to the *Premise* in an SOS rule and the condition is used to define constraints over

the store, which manages data in form of primitive variables. Consequently, the source language pattern and the condition define the application condition of the rule.

In contrast to SOS, the input for the transformation language is a graph. To decouple SOS from the actual input model, the "residual program" is not explicitly represented in a configuration, but indicated by a pointer which refers the current model element in the computation. The meaning of second-level rules is the following: If the element referred by the source configuration of the SOS transition satisfies the precondition, i.e., it conforms to the type referred in the pattern, the execution continues at the element in the input model, which is represented in the target configuration of the transition, resulting in the store represented by the target configuration.

4.2 Generation of Two-Level Transformation Languages

Figure 4.1 gives an overview of the generation process for two-level transformation languages. In this section, the generation process is exemplified using the SOS-based auxiliary language.

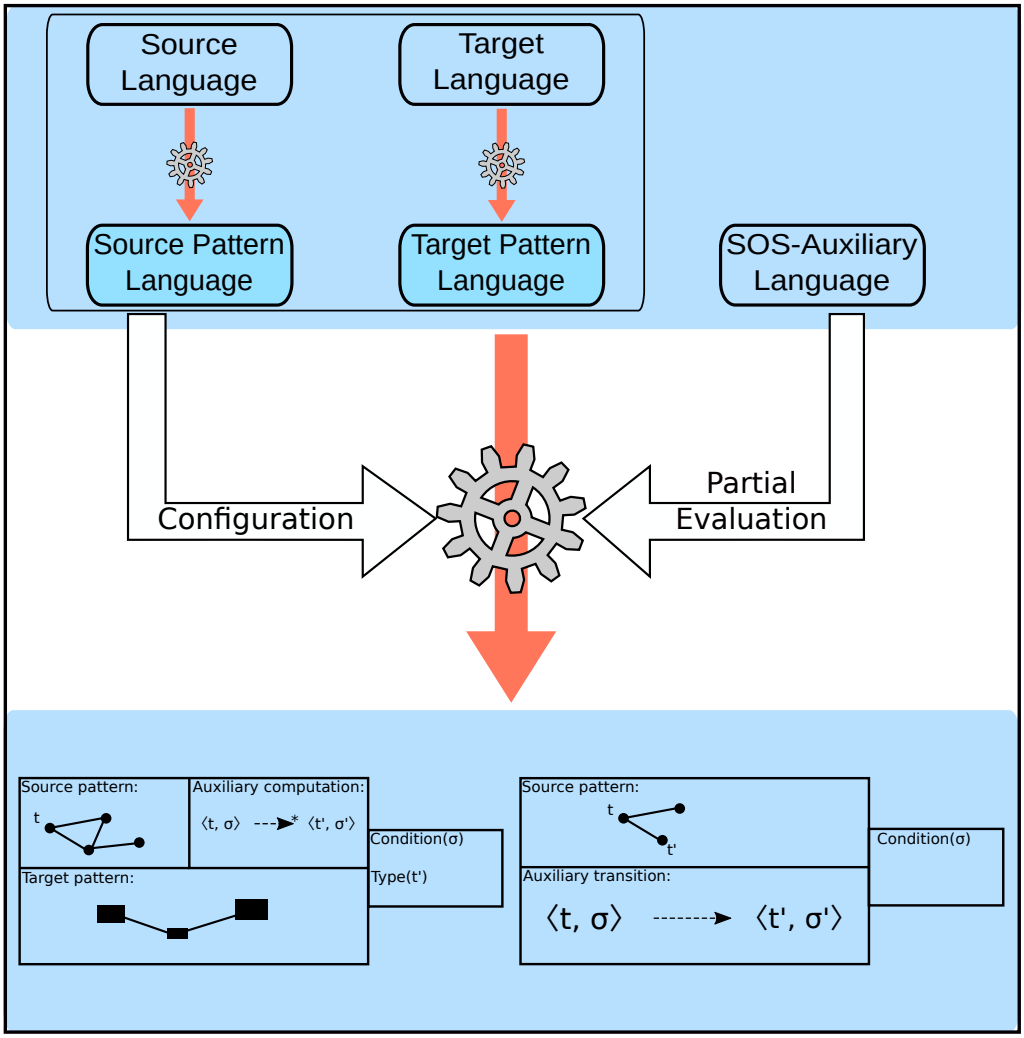


Figure 4.1: Overview of the generation process of the SOS-based transformation language.

The generator is parameterized by the *source*, *target*, and *auxiliary* language and the generation process consists of three parts.

- The generation of source pattern language and target pattern language is a static extension of the corresponding original languages. Based on those pattern languages, source model queries and target model modifications are specified in the corresponding parts of first-level and second-level rules (cf. Sect 4.2.1).

- The major parts of the SOS-based auxiliary language are either static or restricted by the domain of source and target language. This allows for a partial evaluation of the auxiliary language by the generator (cf. Sect. 4.2.3).

- The dynamic parts of the auxiliary language are defined by the generator configuration. It provides an initialization of the store and defines how elements in the target model are identified (cf. Sect. 4.2.4).

4.2.1 Source Language Pattern

One of the key aspects for transformation languages is the possibility to query the source model and create elements in the target model. I will first describe the static extensions of the source language allowing for the definition of patterns to specify model queries. The extensions are oriented at the concepts presented in [KMS+10].

Language Types and Attribute Constraints The pattern language should allow for the definition of rules based on the types defined in the source language. Consequently, it has to include all types and their interrelations defined in the source language. Additionally, the source pattern language has to allow for the definition of conditions over type attributes. Considering the WebStory language as source language, one could desire to find, for instance, *Modify Variable* nodes, whose Boolean attribute *value* is set to `true`.

Inheritances and Abstract Types Type inheritance and abstract types are used in DSLs (and programming languages like Java) to accumulate similar behavior shared among several types in their super-type. This behavior is then propagated using inheritance. Addressing a possibly abstract type is crucial in transformation rules. It prevents the replication of transformation rules entailed by enumerating all non-abstract sub-types of an abstract type. Therefore, abstract types in the source language are made available in a transformation language by changing them to concrete ones in the specification of the transformation tool.

Type Wildcards Sometimes the specific type of an element is not relevant in a rule. The WebStory pattern in the upper left part of the rule displayed in Fig 1.3 exemplifies this situation. It is only relevant that the *Click Area* node modeled in the pattern has a successor connected by a *Transition* edge. Therefore, wildcard types are introduced in the pattern language. In Cinco-based transformation languages three wildcard types for the node, container, and edge meta type are introduced.

Relaxing Static Semantics The relaxation of a DSVL's static semantics is a basic requirement to allow for, e.g., the usage of type wildcards. In the generated DSVL, the wildcard is also represented as a type. Thus, edge types' constraints have to be relaxed to allow wildcard nodes to be an edge's source or target node. Similarly, node types have to be modified to allow for incoming and outgoing edge wildcard types and the containment constraints of container types are relaxed to facilitate the containment of wildcard elements.

Rule Types One characteristic of the transformation language presented in [KLNS20] is the additive transformation system, i.e., the target model is built from scratch, and the source model is only used to retrieve information for the transformation. Thus, we constrain the rule type of an element x defined in the source pattern as follows.

- $x \in LHS \Leftrightarrow x \in RHS$, meaning that x should be matched in the input model. It is neither deleted nor created.
- $x \in NAC$, preventing the matching of model structures in which x occurs.

4.2.2 Target Language Pattern

Resulting from the design decisions of the transformation language (*separation of source language and target language* and *additive transformation system* [KLNS20]) the target pattern languages are not as complex as the source pattern languages. Essentially, they facilitate the modeling of structures that also can be modeled in the target language. Types in the target pattern language represent an *enhanced creator semantics*: If they do not exist in the target model, they are created, and updated otherwise. The usage of wildcards or abstract types in a target pattern rule is pointless, because it is not determinable which concrete type is represented by the wildcard or abstract type. The target language is extended by an expression language allowing for the derivation of attribute values of created elements.

4.2.3 The Auxiliary Rule Language

The SOS-based auxiliary language should facilitate the definition of rules by which the aggregation process is defined. More precisely, the transformation developer has to define a *step* function, which, for each type of the source language, defines how to retrieve its successor. Furthermore, the function has to describe the step's effect on the store. The definition of the step function depends on the current position in the input graph and the current store. Even without the configuration (cf. Fig. 4.1), the major parts of the auxiliary language can be generated.

To explain the ability to generate the language, I will provide the static parts of the abstract syntax of the SOS-based auxiliary language, and subsequently describe the parts of the concrete syntax which can be generated. Thereby, I will focus on the requirements and the knowledge implied by the domain-specificity of the transformation language.

Abstract Syntax Without the prior configuration of the store, it is not known what kind of data will be processed during the execution of the transformation. Although the data types can be complex (composed of primitive data types), eventually it has to be defined how to evaluate and modify complex data types. This usually is realized by a mapping from the complex type to one of its primitive components. As a consequence, we restrict the types of variables which can be defined in

the store to primitive ones, allowing us to generate the evaluation and modification function, independent of the actual store.

Definition 1 *Let $\mathcal{V}ar$ be a set of named, primitive variables and $\mathcal{V}alue$ a set of possible* values.[1] *The* store *σ assigns each variable a corresponding value.*

$$\sigma : \mathcal{V}ar \rightarrow \mathcal{V}alue$$

The set of all possible variable valuations is defined by

$$\Sigma = \{\sigma \mid \sigma : \mathcal{V}ar \rightarrow \mathcal{V}alue\}$$

To modify the values of variables we define a substitution function on the store.

Definition 2 *Let $X, Y \in \mathcal{V}ar$, $v \in \mathcal{V}alue$, and $\sigma \in \Sigma$. The substitution function on σ is defined by*

$$\sigma\{X \rightarrow v\}(Y) = \begin{cases} v & , \textit{ if } X = Y \\ \sigma(Y) & , \textit{ otherwise} \end{cases}$$

With this notion of the store, we can now define the domain of the step function, that is the main artifact to be modeled (defined) by the transformation developer.

Definition 3 *Let $\mathcal{T}_{src}$ be the set of types in the source language and Σ as defined in Def. 1. The domain of the step function is defined as follows.*

$$step : (\mathcal{T}_{src} \times \Sigma) \rightarrow (\mathcal{T}_{src} \times \Sigma)$$

Please note, as the source language is a parameter of the generator (it is not the result of the configuration) the domain of the *step* function can be restricted to the types specified in the source language. The defined domain of the step function represents the abstract syntax of the SOS-based auxiliary language. To model an executable transformation, the function definition has to be provided by the transformation developer. Therefore, we generate a concrete syntax for the SOS-based auxiliary language to ease the definition of the introduced function.

Concrete Syntax Given the abstract syntax of a two-level transformation language, the goal is to provide a modeling language, which eases the specification of rules defining the step function. In the scope of visual languages, the concrete syntax of the two-level transformation language is also realized by a DSVL. The resulting concrete syntax is shown at the bottom of Figure 4.1, where a first-level rule is depicted on the left-hand side, a second-level rule on the right-hand side. As mentioned in Sect. 4.2.1, queries and element creations are specified by patterns. Consequently, the concrete syntax of first-level and second-level rules should provide areas in which the corresponding patterns can be specified

[1] We consider *Boolean, character, integer, string* etc. as primitive types.

Therefore, the rules are realized as containers, separated in different areas. First-level rules consist of four parts, providing areas for the definition of `Source pattern`, `Target pattern`, `Auxiliary aggregation`, and a `Condition`. Second-level rules consist of three parts facilitating the specification of `Source pattern`, `Auxiliary step pattern`, and a `Condition`. The rule areas are typed by the corresponding languages, forcing a strict separation between the constituents.

The *step* function should be defined by the specification of a transition between two configurations, i.e., given the type $t \in \mathcal{T}_{src}$ of the currently processed node in the source model and a variable valuation $\sigma \in \Sigma$, the modeler has to specify the successor $t' \in \mathcal{T}_{src}$ and the possibly modified variable valuation $\sigma' \in \Sigma$:

$$\langle t, \sigma \rangle \dashrightarrow \langle t', \sigma' \rangle$$

This notation represents the concrete syntax for the transition system. It consists of an identifier (t, t') and a representation of the store (σ, σ'). In first-level rules, an aggregation process can be indicated by the definition of its start state and end state, connected by a transition which is marked by the Kleene Star (cf. Fig. 4.1, `Auxiliary computation`). The individual steps are defined in second-level rules in the `Auxiliary transition` area. In both cases, the identifier in a transition system's state ($\langle t, \sigma \rangle$), has to be related to a type of the source language. Therefore, the source pattern language is additionally extended to allow for the definition of labels in the pattern, and the relation is defined by string comparison.

Of course, determining the next element during the aggregation can depend on the current store. Therefore, conditions can be defined in the `Condition` parts of first-level and second-level rules. Resulting from the restriction of the store to primitive variables, a simple expression language suffices for this purpose. This language is independent of the configuration (and the generator parameters) and is generated as static part of the concrete syntax. Additional to the specification of Boolean conditions, the `Condition` part in first-level rules can be used to define patterns in the source language to, e.g., specify a termination condition for the computation. For instance, the node's type identified in the end state of the indicated aggregation can determine the termination of a computation.

4.2.4 Generator Configuration

To generate the transformation language, the generator has to be configured by an initialization of the store σ, and an identification function for elements of the target language.

Store Initialization The store variables used during an aggregation process depend on the desired transformations. In one case, a transformation may require one variable per node of a specific type from the source domain. In another case, one local variable managing some information may be sufficient for the aggregation. Consequently, the generator has to be configured with an initialization of the store,

i.e., an explicit specification of the set of variables $\mathcal{V}ar$ and their initial values. Furthermore, on the one hand the domain of the store is restricted to variables of primitive data types. On the other hand, DSVL usually allow for the usage of complex types. Therefore, the initialization has to map types and their attributes to primitive variables. The domain of the store initialization function *init* is defined as follows.

Definition 4 *Let $\mathcal{T}_{src}$ be the set of types of the source language and Σ the set all variable valuations. The domain of the store initialization function* init *is given by*

$$init : \mathcal{T}_{src} \to \Sigma$$

.

To define the *init* function, we provide a text-based language which allows for the definition of primitive variables, as well as a mapping from types (and their attributes) to primitive variables. Listing 4.1 shows the EBNF of the language to configure the store. The grammar facilitates to create single primitive typed variables (Line 3), as well as a variable for each occurrence of an element typed by $t \in \mathcal{T}_{src}$ (Lines 5-7). In the latter case, variable names can be generated or specified by referencing an attribute of the corresponding type (Line 6). Initial variable values can be statically set or derived from the type's attribute (Line 7).

Target Identification The *enhanced creator semantics* of a first-level rule application is to create elements defined in the target pattern, if they do not exists and update them otherwise. The easiest way to identify the elements is to consider their type and attribute valuation, i.e., two elements are equal, if they have the same type and all their attribute values are equal.

In some cases this kind of identification is too detailed. In model-checking, this leads to the *state explosion* problem [CKNZ12]. This problem is tackled by, e.g., abstraction: Instead of verifying properties on the actual model, an abstraction of the model is considered. The abstraction is defined through a mapping from the actual state space to a subset of states by abstracting irrelevant information.

In a similar way, a mapping is specified to define what information a target element should be identified by. Since, target elements may be related by source language

```
1 <AbstractVariable> :== <PrimitiveVariable> | <Variable>
2
3 <PrimitiveVariable> :== <VariableType> <Identifier> ('=' <String>)?
4
5 <Variable> :== 'forall' <TypeReference> 'create' <VariableType>
6  ('generateName'|<TypeReferenceAttribute>)
7  ("=" (<DerivedValue> | String))?
8
9 <VariableType> :== 'int' | 'boolean' | 'string'
```

Listing 4.1: EBNF for the store initialization language.

types and information aggregated by second-level rules, the mapping function is defined as follows.

Definition 5 *Let $\mathcal{T}_{src}$ ($\mathcal{T}_{trg}$) be the set of types of the source (target) language, and let $\mathcal{V}ar$ be a set of primitive variables. The domain of the mapping function* map *is defined by*

$$map : (\mathcal{P}(\mathcal{T}_{src}) \times \mathcal{P}(\mathcal{V}ar)) \rightarrow \mathcal{T}_{trg}$$

.

Since the *map* function's domain involves the set of variables $\mathcal{V}ar$, the store configuration has to be specified previous to the mapping.

4.2.5 The WebStory to Kripke Transition System Transformation Language

In this section, I will exemplify the generation process by means of the WebStory to KTS transformation language incorporating the SOS-based auxiliary language. Thus, the WebStory specification serves as source language parameter, the KTS language as target language parameter. In the following, I will emphasis on the configuration, since its definition is the main task of the developer.

The purpose of the transformation is to verify properties of WebStory models by model-checking [KLNS20]. We want to define CTL properties over the *Variable* nodes used in a WebStory model. *Variable* nodes are manipulated by *Modify Variable* nodes. With this information, we can define the store initialization as follows.

$$\begin{aligned}
&init: && \mathcal{T}_{WebStory} \rightarrow \Sigma && \text{, with} && \\
&init: && v \mapsto (bool \ \mapsto \texttt{false}) && \text{,} \forall v \in \textit{Variable} && (4.1)\\
&init: && m \mapsto (bool \ \mapsto m.value) && \text{,} \forall m \in \textit{Modify Variable} && (4.2)
\end{aligned}$$

Definition (4.1) specifies that for each element of type *Variable* a Boolean variable should be created and initialized with the value `false`. Similar, Def. 4.2 creates a Boolean variable for each element of type *Modify Variable* but initializes it with the value of the *Modify Variable*'s attribute *value*. Applying this definition on the WebStory model depicted in Fig. 2.2 results in the following store:

$$\begin{aligned}
\sigma[\quad \boldsymbol{v} \quad &\mapsto \texttt{false},\\
\boldsymbol{v1} \quad &\mapsto \texttt{false},\\
\boldsymbol{m} \quad &\mapsto \texttt{true},\\
\boldsymbol{m1} \quad &\mapsto \texttt{true}]
\end{aligned}$$

Now, we can specify the identification for target model elements.

$$\begin{aligned}
&map: && (\mathcal{P}(\mathcal{T}_{WebStory}) \times \mathcal{P}(init(\mathcal{T}_{WebStory}))) && \rightarrow \mathcal{T}_{KTS}\\
&map: && (\{Screen\}, init(Variable)) && \mapsto S_{KTS}
\end{aligned}$$

We use *init*(X) for a set X to describe the image of the *init* function. Consequently, we identify the state in a KTS by the *Screen* type and the variables in the store which originate from *Variable* nodes. The generated SOS-based transformation language can be used to model the rules shown in Figure 1.3 (*first-level* rule) and Figure 1.4 (*second-level* rules).

5 Future Work

5.1 Application

In Sect. 3.1, I presented several projects, bachelor and master theses in which model transformations were defined using the Cinco-generated API. It would be interesting to model these transformations using the presented approaches in this thesis and compare the resulting transformation systems. For instance, the semantics definition of the CMMN language by means of the transformation to Dime models would be an interesting application, since the transformation has to consider three DSVLs which constitute Dime.

Furthermore, it would be interesting to investigate the usability of the model transformations in the context of code generators, since their development is usually based on textual template-based languages.

5.2 Higher-Order Transformations

During the development of DSVLs, several model transformations are created, e.g., for the purposes described in Sect. 3. These transformations can involve different kinds of DSVLs. Using graph-based, domain-specific transformation languages to model these transformations entails many advantages. However, since the transformation languages are designed to conform to the languages involved in the transformation, changing one of these languages can result in the invalidation of models and, additionally, all transformations involving the modified language. Considering the transformation from WebStory to KTS, the modification of the WebStory language would invalidate the modeled transformation. To enable the verification of models conforming to the modified WebStory language, one has to model the transformation anew, or migrate the existing transformation rules.

Higher-Order Transformations (HOTs) are "model transformations that analyze, produce or manipulate other model transformations" [TCJ10]. These kind of transformations could be used to, e.g., transform the model transformations from the

WebStory language to KTS into transformations of a modified WebStory language to KTS.

5.3 Auxiliary Language

I believe that the two-level transformation languages represent powerful tools to specify computational model transformations. It would be interesting to investigate if other kinds of auxiliary languages exist which support the definition of different aspects in model transformation languages. For instance, the transformation from truth tables to binary decision diagrams, as proposed in this year's transformation tool contest [8], represents an interesting challenge. The transformation requires the identification of columns in the truth table satisfying special conditions, before transforming them to elements in the binary decision diagram.

6 Conclusion

In this thesis, I presented the tasks and problems during the development of domain-specific visual languages, alongside three specific activities. To handle those activities, I described how the domain-specific, full-generation, and service-orientation (DFS) approach can be applied to the domain of model transformation languages. The resulting transformation languages provide means to define model transformations in a declarative way. Furthermore, I explained model transformations, in which the information required for the transformation is not statically or locally represented in the source languages. These transformations need to compute the required information. I sketched how such transformations can be specified using existing graph transformation approaches. To reduce the complexity introduced by the computation, I additionally considered the computation during a transformation as a domain-specific aspect and applied the DFS approach on the domain of "computational model transformations". I introduced a language pattern for transformation languages which impose restrictions on the traditional way to define graph-rewrite rules, separate the concerns of these transformations, and provide domain-specific capabilities to define the individual concerns of the transformations. I employed Plotkin's Structural Operation Semantics (SOS) to create a domain-specific language for the specification of computations during a transformation. This way, I facilitate the definition of computations in a declarative, elegant way that allows business experts without programming skills to specify the computation.

Resulting from the amount of possible combinations of source and target languages, and the purpose of the actual transformation, I showed how the transformation languages can be generated by analyzing the interrelations between the involved languages, generalizing them, and providing means to specify the interrelations which can not be generated. I used the two-level WebStory to KTS transformation language to exemplify the required specification.

I believe, that these types of transformation languages are a powerful tool for the considered transformation. It would be interesting to research if other auxiliary languages exist, which can be utilized to specify different aspects of a transformation languages.

References

[AKN+06] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The design of a language for model transformations. *Software & Systems Modeling*, 5(3):261–288, 2006.

[AKS03] Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph Transformations on Domain-Specific Models. *Journal on Software and Systems Modeling*, 37:1–43, 2003.

[BCK+19] Daniel Busch, Beka Chkopoia, Sascha Kiesow, Niclas Müller, Johannes Mundorf, Alnis Murtovi, Benedikt Oesing, Sören Ohlsen, Andreas Sitta, Robin Weisbauer, and Jonas Wielage. neXtGen - Model-based Code Generation, 2019.

[BCT05] Alan W. Brown, Jim Conallen, and Dave Tropeano. *Introduction: Models, Modeling, and Model-Driven Architecture (MDA)*, pages 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[BDG+15] Agata Berg, Cedric Perez Donfack, Julian Gaedecke, Eike Ogkler, Steffen Plate, Katharina Schamber, David Schmidt, Yasin Sönmez, Florian Treinat, Jan Weckwerth, Patrick Wolf, and Philip Zweihoff. PG 582 - Industrial Programming by Example. Technical report, TU Dortmund, 2015.

[BFK+16] Steve Boßelmann, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff, and Bernhard Steffen. DIME: A Programming-Less Modeling Environment for Web Applications. In *Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part II (ISoLA 2016)*, volume 9953 of *LNCS*, pages 809–832. Springer, 2016.

[Bé05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45:621–645, 2006.

[CKL+17] Mounir Chadli, Jin H. Kim, Kim G. Larsen, Axel Legay, Stefan Naujokat, Bernhard Steffen, and Louis-Marie Traonouez. High-level frameworks for the specification and verification of scheduling problems. *Software Tools for Technology Transfer*, 20(4):397–422, 2017.

[CKNZ12] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[CREP08] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany*, pages 222–231, 2008.

[Cua12] Jesús Sánchez Cuadrado. Towards a family of model transformation languages. In *International Conference on Theory and Practice of Model Transformations*, pages 176–191. Springer, 2012.

[Don19] Joal Tagoukeng Dongmo. A domain-specific management framework for environment independent, partial code generation. Bachelor thesis, Technische Universität Dortmund, Fakultät für Informatik, Lehrstuhl für Programmiersysteme, 2019.

[EEGH15] Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation - General Framework and Applications*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2015.

[Fav05] J-M Favre. Languages evolve too! changing the software time scale. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 33–42. IEEE, 2005.

[Fel19] Marius Feltmann. Generierung von domänenspezifischen Modell-zu-Modell-Transformationsumgebungen in Cinco. Master thesis, TU Dortmund University, 2019.

[Fow05] Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.

[FP11] Martin Fowler and Rebecca Parsons. *Domain-specific languages*. Addison-Wesley / ACM Press, 2011.

[Fro13] Markus Frohme. Agile Domänenmodellierung für prozessgesteuerte Webanwendungen. Bachelor thesis, TU Dortmund, 2013.

[Fuh18] Annika Fuhge. Graphische Modellierung von Cinco Produktspezifikationen. BSc thesis, TU Dortmund, 2018.

[GKP07] Boris Gruschko, Dimitrios Kolovos, and Richard Paige. Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution*, page 3. IEEE, 2007.

[Gol19] Phillip Goldap. Multi-Level Transformation Framework for the Cinco Framework. Master thesis, TU Dortmund University, 2019.

[HBJ09] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Jürgens. COPE - automating coupled evolution of metamodels and models. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 52–76, 2009.

[HRW11] John Hutchinson, Mark Rouncefield, and John Whittle. Model-Driven Engineering Practices in Industry. In *Proc. of the 33rd Int. Conf. on Software Engineering (ICSE '11)*, pages 633–642. ACM, 2011.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 136–145. IEEE, 2015.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[Jör13] Sven Jörges. *Construction and evolution of code generators: A model-driven and service-oriented approach*, volume 7747. Springer, 2013.

[Jö11] Sven Jörges. *Genesys: A Model-Driven and Service-Oriented Approach to the Construction and Evolution of Code Generators*. PhD thesis, Technische Universität Dortmund, 2011.

[KBRC+18] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Daniel Varro. Survey and classification of model transformation tools. *Software and Systems Modeling*, 03 2018.

[KLNS20] Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, and Bernhard Steffen. Towards Language-to-Language Transformation. *International Journal on Software Tools for Technology Transfer (STTT)*, 2020. Accepted.

[KMS+10] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In *Proceedings of the 2009 International Conference on Models in Software Engineering*, MODELS'09, pages 240–255, Berlin, Heidelberg, 2010. Springer-Verlag.

[Kop14] Dawid Kopetzki. Model-based generation of graphical editors on the basis of abstract meta-model specifications. Master thesis, TU Dortmund, June 2014.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society Press, Hoboken, NJ, USA, 2008.

[Kue18] Dennis Kuehn. Model-to-model transformation in meta-modeled cinco domains. *Electronic Communications of the EASST*, 75, 2018.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[LKS18] Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. Design for 'X' through Model Transformation. In *Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018)*, volume 11244 of *LNCS*, pages 381–398. Springer, 2018.

[LKZ+18] Michael Lybecait, Dawid Kopetzki, Philip Zweihoff, Annika Fuhge, Stefan Naujokat, and Bernhard Steffen. A Tutorial Introduction to Graphical Modeling and Metamodeling with Cinco. In *Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018)*, volume 11244 of *LNCS*, pages 519–538. Springer, 2018.

[Lyb19] Michael Lybecait. *Meta-Model Based Generation of Domain-Specific Modeling Tools.* Dissertation, TU Dortmund University, Dortmund, Germany, 2019.

[Men13] Tom Mens. Model transformation: A survey of the state of the art. *Model-Driven Engineering for Distributed Real-Time Systems: MARTE Modeling, Model Transformations and their Usages*, pages 1–19, 03 2013.

[MS09] Tiziana Margaria and Bernhard Steffen. Business Process Modelling in the jABC: The One-Thing-Approach. In Jorge Cardoso and Wil van der Aalst, editors, *Handbook of Research on Business Process Modeling*. IGI Global, 2009.

[MSS99] Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-Checking - A Tutorial Introduction. In *Proceedings of the 6th International Symposium on Static Analysis (SAS '99)*, pages 330–354, 1999.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[MVG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. volume 152, 03 2006.

[MWD+05] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *8th International Workshop on Principles of Software Evolution (IWPSE 2005), 5-7 September 2005, Lisbon, Portugal*, pages 13–22, 2005.

[Nau17] Stefan Naujokat. *Heavy Meta. Model-Driven Domain-Specific Generation of Generative Domain-Specific Modeling Tools*. Dissertation, TU Dortmund, Dortmund, Germany, August 2017.

[NFSM14] Johannes Neubauer, Markus Frohme, Bernhard Steffen, and Tiziana Margaria. Prototype-Driven Development of Web Applications with DyWA. In *Proc. of the 6th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I (ISoLA 2014)*, number 8802 in LNCS, pages 56–72. Springer, 2014.

[NLKS17] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools. *Software Tools for Technology Transfer*, 20(3):327–354, 2017.

[NTI+14] Stefan Naujokat, Louis-Marie Traonouez, Malte Isberner, Bernhard Steffen, and Axel Legay. Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems. In *Proc. of the 6th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I (ISoLA 2014)*, volume 8802 of *LNCS*, pages 463–480. Springer, 2014.

[Plo81] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical report, University of Aarhus, 1981. DAIMI FN-19.

[Poo01] John D Poole. Model-driven architecture: Vision, standards and emerging technologies. In *Workshop on Metamodeling and Adaptive Object Models, ECOOP*, volume 50. Citeseer, 2001.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, pages 479–485, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[SBPM08] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, Boston, MA, USA, 2008.

[Sch06] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[Sch19] Till Schallau. Co-Evolution of Metamodels and Models in Cinco. Master thesis, Technische Universität Dortmund, Fakultät für Informatik, Lehrstuhl für Programmiersysteme, 2019.

[SGNM19] Bernhard Steffen, Frederik Gossen, Stefan Naujokat, and Tiziana Margaria. Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages. In Bernhard Steffen and Gerhard Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, volume 10000 of *LNCS*. Springer, 2019.

[SGV13] Eugene Syriani, Jeff Gray, and Hans Vangheluwe. *Modeling a Model Transformation Language*, pages 211–237. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Spi12] Diomidis Spinellis. Git. *IEEE Software*, 29(3):100–101, May 2012.

[SV10] Eugene Syriani and Hans Vangheluwe. De-/re-constructing model transformation languages. *Electronic Communications of the EASST*, 29, 2010.

[SVL15] Eugene Syriani, Hans Vangheluwe, and Brian LaShomb. T-core: a framework for custom-built model transformation engines. *Software & Systems Modeling*, 14(3):1215–1243, Jul 2015.

[SVM+13] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student*

Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 29 - October 4, 2013., pages 21–25, 2013.

[TCJ10] Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Improving higher-order transformations support in atl. In Laurence Tratt and Martin Gogolla, editors, *Theory and Practice of Model Transformations*, pages 215–229, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Wec16] Jan Weckwerth. Cinco Evaluation: CMMN-Modellierung und -Ausführung in der Praxis. Master's thesis, TU Dortmund, 2016.

[Wei12] I. Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software-Engineering. Shaker, 2012.

[Wir15] Dominic Wirkner. Merge-Strategien für Graphmodelle am Beispiel von jABC und Git. Diploma thesis, TU Dortmund, February 2015.

[WMN16] Nils Wortmann, Malte Michel, and Stefan Naujokat. A Fully Model-Based Approach to Software Development for Industrial Centrifuges. In *Proc. of the 7th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part II (ISoLA 2016)*, volume 9953 of *LNCS*, pages 774–783. Springer, 2016.

[ZNS19] Philip Zweihoff, Stefan Naujokat, and Bernhard Steffen. Pyro: Generating Domain-Specific Collaborative Online Modeling Environments. In *Proc. of the 22nd Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2019)*, 2019.

[Zwe15] Philip Zweihoff. Cinco Products for the Web. Master thesis, TU Dortmund, November 2015.

Online References

[1] The eclipse foundation. https://www.eclipse.org/. [Online; last accessed 08-January-2019].

[2] GEF (Graphical Editing Framework). http://www.eclipse.org/gef/. [Online; last accessed 08-February-2019].

[3] Graphical Editing Framework - Draw2d. http://www.eclipse.org/gef/draw2d/index.php. [Online; last accessed 08-February-2019].

[4] Graphiti - a Graphical Tooling Infrastructure. http://www.eclipse.org/graphiti/. [Online; last accessed 13-February-2019].

[5] Xtend - Modernized Java. http://xtend-lang.org. [Online; last accessed 08-February-2019].

[6] Xtext - Language Engineering Made Easy! http://www.eclipse.org/Xtext/. [Online; last accessed 13-February-2019].

[7] Case management and modeling notation. https://www.omg.org/cmmn/, 2019. [Online; last accessed 11-August-2019].

[8] Transformation Tool Contest. https://www.transformation-tool-contest.eu/solutions_tt2bdd.html, 2019. [Online; last accessed 28-August-2019].

[9] client IO. Joint API. http://www.jointjs.com/api. [Online; last accessed 13-February-2019].

[10] Object Management Group (OMG). Object Management Group. www.omg.org. [Online;last accessed 20-May-2014].

[11] Object Management Group (OMG). OMG Meta Object Facility (MOF) Core Specification Version 2.4.1. http://www.omg.org/spec/MOF/2.4.1/PDF. [Online; last accessed 23-April-2014].

[12] Nico Rehwaldt. Element Templates in the Camunda Modeler | Camunda Team Blog. https://blog.camunda.org/post/2016/05/camunda-modeler-element-templates/. [Online; last accessed 23-November-2016].