Meta-Model Based Generation of Domain-Specific Modeling Tools

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Michael Lybecait

Dortmund

2019

Tag der mündlichen Prüfung:
12. März 2019

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

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

Acknowledgements

It is always important to acknowledge the people who you owe your achievements to. First and foremost I would like to thank Bernhard Steffen for being my supervisor of this dissertation. His top-down approach to problems, irrelevant if interrelated to my dissertation or not, always helped when I was running the risk of getting caught up into minor details. I also want to thank him for helping to make our work at his chair at TU Dortmund University as pleasant as possible.

Furthermore I want to thank Sven Jörges for being the second referee of my dissertation. I would also like to thank Heinrich Müller and Günter Rudolph for being part of my doctoral committee.

Without Stefan Naujokat I certainly would never had the idea to write my dissertation in the first place. I would like to thank him for being my supervisor of my diploma-thesis and all the fruitful discussions that spawned many of Cinco's features. I would also like to thank Dawid Kopetzki for being my office mate and for the continuing efforts he puts into the Cinco project. Speaking of Cinco, I would like to thank everyone who has contributed to the Cinco project over the years being it through final exams, as part of their daily work or otherwise, too numerous to name them all.

Last but not least I would like to thank my family, my parents Brigitte and Wolfgang and my brother Daniel, for lending me all the needed support through the years, and for comforting my nervousness in the final days of writing this dissertation.

Abstract

Today software development often depends on the communication between different shareholders with various professional backgrounds. Domain specific languages (DSL) aim to close the semantic gap between these shareholders by providing a common method for communication. When using meta-tooling suites or language workbenches it is quiet easy to create DSLs for small scenarios or even for single use. But with the more frequent use of DSLs the need for domain-specific tooling has also risen.

This dissertation deals with the challenges of creating domain-specific modeling tools using high-level specification languages via code generation. It focuses on three important elements of domain-specific tool generation such as: specification languages, the tool generation process and the generation of domain-specific APIs, for amplifying the development of plug-ins for the generated tool, which are the main contributions of this dissertation.

The first main contribution focuses on the formalization of the specification languages. It is illustrated by detailing the three specification languages of the Cinco meta-tooling suite. The second main contribution introduces the product generation process, which is used to create domain-specific modeling tools from the high-level domain specific languages, defined in the first contribution. The approach is illustrated by the Cinco product generation process (*CPGP*), which defines the necessary steps to produce a standalone modeling tool in the Cinco meta-tooling suite. The third main contribution of this dissertation is the generation of a domain-specific API based on the same high-level descriptions used for the product generation. It uses information present at generation time to create specific operations that are useful for transformations on graph-models (such as typed successor/predecessor or containment relationships). Therefore the API generation of any product is generated during the execution of the *CPGP*. The API makes it easy to develop any extensions for the Cinco product due to its domain-specific nature and the ability to resemble user actions in the generated editors.

Contents

List of Figures

Listings

Chapter 1

Introduction

The CINCO project was created out of the need for graphical modeling tools. It follows the DFS (Domain-Specificity, Full-Code-Generation, Service-Oriented) approach detailed by Stefan Naujokat [Nau17]. Building on the traditions of process modeling tools such as the *jABC* [Nag09, Neu14] that have been around since the early nineties, the creation of more domain-specific tools using *graphical domain-specific languages* (GDSL) is the core idea of the project.

Implementations of such tools tend to lead to recurring implementation of boiler-plate code. This code, while serving comparable purposes in various tools, is often different enough that no feasible level of code reuse can be found. The inability of reusing code leads to a form of copy-and-paste development which is tedious and error-prone. These considerations motivate the idea that an automatic generation of modeling tools should be possible. To achieve this full generation, a family of high-level languages that are simple, yet powerful enough to describe modeling tools needs to be designed. These languages need to be precise enough to describe the abstract and concrete syntaxes of the GDSLs and also the form and function of the complete tool. From a description expressed in these languages the tool can be generated using a full code generation [KT08] approach.

Full code generation has several beneficial properties. First, it treats models as first-class citizens meaning that changes in the models will be directly propagated into the generated parts of an application. Second, full-code-generation allows only modifications on such first-class citizens which prevents model and generated code running out-of-sync, indirectly solving problems known from round-trip-engineering [HLR08]. It is important to notice that full-code-generation does not necessarily mean that applications are created from generated code alone. It is merely a method to generate vitally important parts of an application into a given framework.

In case of the CINCO project, this leads to the conclusion: To reduce verbosity and increase simplicity of the specification languages, it is important to have two parts: First, a powerful framework that serves as the base of every generated modeling tool, and second,

a capable code-generator that can use the more abstract descriptions of the languages and generate all the needed artifacts to turn the framework into the desired modeling tool.

1.1 My Contribution

The first main contribution of this dissertation is the formal description of a meta specification language for graph-based model structures. The CINCO meta-tooling suite features three specification languages, first introduced in [NLKS17], that are used to define domain-specific graphical modeling tools. The first language is the *Meta-Graph Language* (MGL), a specification language that is used to describe graph-models and its model element types (nodes, edges and containers). The other languages used in CINCO are the *Meta Style Language* (MSL), a language which is used to describe the visual appearance of defined element types, and the CINCO *Product Definition* (CPD) language that is used to define the tool configuration. These two languages, together with the MGL, form the essential artifacts for the CINCO *Product Generation Process* (CPGP). This contribution focuses on the MGL specification which was designed and implemented by myself, while the MSL was designed by Dawid Kopetzki originating from his Master thesis [Kop14].

The second main contribution of this dissertation is the introduction of a product generation process that can be used to create full-fledged modeling environments from high-level domain-specific language descriptions. The approach is illustrated by the *CPGP*. The *CPGP* defines the necessary steps to produce a standalone Eclipse [MLA10] based modeling tool (a CINCO product) from a set of textual meta-level specification languages, predefined libraries and other resources necessary to complete the generation. The realization of the *CPGP* builds on concepts of CINCO introduced in [NLKS17] and [Nau17] but has not so far been detailed itself in any publication.

The described process can be divided into 4 phases:

1. **Resource Aggregation:** Necessary resources, such as language descriptions, are gathered. Language descriptions are topologically sorted by dependency [Kah62]. Unchanged and previously generated resources that have no changed dependencies do not need to be generated.

2. **Abstract Syntax Generation:** In case of the *CPGP*, for each language definition used in the product, Ecore meta-models [SBPM08] and according generator models are generated that serve as abstract syntax for the DSLs. In addition to typical EMF operations, the generated Ecore model includes an API that is specially created to simplify the use of the CINCO models. The Java Classes for the Ecore models is generated from the Generator models.

3. **Concrete Syntax and Editor Generation:** Using the descriptions, editors that ensure conform the models are generated. For every language at least one editor is gen-

erated. In case of the *CPGP*, two editors a Graphiti [6] based editor and a Xtext [16] based editor are generated by default.

4. **Product Configuration:** After the generation is finished, the whole product has to be assembled out of the created artifacts.

The third main contribution of this dissertation is the generation of a domain-specific API based on the same high-level descriptions used for the product generation. The generation of the API makes use of concepts and technologies described in [Jö11, JMS08] and extended in [Lyb12]. In case of Cinco the generation of the Cinco *Transformation API* (CT-API) has been realized. The CT-API provides the vital functionality of a Cinco product which makes it easy to implement tool behavior and semantics through plug-ins. It uses information present at generation time to create specific operations that are useful for transformations on graph-models (such as typed successor/predecessor or containment relationships). Therefore the CT-API for all products is generated during the execution of the *CPGP*. The CT-API makes it easy to develop any extensions for the Cinco product due to its domain-specific nature and the ability resembling user actions in the generated editors. Both the realization of the *CPGP* and the generation of the CT-API follow the concepts of full-code-generation [KT08].

To demonstrate the suitability of the *CPGP* and the usability of the Cinco transformation API, several papers using Cinco have been published. Some of the publications listed in the next section show different examples for the usage of Cinco meta-modeling suite. Cinco was also used in numerous student project groups of which I have supervised the following [BDG+15, STK+16, AKS+18, BCK+19].

1.2 Context of Attached Publications

Following the development of the Cinco meta-tooling suite, various different examples of applications have been developed to illustrate the concepts behind the suite. The publications affiliated with this cumulative dissertation show a variety of applications (e.g. for educational purposes [LKZ+18], model transformations [LKS18] or web application development [BFK+16]).

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

This paper [NLKS17] introduces Cinco, a domain-specific tool which is used to define domain-specific modeling products. Built on the foundations of the *Eclipse Rich Client Platform* (RCP) [MLA10] and more importantly on the Eclipse project *Eclipse Modeling Framework* (EMF) [SBPM08] and technologies based thereof, such as Graphiti [6] and Xtext [16], Cinco uses a family of high level specification languages to describe graph-based modeling tools. This publication introduces basic concepts and the meta-level specification

languages, but also shows a variety of academic and industrial applications, and features a comparison with similar tools.

A Tutorial Introduction to Graphical Modeling and Meta-modeling with Cinco

This paper highlights the capabilities of Cinco in terms of adaptability and ease of use. It is done by showing a short educational example and presenting how the modification of the specification affects the code generator [LKZ+18]. The tutorial uses a newly created graphical Cinco specification language *GCS* detailed in [Fuh18] and makes use of the Cinco generation target for the web: *Pyro* [Zwe15]. Cinco has been used for the implementation of various tools. The next three papers show examples of varying complexity that can be implemented.

Design for 'X' through Model Transformation

In this paper [LKS18] a framework for model transformations is sketched that can be used to ensure a series of *X by Construction* (XbyC) [tBCSW18] properties such as model-checkability, learn-ability, or performance by construction. It is done by specifying the source and target languages but also the transformation language using the Cinco meta-modeling suite. It makes use of a graphical syntax similar to structured operational semantics [Plo81] using a multi-level rule pattern with micro and macro steps, similar to the semantics of synchronous systems [BG92].

DIME: A Programming-Less Modeling Environment for Web Applications

This paper [BFK+16] describes a complex application of Cinco, the *DyWa Integrated Modeling Environment* (Dime). Dime is a fully realized *Integrated Modeling Environment* (IME) for modeling complete web-applications. It features a family of DSLs for implementing GUIs, Data and Processes. Dime's process model is based on principles used in predecessing technologies namely the *jABC4* [Neu14] and the data modeling relies heavily on the *Dynamic Web Application* (DyWA) [NFSM14]. Dime itself depends on the concepts of the *One Thing Approach* (OTA) [MS09] and *Exteme Model-Driven Development* (XMDD) [MS12].

1.3 Nomenclature

During this dissertation, it may not be easy to distinguish between keywords in presented domain-specific languages (DSL), types of the abstract syntax of these DSLs, and types used in a Cinco product, generated from these DSLs or previously implemented, since keywords and types are often called similar (e.g. `node`, for the keyword that tags a *Node*

type definition). So, whenever a keyword or anything code-related is mentioned, it is displayed this way: `keyword`. When a type of the abstract syntax is used, it is displayed this way: *Type*. When a generated type specific for a CINCO product is described, it will look this way: ***Product Type***.

1.4 Organization of this Dissertation

Following this introduction, Chapter 2 will introduce the concepts and technologies used for CINCO and the *CPGP*. Chapter 3 describes the specification languages of the CINCO meta tooling suite, with focus on the MGL (the first main contribution). While the MSL and CPD are not part of the contribution but are connected to the MGL and the *CPGP*, and thus are only described briefly. Chapter 4 describes the other contributions of this dissertation, the *CPGP* and the CT-API, with the construction of the CT-API covered in Section 4.3. The dissertation finishes with a conclusion and a view on possible future extensions in Chapter 5.

Chapter 2

Background

This chapter introduces the main concepts and technologies used for the realization of the the CINCO meta-tooling suite.

2.1 Domain-Specific Languages

The term *Domain-Specific Language* (DSL) has been in use for quite some time [Ham75]. Usually, it describes a specialized language that is tailored to describe solutions for a specific problem domain. According to Fowler, DSLs are popular for several reasons. The two main ones are: "improving productivity for developers" and "improving communication with domain experts." [FP11] An issue that is often tried to solve using DSLs is the semantic gap [Gho11]. Domain expert may not understand the concepts of general-purpose languages, while the developer that needs to implement the solution does not necessarily have the knowledge of the domain experts required for developing solutions for the domain's problems. DSLs can be introduced to close or at least narrow this gap by empowering the domain expert to express his own solutions or to understand solutions formalized by others.

Types of DSLs can be distinguished between internal DSLs (also called embedded DSLs [FP11]) and external DSLs. Internal DSLs rely on a host language and use a subset of the host's syntax. Building on top of another language, an internal DSL has the advantage that it can make use of the host language's tool chain (e.g. editors, compilers etc.). Languages that support the creation of internal DSLs are for example: Racket (see [FFF+18] or [12]) or Scala [13]. External DSLs, on the other hand, are independent from other languages which means they are more free to introduce syntactic concepts that are easier to understand for the domain expert. The focus of the CINCO meta-tooling suite clearly lies on external DSLs. These external DSLs can also be categorized into graphical(or visual) and textual DSLs. An example for tools that helps creating textual DSLs is the framework Xtext [16]. A framework for the development of graph-based external DSLs with a graphical syntax, is the Graphiti [6] framework. Both frameworks, Xtext and Graphiti, are based on the *Eclipse Modeling Framework* (EMF) which itself is part of

Eclipse [2]. The semantics of DSLs can be formalized in several ways, being it operational semantics [Plo81], axiomatic semantics [Hoa69], or translational semantics [Kle08]. The *CPGP* clearly focuses on the latter using code generation, while CINCO also supports the creation of tools using other forms of semantics (for example as presented in [LKS18]).

2.2 Eclipse

Eclipse is an open source platform, that can be used to develop software tools. It has been used as a basis for several tools, for example Java, C++ or Python development environments. Eclipse is divided into several top-level projects, such as the *Eclipse Modeling Project*, that encompass smaller sub-projects.

2.2.1 EMF

An important part of the *Eclipse Modeling Project* [3] is the *Eclipse Modeling Framework* (EMF) [SBPM08]. This project provides modeling and code generation tools for software development with a structured data model. Data models are described in Ecore, a reflexive meta meta-model. This model describes data in form of a class-based model, not unlike the class diagram of UML [19]. It provides data types that mirror Java types (e.g. `EInt` for `int`, `EBoolean` for `boolean`). It is also possible to add operations to the Ecore model with so called `EOperations`. These operations need to be implemented either inside the generated code, which conflicts with the full-code generation paradigm, or by annotating the defined `EOperations` with the operation body inside the Ecore model, which lacks any IDE features. To use the Ecore models, instances of models can be created that confirm to the definitions made in the Ecore model. These model instances are also called EMF models. The EMF models can be created and edited through various means. EMF itself, for example, can generate a simple tree-based editor that can be used. It is also possible to create and edit the models using the EMF-API. Ecore is a reflexive meta-model, in such a way, that it can be used to describe itself, which means that every Ecore model is also an EMF model. Several projects use EMF and the Ecore model, such as Xtext and Graphiti, which are are used by the CINCO meta-tooling suite.

2.2.2 Xtext

Xtext [16] is a development environment for creating external textual DSLs. It uses a *grammar language* to describe the concrete syntax of a DSL and how it is mapped to the in-memory model. Internally, Xtext generates an Ecore model from such a description, that represents the data model of the abstract syntax. Xtext also generates an editor for the language with syntax checks, a framework for additional validation and a framework for code generators. Code generators and validation, can be implemented using Xtend (See Section 2.3.1) or Java.

2.2.3 Graphiti

Graphiti [6] is a tooling infrastructure that facilitates the development of Eclipse based diagram editors, that can use Ecore models as domain models. It uses GEF [4] and Draw2d [5] as base technologies, but abstracts from using an API.

Graphiti uses them to define the graphical appearance of its models and to link its own Diagram meta-model to the corresponding domain models. For each graphical model element type, a set of features is required that defines the tool behavior, for example *Create Feature* to create an instance of a model element or *Add Feature* to add a created element into the model. Other examples for features are *Move Feature* or *Delete Feature*. All defined elements can be added to a palette from which they can be drawn to the editor canvas which will trigger the *Create Feature* first and then the *Add Feature*.

2.3 Code Generation

An essential part and main focus of the Cinco meta-modeling suite is the generation of standalone modeling tools, which make use of code generation techniques, while depending heavily on the reuse of already existing software. To clarify which type of code generation used in the aforementioned *CPGP*, it is important to classify the term correctly. Code generation has been used in several fields, such as compiler construction [Sun13] or in model-driven approaches, such as *Model-Driven Software Development* (MDSE) [SVEH07]. It also plays an important part in *Language-Driven Engineering* (LDE) [SGNM19]. Though many of the code generation concepts used in compiler construction can directly be applied to MDSE [Sel03], generators of both fields differ from each other in terms of abstraction. Code generation for domain-specific languages has more in common with code generation in model-driven software development than compiler construction, since DSLs usually work on an even higher abstraction level than Model-Driven approaches. In context of the meta-tooling suite the code is generated by collecting the information found in the specification languages.

There are several approaches to code generation. The simplest form is programming a code generator by hand, which traverses the underlying specifications and creates the output via string concatenation. While this seems feasible for small portions of code, there are some problems. First, there is no separation between generator logic and the output. This makes it hard to reuse the code generator or to maintain it. Second the concatenated code is made of small parts of predefined code and data read from the underlying model (e.g attribute values). Also, special characters (e.g double-quotes) have to be escaped. This makes it hard to maintain the generated code if errors occur or features need to be added.

2.3.1 Template-Based Code Generation

Another, more sophisticated approach for code generation is the use of templates. Templates are basically larger portions of text with placeholders filled with the information found in specifications. Template-based generation has been an important part for web-development, but can also be used to generate code where large parts stay the same (e.g. boiler-plate code). It was used in server-side languages, such as *PHP* [11] (often assisted by external templating languages, such as *Twig* [14]) or *Active Server Pages.NET* (ASP.NET) [1], but is also used in client-side approaches, such as *Mustache* [10] or *Handlebars* [7], often in connection with *JSON* [18] as the underlying data model. But template-based approaches have also been used outside the web environment. *Xtend* [15] is a programming language that features templating using *Template Expressions*. It is the default language for implementing code generators and tool behavior for DSLs created with Xtext. Xtend itself is implemented in Xtext and generates to plain Java, which makes it fully compatible with all java projects. The *Template Expressions* can be used inside code but can also be used in separate methods to allow separation of templates and business code. Xtend uses a special syntax with guillemets (« and ») to highlight dynamic code inside the template. Listing 2.1 shows a simple Xtend method that returns a HTML bullet-point list made from a list of Strings using a *Template Expression*. Template based code generation with Xtend is used for the generation of the CINCO transformation API.

```
1 def bullet(List<String> strs)'''
2   <ul>
3   «FOREACH s: strs»
4     <li>«s»</li>
5   «ENDFOR»
6   </ul>
7 '''
```

Listing 2.1: Example for a Xtend *Template Expression*

2.3.2 Model-Driven Code Generation Approaches

Sven Jörges provided in his dissertation [Jö13] a model-driven and service-oriented approach for code generation, Genesys. The Genesys approach is based on the concepts of XMDD [MS12], and the default implementation was implemented in jABC [Nag09]. The code generators are modeled using graphical process models. These processes consist of various predefined services that can be added to the models by using the *Service Independent Building Blocks* (SIB). Third party technologies (e.g., template engines) are inside Genesys by providing SIBs for these technologies. Models can be divided into hierarchical sub-models, which allows for separation of concerns and reuse of models. Genesys also allows for a better separation between generation logic and output by using the jABC processes to define the logic, while templates are used only for text output and more ad-

vanced features of template engines (e.g. loops) are forbidden by convention. Another key feature of Genesys is the avoidance of round-trip-engineering. Process models are the only artifacts that are changed during the construction of the code generators. Generators can be designed in a way that the target code is fully generated from these models.

Genesys had a major impact in earlier versions of CINCO. Jörges describes a code generator in [Jö13, Chapter 7], which allows to generate SIBs from Ecore meta-models, that can be used to generate code from instances of this meta-model. This SIB generator was the basis for the TRANSEM *Framework*, which I developed in the course of my diploma thesis [Lyb12], which is used to describe process-based model transformations on EMF models. Since Ecore models are also EMF models, TRANSEM can be used to describe model transformations of Ecore models. Earlier versions of CINCO used TRANSEM to generate Ecore models from high-level specifications (see Section 4.2). This approach has since then been temporarily replaced with a template-based implementation using Xtend to remove old legacy technology, Genesys and TRANSEM were based on (jABC). Though TRANSEM is not present in the current version of CINCO, the ideas and concepts can still be found in the *CPGP* and could be reused when developing a successor using a CINCO process model (e.g. like the Process models in DIME [BFK+16]).

Chapter 3

CINCO

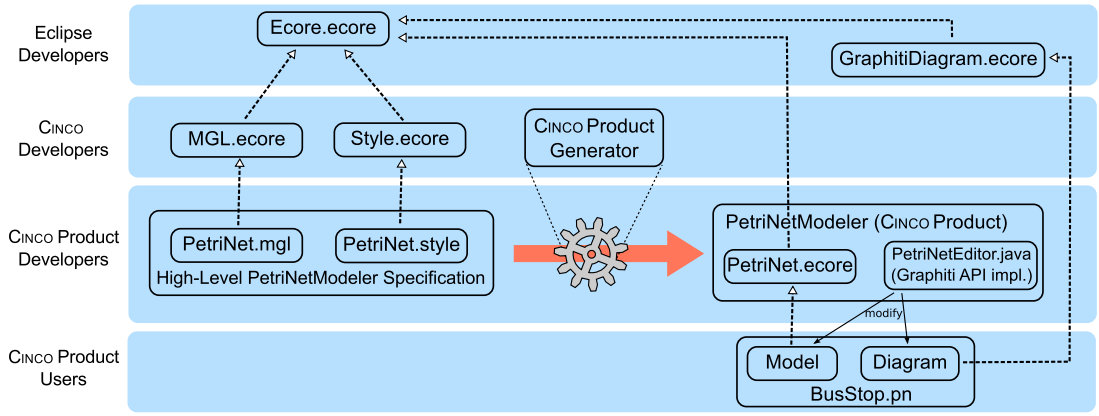


Figure 3.1: Overview of relevant Artifacts (reprinted from [NLKS17])

This chapter describes the CINCO[NLKS17] meta-tooling specification languages with focus on the *Meta Graph Language* (MGL). CINCO is a tooling suite for creating domain-specific modeling tools using high-level specification languages through code generation. The generation is provided by the *CPGP*. CINCO aims to be a tool that can be used to define any tool that is based on graph-based modeling languages consisting of types of nodes and edges. All these tools are specified using three languages:

1. The central language is the CINCO *Product Definition* (CPD). It describes which languages are part of the tool, the branding of the tool, and the other components used in the tool, which are not specified by CINCO's specification languages.
2. The *Meta Graph Language* (MGL) is used to define the abstract syntax of the languages, but also maps defined element types to a concrete syntax.
3. The third language is the *Meta Style Language* (MSL). It defines the styling of elements defined in the languages specified in the MGL.

The three languages are described in the following. Figure 3.1 shows an overview of the artifacts present in one CINCO product and the different roles associated with them:

The CINCO *Product User* only knows the generated tool and how to model their solution. This is the layer where the communication between the domain expert and the developer takes place (as mentioned in Section 2.1).

The CINCO *Product Developer* specifies the tool, using the above mentioned languages and generates it using the *CPGP*.

The CINCO *Developer* defines the specification languages along with the *CPGP*.

The *Eclipse Developer* provides the technologies necessary for CINCO, such as Eclipse, Xtext, Graphiti, and EMF.

3.1 The Meta Graph Language

As mentioned above, the MGL is used to describe the abstract syntax of a model type. A CINCO product often contains editors for multiple languages, so an MGL file is needed for each. This section will define the elements of the MGL syntax. An MGL file can be edited in CINCO using a textual editor generated from an Xtext [16] grammar. This grammar is based on an Ecore [SBPM08] meta-model, which represents the class model of the abstract syntax tree of the MGL. Figure A.1 in the appendix shows the full class diagram of the MGL. Xtext can be used to provide a mapping from concrete syntax to the abstract syntax depicted in the Ecore model. This mapping may distract from the the concrete syntax, so Listing A.1 and Listing A.2 in the appendix show the cleaned up concrete syntax of the entire MGL specification in form of an *EBNF*[1], which omits the mapping to the Ecore meta-model. In the following, the syntactical elements are described highlighting the concrete syntax, when needed.

3.1.1 MGL Types

Every MGL file defines a number of types that correspond to a set of generated classes in the final product. The MGL always contains a *GraphModel* type as the main type. Other types can be defined, too. These types can either be graphical model elements such as *Node* or *Edge* types or other non graphical types such as the *UserDefinedType* type that serves the purpose of a composed data-type or the *Enumeration* type. Figure 3.2 shows the class hierarchy of the MGL types.

Attributes All types except the *Enumeration* types may have attributes which are defined in the type definition. These attributes specify properties for model elements, if they need to carry more information, than for example, position or connections for *Node* types. To specify attributes, the `attr` keyword is used. Attribute definitions always contain a

[1] Used EBNF Notation from the W3C XQuery 3.1 recommendation [17]

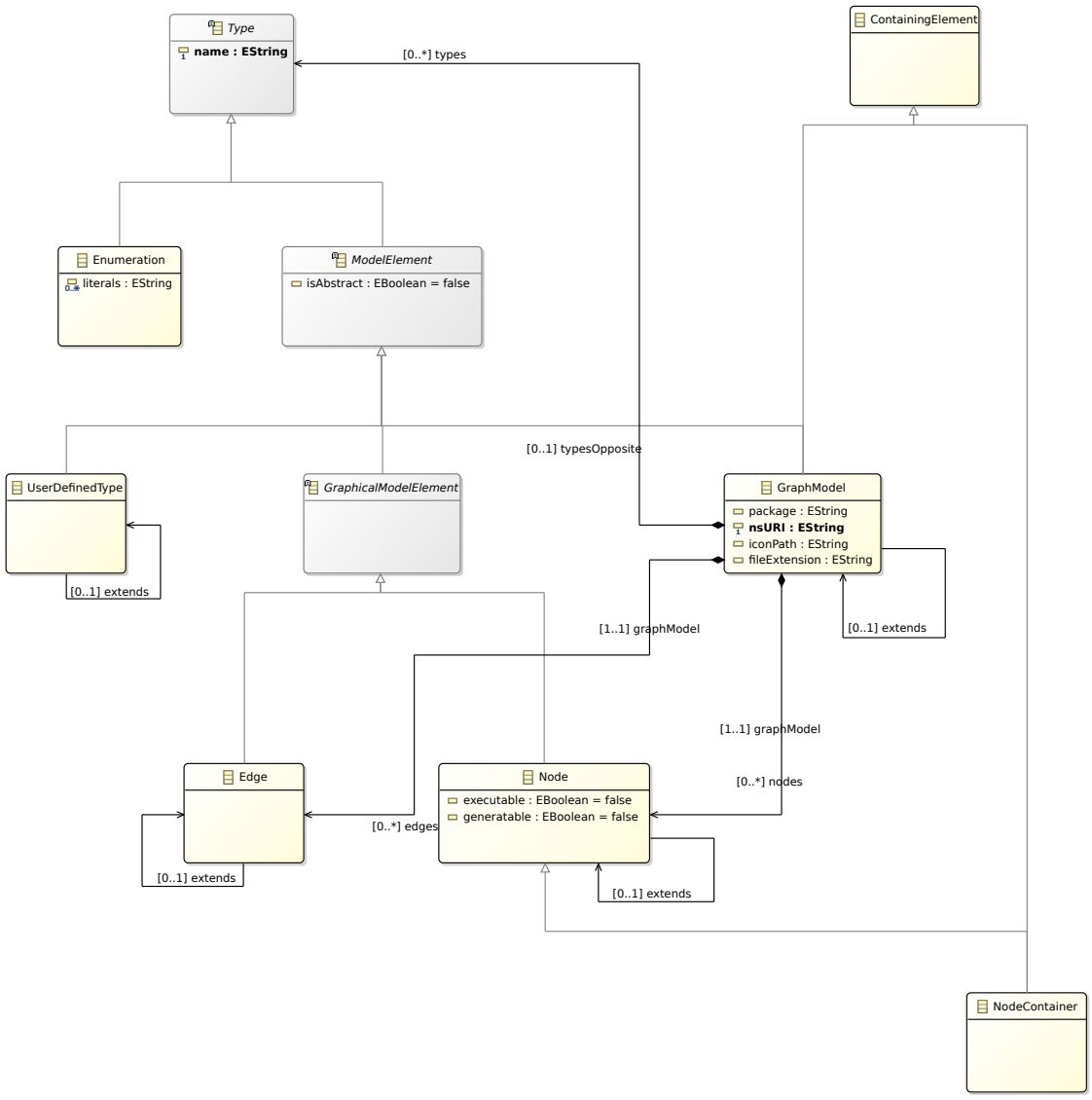


Figure 3.2: Class hierarchy of the MGL types

name and a type. A type may be a primitive data type (such as integer or boolean), but may also be a predefined complex type (for example a *UserDefinedType* or a *Node* type). An attribute may also carry a list of elements which are defined by providing an lower and upper bound for it (as seen in Listing 3.1), for example:

```
attr UserData as users[0,*]
```

GraphModel Type

Every MGL description starts with a *GraphModel* element emphasized by the `graphModel` keyword and a *name*. Listing 3.2 shows the grammar rule for this element. The *GraphModel* type consists of all elements that can be used in a graph-model of this type. Other MGL files or Ecore files may be imported for prime references (see pg. 17) using `import` as the

```
13 Attribute ::= PrimitiveAttribute | ComplexAttribute
14 PrimitiveAttribute ::=
15   Annotation*
16   'final'? 'unique'? 'attr' EDataTypeType 'as'  EString ( '[' EInt ( ',' BoundValue )
         ? ']' )? ( ':=' EString )?
17 ComplexAttribute ::=
18   Annotation*
19   'final'? 'unique'? 'override'? 'attr' EString 'as' EString ( '[' EInt ( ','
         BoundValue )? ']' )? ( ':=' EString )?
```

Listing 3.1: Rule for Attributes

```
1 GraphModel ::=
2   Import*
3   Annotation*
4   'graphModel' EString ( 'extends' EString )?  '{'
5     ('package' QName )?
6     'nsURI' URI
7     ( 'iconPath' EString )?
8     'diagramExtension'  EString
9     ( 'containableElements' '(' GraphicalElementContainment ( ','
          GraphicalElementContainment )* ')' )?
10    Attribute*
11    ( Node | Edge | NodeContainer | Type )*
12  '}'
```

Listing 3.2: Rule for the *GraphModel* type

keyword. At the top of the *GraphModel* definition information is entered for the generated editor such as a main Java package name or the file extension of the model type. It is also defined which model elements may be contained directly in the *GraphModel* using the `containableElements` keyword. A graph model may have attributes. Inside the *GraphModel* every other model element part of the model type is defined. Possible types of these elements are *Node*, *Container*, *Edge*, *UserDefinedType* or *Enumeration*. Listing 3.3 shows an example of the head of a *GraphModel* type.

```
3 @style("model/FlowGraph.style")
4 graphModel FlowGraph {
5   package info.scce.cinco.product.flowgraph
6   nsURI "http://cinco.scce.info/product/flowgraph"
7   diagramExtension "flowgraph"
8
9   attr EString as modelName
10  [...]
11 }
```

Listing 3.3: Example for a *GraphModel* type

```
24 Node ::=
25   Annotation* 'abstract'?
26   'node' EString ( 'extends' EString )? '{'
27     ( Attribute*
28       | ReferencedEClass
29       | ReferencedModelElement
30       | ( 'incomingEdges' '(' IncomingEdgeElementConnection ( ','
             IncomingEdgeElementConnection )*
31       |  'outgoingEdges' '(' OutgoingEdgeElementConnection ( ','
             OutgoingEdgeElementConnection )* ) ')' )*
32   '}'
```

Listing 3.4: Rule for the *Node* type

```
52 ReferencedType ::= ReferencedEClass | ReferencedModelElement
53 ReferencedEClass ::= Annotation* 'prime' ID '.' ID 'as' EString
54 ReferencedModelElement ::= Annotation* 'prime' ( 'this' | QName ) '::' QName 'as'
       EString URI ::= EString
```

Listing 3.5: Rule for the *ReferencedModelelement* and *ReferencedEclass*

Node Type

Using the `node` keyword, a *Node* type is defined. Listing 3.4 shows the rule for the *Node* type. Like all other model element types nodes can also have attributes. For reuse purposes *Node* types may inherit properties from other *Node* types. A *Node* type may also be `abstract` meaning that it can't be instantiated in a model editor, but may be used as parent element for other *Node* types. Possible connections between nodes are defined by the `incomingEdges` and `outgoingEdges` keywords using the connection constraints which will be described in Section 3.1.1.

Prime References A special *Node* type can be specified by adding the `prime` keyword to the *Node* type definition. This keyword indicates that the defined type will act as a representative of another model element from a different file. This may be a model element of a type defined in this MGL or an element whose type is defined in an imported Ecore or an imported MGL file (see Listing 3.5). This feature can be used, for example, for defining hierarchy in models by referencing a *GraphModel* or referencing information from completely different model types. Listing 3.6 shows the prime node ***ExternalActivity*** that references an activity, that was defined in an external library model, which was modeled using EMF.

Container Type

The *NodeContainer* type behaves like a *Node* type, but with the addition that it may contain other nodes or containers. Other than that, a *Container* type has the same properties

```
41  node ExternalActivity {
42    @pvLabel(name)
43    @pvFileExtension("elib")
44    prime externalLibrary.ExternalActivity as activity
45    incomingEdges (*[1,*])
46    outgoingEdges (LabeledTransition[1,*])
47  }
```

Listing 3.6: Example for a Prime *Node* type

```
46 GraphicalElementContainment ::= ( QName | '{' QName ( ',' QName )* '}' | '*' ) ( '['
       EInt ',' BoundValue ']' )?
```

Listing 3.7: Rule for the *ContainmentConstraints*

as a *Node* type. A *Container* type may even inherit properties from defined *Node* types or act as an representative for a prime referenced model element. Adding a *NodeContainer* type to the MGL definition is done using the `container` keyword. Otherwise, it is defined just like a *Node* type. Containment may be specified by using the *Containment Constraints*.

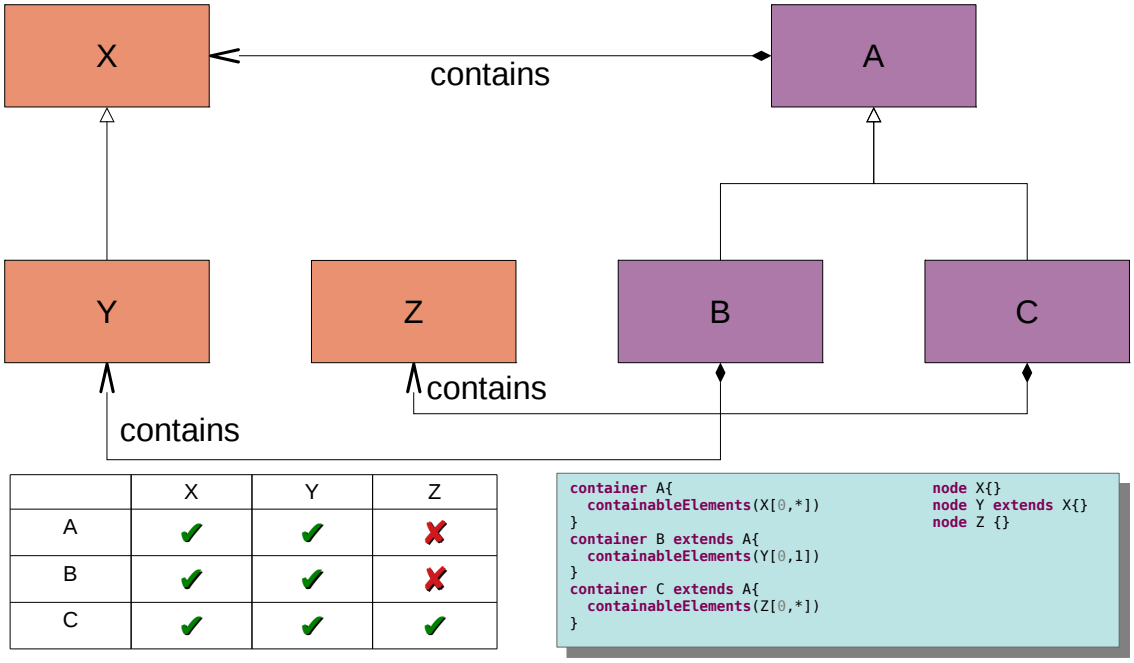
Containment Constraints define possible combinations of nodes and containers that can be contained in a *ModelElementContainer*, such as a *NodeContainer* or a *GraphModel* type. Listing 3.7 shows the syntax of a containment constraint. A constraint consists of a list of possible *Node* or *Container* types which the container may contain. It also features an upper bound and a lower bound. The bounds may be omitted if the number is arbitrary. Instead of a list of elements it may also be a single element, or a wild card (*).

It is also possible to define that a specific type may not be contained, by using the bounds `[0,0]`. The generated editor will take care that no upper bounds can be violated. Lower bounds are checked using a validator plug-in.

A constraint reads as followed: The amount of all elements that are of the types listed in the constraint inside this graph model must be higher or equal to the lower bound and must be lower or equal to the upper bound. Multiple constraints may be defined which may also overlap. Listing 3.8 shows five constraints, in this example one for each *Node* type listed, which have to be respected by the *Container* type ***Swimlane***. The *Node* types

```
57  container Swimlane {
58    attr EString as actor
59    containableElements (Start[1,1], End[1,1], Activity, ExternalActivity,
          SubFlowGraph)
60  }
```

Listing 3.8: Example for a *Container* type with a containment constraint



	X	Y	Z
A	✔	✔	✘
B	✔	✔	✘
C	✔	✔	✔

Figure 3.3: Example of Containment Constraint Inheritance

```
21 Edge ::=
22   Annotation*
23   'abstract'? 'edge' EString ( 'extends' EString )? ( '{' Attribute* '}' )?
```

Listing 3.9: Rule for the *Edge* type

Start and ***End*** must occur exactly once. A container may have an abritrary number of *Activity* nodes and must contain one of either ***ExternalActivity*** or ***SubFlowGraph***.

Figure 3.3 shows an example for containment constraints with inheritance. Container type ***A*** may contain an arbitrary number of type ***X*** nodes. Node type ***Y*** is child of ***X***, which means that ***A*** also may contain nodes of ***Y***. ***A*** has two children ***B*** and ***C***. Both children inherit the containment constraint from ***A***. Type ***C*** also adds a containment constraint stating that it may contain an arbitrary number of nodes of type ***Z***. Type ***B*** restricts the constraint of type ***A*** with the added constraint for node type ***Y***. It states that ***B*** may contain exactly one node of type ***Y***. Since ***Y*** is child of ***X*** this means that, while ***B*** may contain an arbitrary number of ***X*** nodes, only one of them may be of type ***Y***.

Edge Type

Edges define connections between nodes. An *Edge* type can be defined using the `edge` keyword. *Edge* types may inherit properties from other *Edge* types and may be defined as `abstract`. Like other model elements, edge types may also contain attributes. To specify which types of connections are available in the editor, for each *Node*, it is possible

```
67  edge LabeledTransition {
68    attr EString as label
69  }
```

Listing 3.10: Example for an *Edge* specification

```
44 OutgoingEdgeElementConnection ::= ( QName | '{' QName ( ',' QName )* '}' | '*' ) ( '[
       ' EInt ',' BoundValue ']' )?
```

Listing 3.11: Rule for the outgoing *ConnectionConstraints*

to define connection constraints. Listing 3.9 shows the grammar rule for the *Edge* type and Listing 3.10 shows a simple edge with an `EString` attribute for a label.

Connection Constraints Connection constraints define which *Edge* types may start or end in a *Node* type. They are defined inside the `node` definition using the `outgoingEdges` keyword for outgoing (starting) constraints and the `incomingEdges` keyword respectively for incoming (ending) constraints. A constraint consists of a list of possible edge types listed in curly brackets and an upper and lower bound. This means that the amount of the edges that are affected by this constraint must be equal or higher to the lower bound but lower or equal to the upper bound. Multiple constraints may be added. Like the containment constraints, the generated editor will take care that none of the upper bound connection constraints will be violated, while lower bounds are checked using the an model validation plug-in. Listing 3.11 defines the Xtext rule for outgoing connection constraints. The rule for incoming constraints is analog to this. When *Node* types inherit properties from their parent types, they also derive connection constraints from them.

Figure 3.4 shows an example how inheritance can affect these constraints and how the connection constraints are expressed in the MGL. The `abstract` *Node* type $\boldsymbol{A}$ has two children $\boldsymbol{B}$ and $\boldsymbol{C}$. Type $\boldsymbol{A}$ may have outgoing edges of type $\boldsymbol{X}$, which is also `abstract`. $\boldsymbol{B}$ may have incoming edges of type $\boldsymbol{Y}$ and $\boldsymbol{C}$ incoming edges of type $\boldsymbol{Z}$. $\boldsymbol{Y}$ and $\boldsymbol{Z}$ are both children of $\boldsymbol{X}$. This leads to four kinds of possible connections: A source node of type $\boldsymbol{B}$ may connect to another type $\boldsymbol{B}$ using an edge of type $\boldsymbol{Y}$. Source $\boldsymbol{B}$ may also connect to a node of type $\boldsymbol{C}$ using an edge of type $\boldsymbol{Z}$. This is possible because $\boldsymbol{Y}$ and $\boldsymbol{Z}$ are both children of $\boldsymbol{X}$, and since $\boldsymbol{B}$ is child of $\boldsymbol{A}$ it inherits the connection constraint saying that $\boldsymbol{A}$ may have outgoing edges of type $\boldsymbol{X}$. Analog to this, a source node of type $\boldsymbol{C}$ may have outgoing connections of type $\boldsymbol{Y}$ to nodes of type $\boldsymbol{B}$ and outgoing connections of type $\boldsymbol{Z}$ to nodes of type $\boldsymbol{C}$.

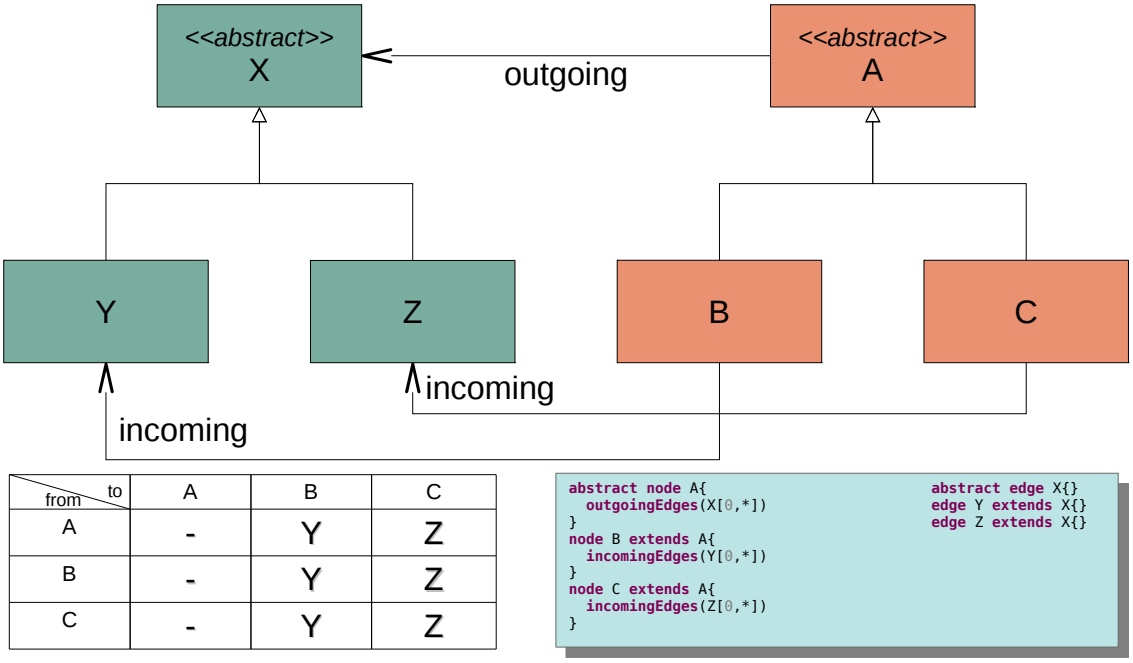


Figure 3.4: Example of Connection Constraint Inheritance

```
1 UserDefinedType returns UserDefinedType:
2   {UserDefinedType}
3   (annotations+=Annotation)*
4   (isAbstract?='abstract')? 'type' name=EString
5     ('extends' extends=[UserDefinedType|EString])?'{'
6       (attributes+=Attribute)*
7     '}'
8 ;
```

Listing 3.12: Rule for the *UserDefinedType* type

UserDefinedType

A *UserDefinedType* type (as seen in Listing 3.12) describes a type that does not have a graphical representation. It is defined using the `type` keyword and is used as a complex data type attribute of other model elements. *UserDefinedType* types can inherit from other *UserDefinedType* types and may be `abstract`. Their complex structure can be edited via a tree-based editor along with other attributes in the CINCO properties view.

Enumeration Type

Listing 3.13 shows the grammar rule for another simple data type, the enumeration, which is defined with the `enum` keyword. It can be used as a type for attributes in all model elements and is used to assign predefined values to a model element. See for Example the Listing 3.14.

```
1 Enum returns Enumeration:
2   {Enumeration}
3   (annotations+=Annotation)*
4   'enum' name=EString '{'
5     literals+=EString
6     (literals+=EString)*
7   '}'
8 ;
```

Listing 3.13: Rule for the *Enumeration* type

```
70   enum Direction{ North East South West }
```

Listing 3.14: Example for an *Enumeration* specification

3.1.2 Annotations

At every defined MGL type or any of its attributes, it is possible to add annotations that have various effects on tool generation or tool behavior. Annotations can be divided into three categories:

Style Annotations are annotations that link MGL types to MSL *Styles* (further explained in Section 3.2.2) and register MSL models to the graph model. Every non-abstract graphical MGL type has to have one Style annotation.

Meta Plug-in Annotations are annotations that activate or configure meta plug-ins (see Section 4.4). Meta plug-ins are activated by adding the main annotation to the `graphModel` definition. Annotations that configure the meta plug-ins can be placed anywhere they are required.

Hook Annotations are annotations that can add additional behavior to the tool based on tool actions (e.g. *create, move, delete*). For any possible action, a hook can be defined that is triggered before or afterwards the event is resolved (e.g. `@preMove`, `@postMove`). The annotation references a Java class which is implemented confirming to a hook-specific interface. Every class implements an `execute` method that is called when the associated action is carried out.

3.2 The Meta Style Language

The *Meta Style Language* (MSL) is used to describe the visual presentation, meaning the concrete graphical syntax of the model types used in a CINCO product. The language was designed by Dawid Kopetzki and described in his Master thesis [Kop14]. It defines graphical styles for all visual elements of the tool, such as nodes, containers and edges. Furthermore, the MSL makes it easy to reuse certain aspects of a style, such as background color, line width, or line style by introducing *Appearance*s.

```
1  Appearance returns Appearance:
2    {Appearance}
3    'appearance' name=ID ('extends' parent=[Appearance])? '{'
4      (
5        ('angle' angle=EFloat)? &
6        ('background' background=Color)? &
7        ('foreground' foreground=Color)? &
8        ('font' font=Font)? &
9        ('lineStyle' lineStyle=LineStyle)? &
10       ('lineWidth' lineWidth=INT)? &
11       ('transparency' transparency=EDouble)? &
12       ('filled' filled = Boolean)? &
13       ('imagePath' '('imagePath=STRING')')?
14     )
15   '}'
16 ;
```

Listing 3.15: Appearance rule of the Meta Style Grammar

3.2.1 Appearances

Appearances (see Listing 3.15) facilitate the reuse of basic styling information throughout a tool. They modify simple styling, such as background color for *NodeStyles*, or line style, width and color for *EdgeStyle*s and border for *NodeStyle*s. They may be defined beforehand using the `appearance` keyword and then be referenced in the style definitions, but may also be defined inside the style definitions instead using *InlineAppearance*s. To facilitate reuse, appearances may also extend existing appearances. Instead of using static appearances,it is also possible to add dynamic appearances by registering an *Appearance Provider*.

3.2.2 Styles

Every non-abstract graphical element type defined in the MGL needs to have a *Style*. Multiple types may share the same *Style* description to reuse basic graphical elements. Styles are distinguished by the MSL between *NodeStyle*s for *Node* and *Container* types, and *EdgeStyle* for *Edge* types. Styles are added to the defined element types using the `@style` annotation. This annotation contains the name of the style. It may also have arguments for parameters, which are written in the *Java Expression Language* [8], making it possible to reference attributes of the defined MGL element type.

NodeStyles are built from shapes. Listing 3.16 shows the Xtext rule for a *NodeStyle*. Every *NodeStyle* needs at least one container shape[2]. Possible container shapes are: *Rectangle*, *RoundedRectangle*, *Ellipse* and *Polygon*. Container shapes may contain other shapes put in position, such as additional container shapes or non-container shapes, such as: a

[2]Note, that container in this context has nothing to do with *Container* types of the MGL, but is used to emphasize that this kind of shape may contain other shapes.

```
1 NodeStyle returns NodeStyle:
2   'nodeStyle' name=ID ('(' parameterCount=INT ')')? '{'
3     ('appearanceProvider' '('appearanceProvider=QName')')?
4     (fixed?='fixed')?
5     mainShape=AbstractShape
6   '}'
7 ;
```

Listing 3.16: Node style rule of the Meta Style Grammar

```
1 RoundedRectangle returns RoundedRectangle:
2   {RoundedRectangle}
3   'roundedRectangle' (name=EString)?
4   '{'
5     (('appearance' referencedAppearance=[Appearance]) | (inlineAppearance=
          InlineAppearance) )?
6     ('position' position = AbstractPosition)?
7     'size' size = Size
8     'corner' '('cornerWidth=INT','cornerHeight=INT')'
9     (children+=AbstractShape)*
10  '}'
11 ;
```

Listing 3.17: Rounded rectangle rule of the Meta Style Grammar

text label (*Text*), multi-lined text label (*Multitext*), an *Image* or a *Polyline* shape. Text labels may be parameterized to allow dynamic values using *Format Strings*[9]. Listing 3.17 shows the grammar rule for the *RoundedRectangle* container shape. Other container shapes have a similar structure with minor differences. Every shape may have an Appearance (see Sec. 3.2.1). Except for the *Polygon* shape, every container shape has a size defined by height and width. *Polygon* are defined by their points, which means the size of the shape is implicitly determined by its vertices. The *RoundedRectangle* shape does have an additional property: *corner*. The `corner` keyword defines horizontal and vertical radius of the round corners. The *RoundedRectangle* shape is shown by the node style defined in Listing 3.18. This style contains a text label which takes one argument and displays it alongside a text in the center of the container shape. Figure 3.5 shows how a node using this style will look like.

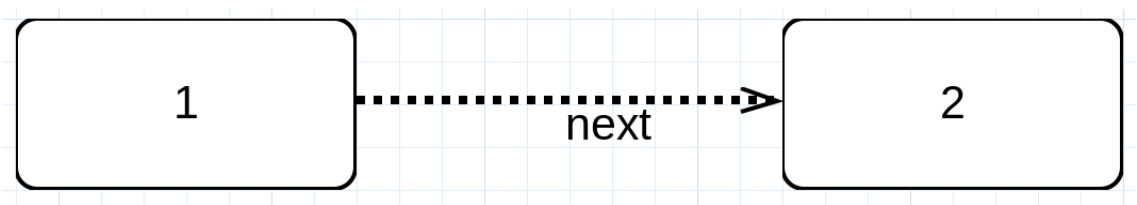


Figure 3.5: Example of two rounded rectangle styled nodes connected with a dotted arrow

```
1 nodeStyle multiRectangle (1){
2   roundedRectangle{
3     size (80,40)
4     corner(10,10)
5     text{
6       position (CENTER,MIDDLE)
7       value "%s"
8     }
9   }
10 }
```

Listing 3.18: Node style with *RoundedRectangle* container shape

```
1 EdgeStyle returns EdgeStyle:
2   'edgeStyle' name=ID ('(' parameterCount=INT ')')? '{'
3     ('appearanceProvider' '('appearanceProvider=QName')')?
4     (('appearance' referencedAppearance=[Appearance]) | (inlineAppearance=
          InlineAppearance) )?
5     ('type' connectionType=ConnectionType)?
6     (decorator+=ConnectionDecorator)*
7   '}'
8 ;
```

Listing 3.19: Edge style rule of the Meta Style Grammar

Edge Styles describe the visual presentation of *Edge* types. Listing 3.19 shows the grammar rule for the *EdgeStyle*. This may be as simple as a thin line, but may also have different line styles, such as dotted lines, or decorators such as arrow heads or text labels for the edges. Line style, color and line width definitions are provided by *Appearance*s. An *EdgeStyle* may have multiple *Decorators*. Possible *Decorators* are shapes, but for easier use macros for common geometric figures are provided such as *ARROW*, *DIAMOND*, *CIRCLE* or *TRIANGLE*. Decorators may be located relatively on the edge by using the `location` keyword and a value between 0 and 1. Listing 3.20 shows an example for an *EdgeStyle*. It defines a dotted line which is 2 pixels wide with 2 decorators. The first decorator is an arrow located at the end of the edge (`location(1.0)`) and the other decorator is a text decorator with the value `"next"` at the middle of the edge (`location(0.5)`).

3.3 The CINCO Product Definition

An important artifact for the *CPGP* is the CINCO product definition (CPD), as it functions as the starting point of the product generation. Every CINCO product needs exactly one CPD file. The CPD files collects all elements that define the CINCO product. It also consists of a list of the MGLs that are used in the finished product, information about the tool branding (e.g. splash screen, icons), and may contain a selection of Eclipse plug-in descriptors which then will be added to the product in addition to required plug-ins. Every

```
1 edgeStyle simpleArrow {
2   appearance {
3     lineStyle DOT
4     lineWidth 2
5   }
6   decorator {
7     location (1.0) // at the end of the edge
8     ARROW
9   }
10  decorator{
11    location(0.5) // middle of the edge
12    text{
13      value "next"
14    }
15  }
16 }
```

Listing 3.20: Example for Edge Style

CPD starts with the `cincoProduct` keyword and the *name* of the CINCO product. It is followed by an optional `id` and product `version`. Main part of the CPD is the list of *MGLDescription* elements (starting with the `mgl` keyword). They define the used MGL files, which will be part of the CINCO product. The CPD also offers options for customizing the presentation of the CINCO product, called branding. Branding may include the use of graphical icons in different sizes (keywords are `linuxIcon`, `image16`, `image32` ...) as well as a customized `about` page. It may also include a `splashScreen` that is shown during start-up. Since CINCO makes use of the Eclipse plug-in environment, any plug-in or feature of an existing Eclipse may be built into the CINCO product. To add an extension, such as a plug-in or feature, to the CINCO product, it is necessary that this extension is present in CINCO. It is added to the CINCO product by using the `features`

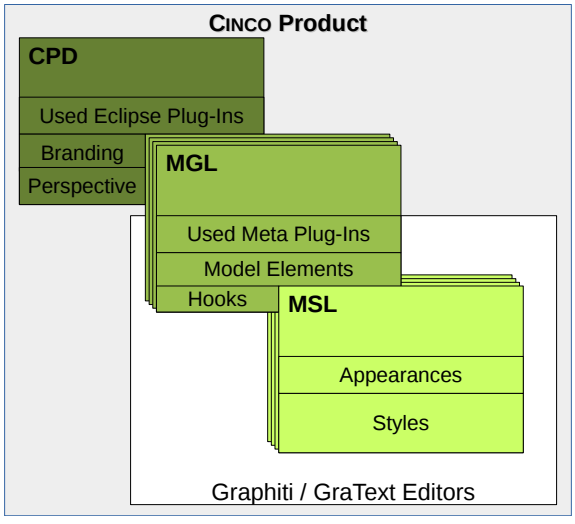


Figure 3.6: Interaction between MGL, MSL and CPD

or `plug-ins` keywords with the `id`s of the extensions. Another option is to define a custom tool perspective. A tool perspective defines which views are open (e.g. editors, property views, project view) or what actions are available. To amplify the *convention over configuration* paradigm, a perspective is generated by default, which can be replaced by a costumized one, if needed.

Figure 3.6 shows how the specification languages are linked between each other and how each language contributes to the CINCO product. As described, the CPD defines the base of the project defining perspective, branding, used Eclipse plug-ins, and which MGLs are used. Each MGL describes a modeling language, defining the model elements, tool hooks and usage of meta plug-ins. Each graphical model element has style and appearances which are defined in an MSL. In the following chapter, the *CPGP* is described, which produces the CINCO product from the three specification languages.

Chapter 4

CINCO Product Generation

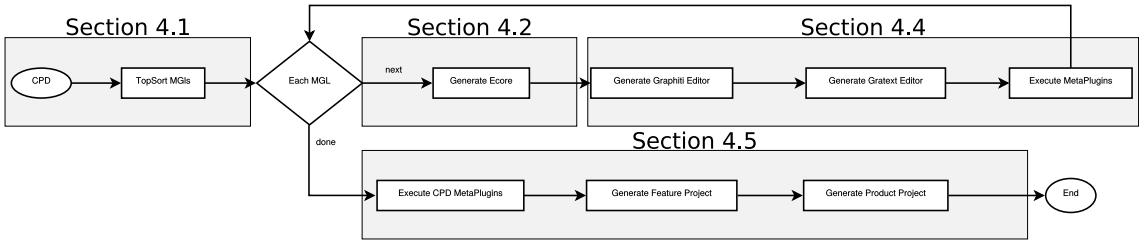


Figure 4.1: Overview of the CINCO Product Generation Process

Central part of a the CINCO product generation is the CINCO Product Generation process (*CPGP*). It describes the process that turns the artifacts created by the tool developer (MGL, MSL, CPD) into a running CINCO product. Figure 4.1 shows an overview of the generation process. The input for the process is a CPD file. The *CPGP* is divided into 4 phases:

1. Resource Aggregation (See Section 4.1)
2. Abstract Syntax Generation (See Section 4.2)
3. Concrete Syntax and Editor Generation (See Section 4.4)
4. Product Configuration (See Section 4.5)

In the first phase a build order has to be determined. It is basically done by applying a topological sort based on dependency to all MGLs defined in the CPD. When the build order is established, phases two and three will be executed by performing the following steps for each MGL file:

- At first, a set of Ecore models is generated. These created Ecore models themselves depend on an Ecore model that describes the basic graph structure and the substantial operations which every editor based on these Ecore models has to provide.
- Alongside the generation of the Ecore model, a generation of the CINCO *transformation API* (CT-API) is carried out. The *CT-API* provides domain-specific operations

that respect the constraints formalized in the MGL file (e.g., typed getters for containers based on containment constraints). These generated operations are added to the Ecore models, before the code generation from these Ecore models to Java classes is executed by the EMF code generation facilities.

- The next steps perform the generation of the editors. The default CINCO product provides two editors a graphical Graphiti based editor and a textual Xtext-based editor (called *GraText*).
- After the generation of the editors, the execution of meta plug-ins is triggered. Meta plug-ins provide functionality for the models (e.g., code generation or interpretation, model transformation). Since meta plug-ins are added to the CINCO environment (the meta-layer) to provide functionality in the generated CINCO product, we call them meta plug-ins to distinguish them from standard Eclipse plug-ins.

After phases 2 and 3 have been completed and the code has been generated for every MGL, the product generation is concluded in phase 4. This phase makes use of the information present in the CPD file starting with the execution of used CPD meta plug-ins. After this two new projects are created that are essential for every product: the *feature* project and the *product* project. In the following, the four phases and their corresponding tasks will be detailed.

4.1 Resource Aggregation

In a bigger CINCO product with multiple MGL files, it is often necessary to have relationships between them, which can be defined using the prime-reference feature introduced on page 17. For example Figure 4.2, for example, shows the dependencies between the three main model types of DIME [BFK+16]. The ***Data*** model type has no dependencies to other model types. But it can be used by them. A model of the ***GUI*** model type may display data, so the model type depends on the ***Data*** model type. ***Process*** models have two dependencies. On the one hand, they may read and write data so the ***Process*** model type depends on the ***Data*** model type. But ***Process*** models may also reference ***GUI*** models which leads to a dependency on the ***GUI*** model type.

As seen in Section 3.1.1, MGL files can import other MGL files by using the `import` statement. Model elements may then reference other model elements from these imported MGLs using prime references. In CINCO, the generated artifacts from the imported MGLs need to be present when building the referencing MGLs. CINCO achieves this by establishing a topological sort on the MGLs of a product. Resource aggregation also has a second use: During tool development it is often necessary to rebuild the modeling tool. Rebuilding all dependencies might take a long time. To reduce this building time, unmodified and already generated MGL files are taken out of the sort, so only new changes need to be considered.

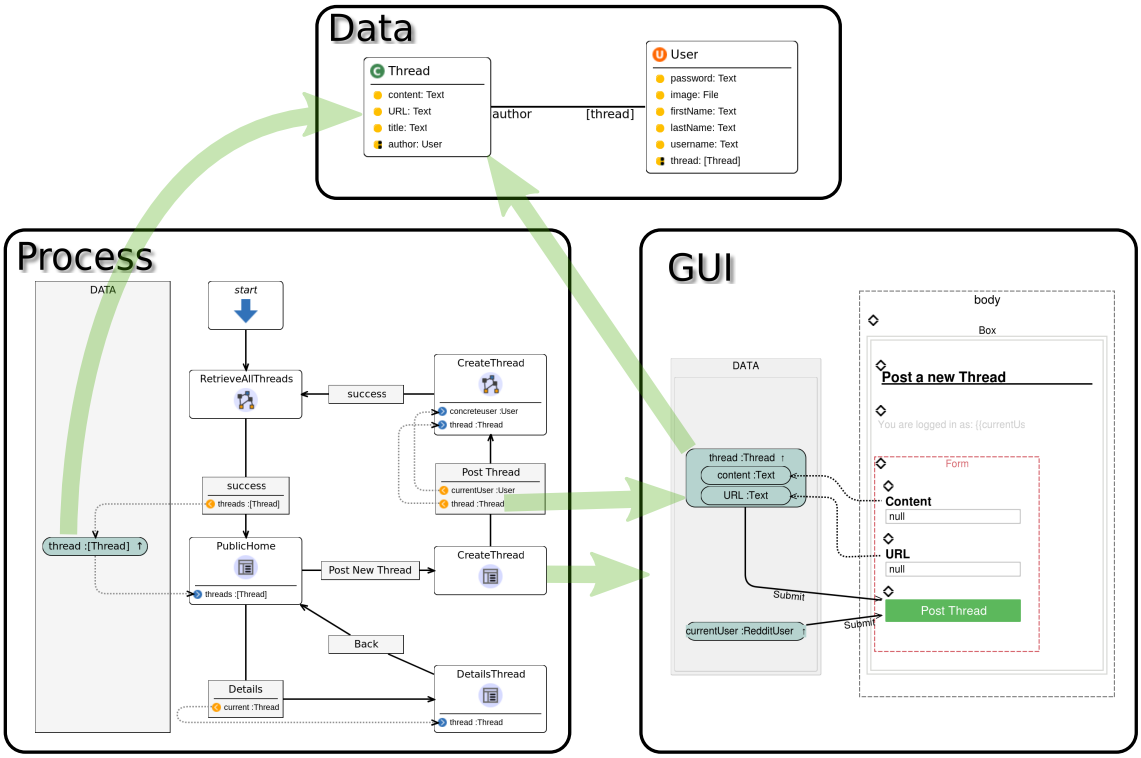


Figure 4.2: Model types and inter-model dependencies in DIME [BFK+16] (reprinted from [NLKS17])

4.2 Abstract Syntax Generation

In the CINCO Product generation process, several steps need to be taken for every single MGL file. CINCO uses the meta meta-model Ecore to describe the class structure of each generated language. The runtime environment of the generated language is divided into generated and predefined parts. The first step for every MGL file is to map the abstract syntax described in the MGL to the basic Ecore class structure. CINCO provides an Ecore model for basic graph structures, such as `Node`, `Edge` and `Container` elements, from which the generated classes inherit default properties (e.g. `x` and `y` positions for ***Node***s and ***Containers***). The basic Ecore model separates the inheritance structure and the data management of the generated model elements. API calls are made on the classes holding the inheritance structure. This separation is done to abstract from internal operations and to provide a clean API for developers and to separate internal functions from it. Therefore, the basic Ecore model is divided into two packages: *graphmodel* package and the sub package *internal*. The *graphmodel* package contains all basic types for model elements and some helper classes.

Basic CINCO Classes Figures A.2 and A.3 in the appendix, show the basic Ecore model every graph in an CINCO product uses. Every element in the basic Ecore model inherits the properties of the ***IdentifiableElement*** class. This takes care that every element can

be identified with a UUID. Every ***IdentifiableElement*** is either a ***Type***, a ***ModelElement***, or a ***ModelElementContainer***. ***ModelElement***s are the visible parts of a graph model. ***ModelElementContainer***s are elements that can contain ***ModelElement***s being it ***Node***, ***Edge*** or ***Container*** elements.

The GraphModel is the base element of a graph model in a CINCO product. Every model element, except the graph model, has to be contained in another model element. A model element can be contained in the graph model itself, or an element that is contained in a graph model (e.g., a container). It is also possible that the contained element itself is a container, which then also can contain elements. The ***GraphModel*** class corresponds directly to the defined *GraphModel* type of the MGL model. Every ***GraphModel*** is also a ***ModelElementContainer***, which allows for the containment of model elements.

The Node class describes the nodes of a graph model. Every ***Node*** is also a ***ModelElement***. For every *Node* type defined in the MGL, a class is generated that inherits all properties of the ***Node***.

The Edge class describes the edges of a graph model. It corresponds directly to the *Edge* type of the MGL. Every generated edge type defined in the MGL, inherits all properties of ***Edge***. Every ***Edge*** is also a ***ModelElement***.

The Container class describes the containers of a graph model. Every generated container type defined in the MGL, inherits all properties of ***Container***. Every ***Container*** is also a ***ModelElement*** and also a ***ModelElementContainer***.

The Type class describes user-defined types that are not visible in the graph editor but can be referenced by ***ModelElement***s or the ***GraphModel*** and may be edited for example using the generated properties view of the generated *Graphiti* editor. The class corresponds directly to the *UserDefinedType* types defined in the MGL model. The defined ***Enumerations*** are generated directly into corresponding Ecore *Enumeration* types.

For each model element defined in the MGL, two classes are generated: an internal class for the data and a visible class that contains the inheritance and an API to manipulate the data stored in the internal class. In addition to that for each hook-annotation (see Section 3.1.2) defined in the MGL, a method is generated that calls the defined hook.

Figure 4.3 shows the generated classes for two defined node types and the connection to the basic CINCO Ecore model. Classes ***A*** and ***B*** are the API classes. ***InternalA*** and ***InternalB*** are the data holding internal classes. ***B*** holds an attribute `t` of type ***T*** defined as an *UserDefinedType*, so methods `getT():T` and `setT(t:T)` are generated in the API class ***B***.

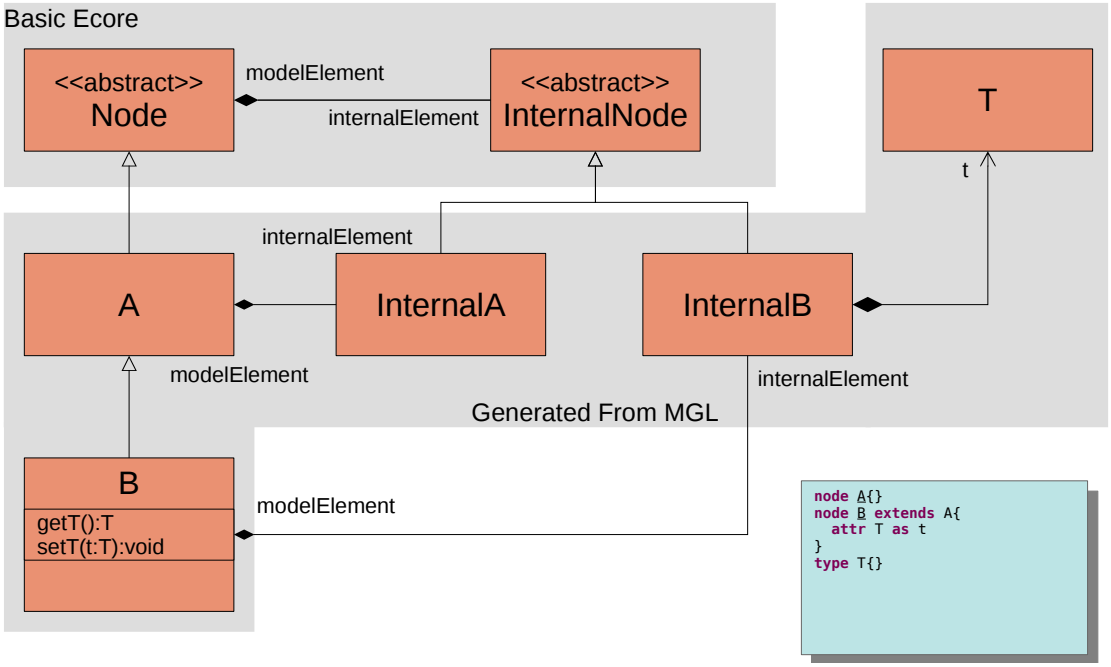


Figure 4.3: Example for generated Class Structure

4.3 CINCO API Generation

The *CPGP* does not only generate the necessary classes for a tool, but it also generates an API (the *CT-API*) that allows developers to program additional functionality for the CINCO product, such as code generation, model transformation. The API is generated to assist writing hooks and plug-ins for the CINCO product with domain knowledge. The API is constructed in a way that a developer does not need to know the internal work-flow of the tool, but can use the knowledge that was defined in the MGL specification. Therefore, for each attribute, a set of setter and getter methods is generated, that are specifically typed. These methods, that are specifically generated, are methods that need to satisfy the *ConnectionConstraints* and *ContainmentConstraints* defined in the MGL. Since all model types and constraints are known at generation time, the method signatures can be generated as specific as possible. It is important to notice that changes in the sub-type hierarchy may also affect the method signatures of parent types. This of course, breaks with the common understanding of inheritance in object oriented languages, but is necessary to get the most specific return types. The trade-off here is that a once generated tool may not be extended, which interferes with possibly wanted reuse (see Chapter 5).

4.3.1 Containment Constraints

Every generated ***Container*** class has a method *getModelElements*. It returns a list of nodes and containers that are contained. The return type of this method is the most specific type that still covers every node and container type any sub-type of the container

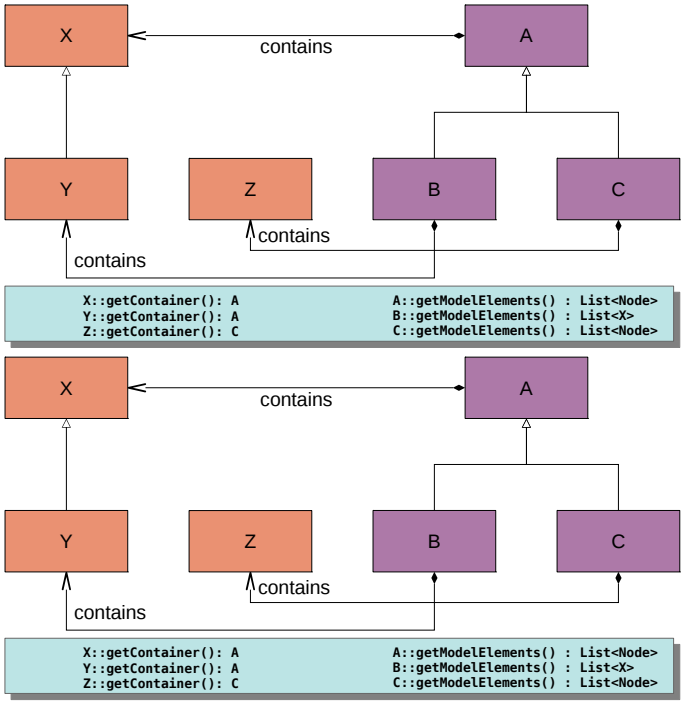


Figure 4.4: Effect on `getContainer`/`getModelElements` types through inheritance

may contain. The worst case is that the most specific type is ***Node***, since every element type that may be contained is at least this. Every generated ***Node*** does also have a *getContainer* method which returns the container the ***Node*** is currently held in, with its return type being the most specific type of container that may contain this node. In the worst case this may be ***ModelElementContainer***. Figure 4.4 shows an example for the `getModelElements` and `getContainer` methods as generated by the containment constraints and how a change in the inheritance can affect the method types. Image (1) of this figure reproduces the example of Figure 3.3 (see page 19). *Container* type ***A*** can contain any *Node* type that is of type *X*. But since type ***C*** inherits from ***A*** and can contain nodes of type ***Z*** which does not inherit from *X*, the best specific type is *Node*. So, the return type of the *getModelElements* method is `List<Node>`. The return type of ***B***'s *getModelElements* method is not affected by ***C***, so its return type is `List<X>`. When the inheritance hierarchy is changed, so that ***Z*** inherits from ***X*** (see Image (2)), the *getModelElements* methods of ***A*** and ***C*** are changed, so that the return type is `List<X>`. This also affects the *getContainer* method of the *Node* type ***Z***. In Image (1), ***Z*** is not part of an inheritance hierarchy. The only containment constraint using ***Z*** is that of container ***C***. So, the return type of the ***Z***'s *getContainer* method is ***C***. Adding ***X*** as parent to ***Z*** in Image (2) changes this. Now, every container that inherits from ***A*** may contain ***Z*** so the return type of the *getContainer* method changes to *A*. Note, that in Image (2), the containment constraints of ***B*** and ***C*** are obsolete, because of the containment constraint of ***A***.

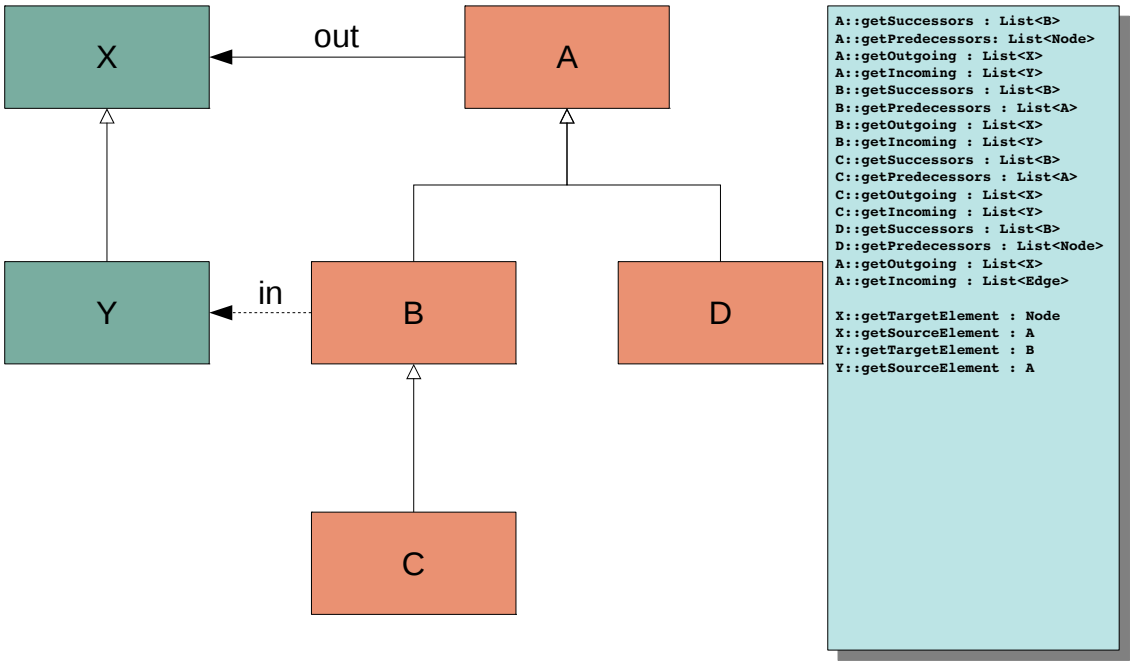


Figure 4.5: Example for generated connection methods

4.3.2 Connection Constraints

Analog to the containment constraints, connection constraints affect methods that base on edge connections. Every node and container has the `getSuccessors` method. It returns a list of nodes (including containers, which are special nodes) that are direct successors. To determine the return type of this method, not only possible outgoing edges (defined by the `outgoingEdges` constraints), but also the nodes that may have these edges as incoming edges have to be taken into account. Identical to this is the generation of the `getPredecessor` methods. Other methods are the `getOutgoing` and `getIncoming` methods, which return edges which start or end in the owning node, or the `getSourceElement` and `getTargetElement` methods, which return for an edge in which node it starts or ends. Figure 4.5 shows an example of methods and their return types for nodes ***A***, ***B***, ***C*** and ***D***, and the edge types ***X*** and ***Y***. Node type ***A*** has a connection constraint for outgoing edges. This constraint states that ***A*** may have outgoing edges of type ***X***. ***B*** and ***D*** are sub-types of ***A*** which means they inherit the connection constraint. ***B*** also has an incoming constraint with the edge type ***Y***. This incoming constraint is inherited by node type ***C***. Analog to the example in Section 3.1.1, this means, that possible starting points for edges are the node types ***A***, ***B***, ***C*** and ***D***. Possible end points are ***B*** and ***C***. So, according to this, the `getPredecessor` methods of node type ***B*** and ***C*** have the return type `List<A>` and the `getSuccessor` methods of node types ***A***, ***B***, ***C***, and ***D*** have the return type `List<B>`.

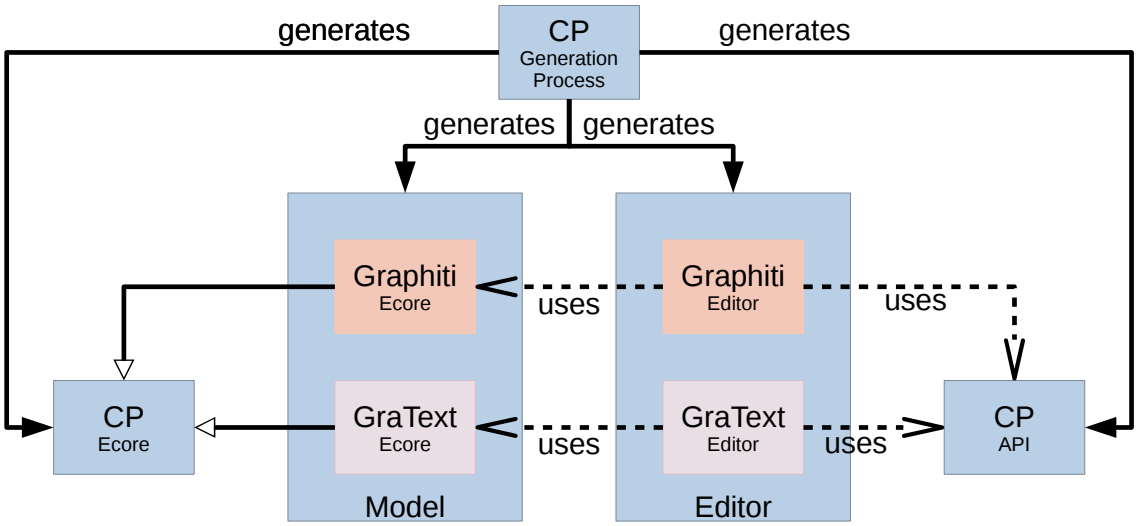


Figure 4.6: Relation between Editors and Models

4.4 Concrete Syntax and Editor Generation

A CINCO product usually consists of two editors for each language, a graphical and a textual editor[1]. The graphical editor is based on the Graphiti framework [6], while the textual editor uses a special Xtext [16] grammar, the *GraText*, which itself is generated by the *CPGP*. The *GraText* Editor is purely optional, but will be used as the serialization format when generated. The editors have in common that they both use specially generated Ecore models that use the Ecore model of the abstract syntax and the generated CINCO API to implement the editor functionality. For each model element, the required features (see Section 2.2.3) are generated. These are: *Add Features*, *Create Features*, *Update Features*, and *Delete Features*. In addition to these features, *Node* and *Container* types have *Move Features*, *Resize Features* and *Layout Features*, while *Edge* types have *Reconnect Features*. Figure 4.6 shows an overview of the editor generation. The *CPGP* generates the Ecore models and the API for the CINCO *Product* (CP) in phase 2. In phase 3 of the *CPGP*, for each editor that is generated, another Ecore model is created where each model element in the CP Ecore has a counterpart in the editor's Ecore model, which inherits all properties from the CP model element, but also adds information needed for editor functionality. Figure 4.7 shows the connection between the three Ecore models using the simple *Node* ***B*** from Figure 4.3 as an example. The Graphiti editor works on model elements that contain graphical information. For every model element defined in the MGL, a class prefixed with *C* is generated (in this example ***CB***). This class inherits from their base classes. When a tool developer uses the *CT-API* to manipulate the model, the Graphiti editor will be notified of any changes. The *GraText* editor uses the internal classes of the class structure to serialize the model, when saving. Since all data is stored in the internal model, only

[1]Other editors are imaginable. For example, the Pyro [Zwe15] plug-in generates web-based editors from the same three languages as the *CPGP*.

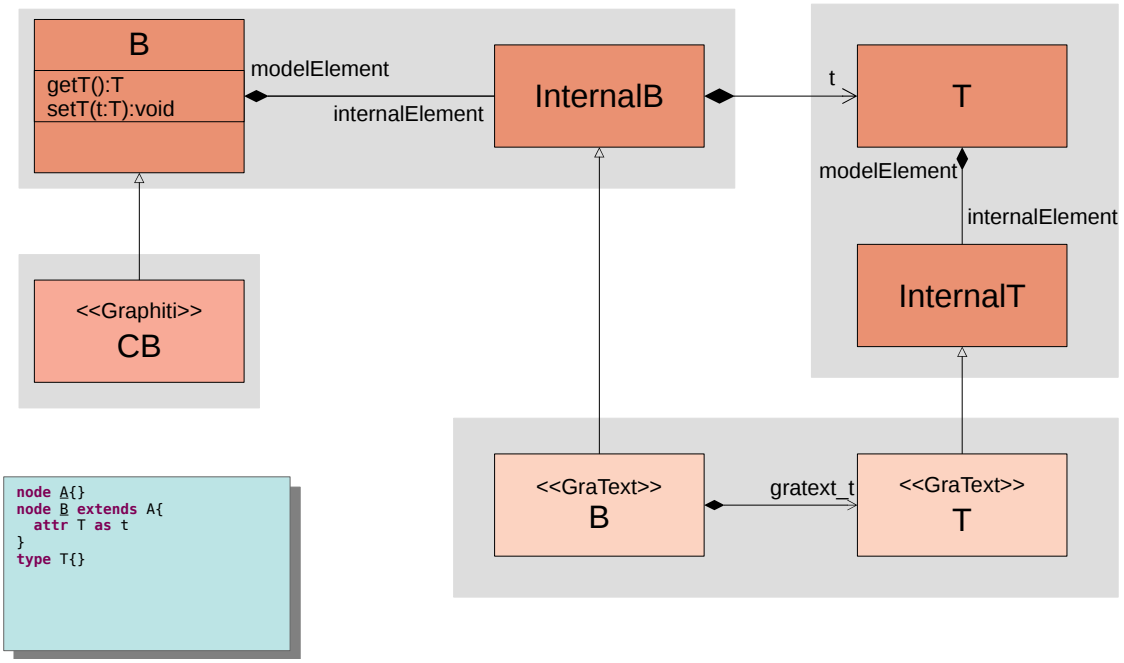


Figure 4.7: Connection between generated Classes for *GraText* and Graphiti editor

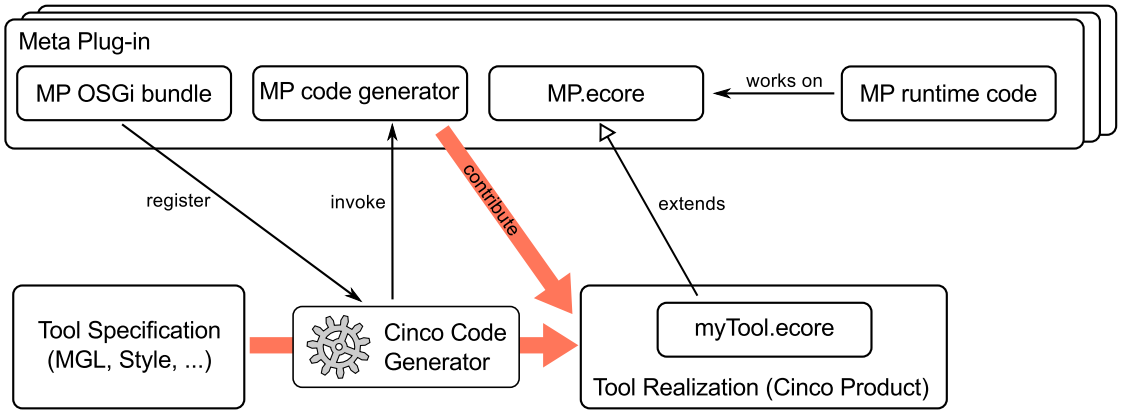


Figure 4.8: Overview of the CINCO Meta Plug-in Architecture (reprinted from [NLKS17])

internal objects need to be saved. The API structure will be recreated when loading the model. The *GraText* editor links the Graphiti classes by generating a counter part in the *GraText* Ecore model for every internal class and allows that changes made in the textual view are visible in the graphical view.

Meta Plug-in Activation

The usage of meta plug-ins is declared in the MGL using annotations (see Section 3.1.2) that are added to model elements. The annotations can be divided into annotations that can be added to model elements or to attributes of model elements. A meta plug-in must have at least one annotation that is added to the `graphmodel` declaration that globally

activate the meta plug-in. Each meta plug-in is identified by the name of this annotation. The annotation may have parameters and may also depend on other annotations added to other model elements or attributes. During the generation process, the meta plug-ins identified by these annotations are activated. To be able to activate the meta plug-ins, they need to be present in the *meta plug-in Registry* of the CINCO product generator. Some meta plug-ins are already provided by the CINCO developer, but new meta plug-ins may also be added. To register a new meta plug-in, few things are needed. First, the meta plug-in has to be implemented as a bundle communicating with the registry, which registers the used annotations, and defines the class that contains the logic of the meta plug-in (e.g. necessary code generators). Using the annotations, the meta plug-in can be invoked during the *CPGP* (See Figure 4.8).

4.5 CINCO Product Configuration

Once abstract and concrete syntax for all MGLs is generated, the last phase of the *CPGP* is the CINCO product configuration. It starts with the activation of CPD meta plug-ins and finishes with the generation of the feature and product projects, which are needed to bundle together everything that is needed to have a complete running bundle.

4.5.1 CPD meta plug-ins

The activation of a CPD meta plug-in behaves in basically the same way as the standard MGL meta plug-in. But instead of being added to the MGL file, a CPD meta plug-in is added to the product description and is used for meta plug-ins that affect the whole CINCO product and not only one MGL.

4.5.2 Product and Feature Project

The finishing steps of the *CPGP* are the bundling of all generated and otherwise needed code, and adding the tool branding. This is done by generating a product plug-in that uses the branding information defined in the CPD and including a feature plug-in. Products in Eclipse can be either based on plug-ins or features. A feature is a set of plug-ins which together serve a specific purpose. It can also depend on other features. The *CPGP* generates a feature project that contains all plug-ins generated during the generation process, basic features and plug-ins needed to have a executable product, and all plug-ins and features as additional dependencies which are defined in the CPD.

Chapter 5

Conclusions and Future Work

This dissertation has shown a technological view on the CINCO meta-tooling suite. While other publications [NLKS17, SGNM19, SN16] have focused on the concepts, this dissertation has presented an approach to generate domain specific tools, the CINCO product generation process, and has also presented the generation of the CINCO transformation API. Several modeling tools were created on this basis. Some of them were part of this dissertation [LKS18, BFK+16], several more created by other student projects, research projects [KLNS19], or for use in the industry (for a small overview, see Figure 5.1).

From the many different projects realized with CINCO, of course, many requests for additional features have arisen. Examples for often requested features are the introduction of reuse in the MGL, or model-driven code generation.

5.1 Improving the Reuse of Model Elements

At the moment, there is no reuse of written MGLs, since previously defined model elements, such as nodes, containers, and edges, can not be imported into new MGLs. As mentioned in Section 4.3, the CT-API depends on the complete knowledge of all defined types at generation time. This hinders the reuse of defined types. For example, the ***GUI*** model and the ***Process*** model of the DIME project both define the concept of a ***DataContext***. The ***DataContext*** in both models is a container that can contain ***Variable***s. And these ***Variable***s may be written or read (modeled by ***DataFlow***) edges. In both models, these elements need to be defined individually, even though their functionality is almost identical. Future work will consist of inspections to find a serviceable way to define language libraries or to extend existing tool descriptions. A possible solution may be the introduction of special keywords in the MGL that define how model elements may or may not be reused:

`final` A model element defined as `final` may not be reused. The generated API can be as specific as possible.

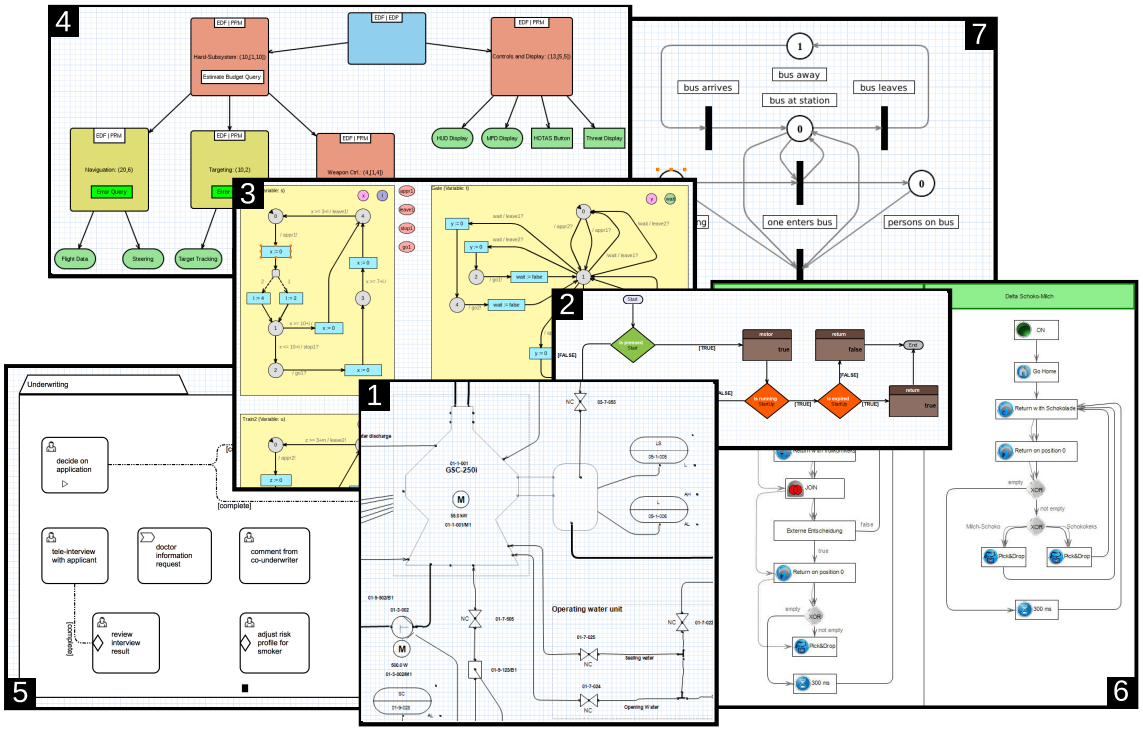


Figure 5.1: Further examples for CINCO applications: (1) Piping & Instrumentation Diagram [WMN16] (2) Flow Graph [WMN16] (3) Probabilistic Timed Automata [NTI+14] (4) Hierarchical Scheduling Systems [CKL+17] (5) OMG's Case Management CMMN [Wec16] (6) EasyDelta Pick and Place DSL [BDG+15] (7) Place/Transition Net [NLKS17] (reprinted from [SGNM19])

`productFinal` An MGL is known to other MGLs in the same project. It may be used as a library inside the project (e.g. reusing elements of the ***DataContext*** defined in the ***Process*** model of DIME, in the ***GUI*** model). As a project is always built by the *CPGP*, the generated API can her also be as specific as possible.

`library` A Model element tagged as library can be used in any newly defined tool. The API generation for this element will be generated without the knowledge which elements subclass this element. Thus, the subclassing can only restrict containment and connection constraints in a classical OOP way.

Implementing this or other features will most certainly affect the *CPGP*. To better adapt to changes in the code generator, possible code generation approaches may need to be reevaluated.

5.2 Using Model-Driven Code Generation for the *CPGP*

The template based code generation using Xtend has several advantages over simple template engines, which can also be found in model-driven approaches such as Genesys. For example, both approaches allow the separation of generator logic from output creation. While it is possible in Xtend to add complete programming expressions inside the tem-

plate expressions, this should be avoided. Instead, the template expressions should be placed inside separate methods or even classes. The templates can be parameterized with method arguments filling in placeholders. Both approaches also allow separation of concerns, either by using hierarchy through sub-models (Genesys), or by applying concepts of object-oriented, or even functional languages (Xtend). These aspects of the code generation frameworks can be used for the *CPGP* to lessen the impact that the complexity of the generation process has. Another important factor is the ability to adapt to changes. For example, CINCO is often improved by adding new features which also affect the *CPGP*. Code generators implemented using the Genesys approach have been proven to be easily to modify when new features are added or existing ones need to be changed. Using the default implementation of the Genesys approach however, is problematic due to the old jABC not being maintained anymore. It would be interesting to apply a new model-driven approach that combines concepts from Xtend and Genesys into a new CINCO-based code generation framework via service-orientation [JS11]. The student project *nextGen* [BCK+19], which I am currently co-supervising with my colleague Dominic Wirkner, has created a new approach for creating graphical code generators from CINCO specifications. With combination of *nextGen* and the *Graphical* CINCO *Specification* (GCS) [Fuh18], the possibilities to use graphical modeling for at least parts of the *CPGP* should be explored.

Bibliography

[AKS+18] Goddy Ntongwe Asale, Oliver Köhler, Kristina Sax, Alexander Kedzia, Sascha Mücke, Oliver Scherf, Jens Knipper, Richard Niland, and Daniel Scholtyssek. Projektgruppe 610: An Engine for Generative Game Development. Technical report, TU Dortmund, 2018.

[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. Projektgruppe 617: NextGen Model-based Code Generation. Technical report, TU Dortmund, 2019. To Appear.

[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.

[BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[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.

[FFF+18] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A Pro-

grammable Programming Language. *Communications of the ACM*, 61(3):62–71, mar 2018.

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

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

[Gho11] Debasish Ghosh. Dsl for the uninitiated. *Commun. ACM*, 54(7):44–50, July 2011.

[Ham75] Michael Hammer. The design of usable programming languages. In *Proceedings of the 1975 Annual Conference*, ACM '75, pages 225–229, New York, NY, USA, 1975. ACM.

[HLR08] Thomas Hettel, Michael Lawley, and Kerry Raymond. Model Synchronisation: Definitions for Round-Trip Engineering. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ICMT '08, pages 31–45, Berlin, Heidelberg, 2008. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[JMS08] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. Genesys: service-oriented construction of property conform code generators. *Innovations in Systems and Software Engineering*, 4(4):361–384, 2008.

[JS11] Sven Jörges and Bernhard Steffen. Leveraging service-orientation for combining code generation frameworks. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 198–207. IEEE, 2011.

[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.

[Jö13] Sven Jörges. *Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach*, volume 7747 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Germany, 2013.

[Kah62] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008.

[KLNS19] Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, and Bernhard Steffen. Towards Language-to-Language Transformation. 2019. to appear.

[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.

[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.

[Lyb12] Michael Lybecait. Entwicklung und Implementierung eines Frameworks zur grafischen Modellierung von Modelltransformationen auf Basis von EMF-Metamodellen und Genesys. diploma thesis, TU Dortmund, 2012.

[MLA10] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform*. Addison-Wesley Professional, 2nd edition, 2010.

[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.

[MS12] Tiziana Margaria and Bernhard Steffen. Service-Orientation: Conquering Complexity with XMDD. In Mike Hinchey and Lorcan Coyle, editors, *Conquering Complexity*, pages 217–236. Springer London, 2012.

[Nag09] Ralf Nagel. *Technische Herausforderungen modellgetriebener Beherrschung von Prozesslebenszyklen aus der Fachperspektive von der Anforderungsanalyse zur Realisierung*. Dissertation, Technische Universität Dortmund, July 2009.

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

[Neu14] Johannes Neubauer. *Higher-Order Process Engineering*. Phd thesis, Technische Universität Dortmund, 2014.

[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.

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

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[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. to appear.

[SN16] Bernhard Steffen and Stefan Naujokat. Archimedean Points: The Essence for Mastering Change. *LNCS Transactions on Foundations for Mastering Change (FoMaC)*, 1(1):22–46, 2016.

[STK+16] Alexander Schäferdiek, Benjamin Tokgöz, Fabian Kotschenreuter, Hang Yu, Jan Möller, Konstantin Tkachuk, Patrik Elfer, Ruikun Chang, and Volkan Gümüs. PG 592 - Planspiele für Krisenmanagement in Rechenzentren. Technical report, TU Dortmund, 2016.

[Sun13] KVN Sunitha. *Compiler construction*. Pearson Education India, 2013.

[SVEH07] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* dpunkt, Heidelberg, 2. edition, 2007.

[tBCSW18] Maurice H. ter Beek, Loek Cleophas, Ina Schaefer, and Bruce W. Watson. X-by-Construction. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*, pages 359–364, Cham, 2018. Springer International Publishing.

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

[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.

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

Web Resources

[1] ASP.NET - The ASP.NET Site. https://www.asp.net/. [Online; last accessed 07-January-2019].

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

[3] Eclipse Modeling Project. http://www.eclipse.org/modeling/. [Online; last accessed 08-February-2019].

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

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

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

[7] Handlebars.js : Minimal templating on steroids. http://handlebarsjs.com/. [Online; last accessed 07-January-2019].

[8] Java Expression Language. http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html. [Online; last accessed 08-February-2019.

[9] Java Platform, Standard Edition 8 API Specification. https://docs.oracle.com/javase/8/docs/api/overview-summary.html. [Online; last accessed 07-February-2019].

[10] {{mustache}} Logic-less templates. https://mustache.github.io/. [Online; last accessed 07-January-2019].

[11] PHP: Hypertext preprocessor. http://php.net/. [Online; last accessed 07-January-2019].

[12] Racket. https://racket-lang.org/. [Online; last accessed 08-February-2019].

[13] The Scala Programming Language. http://www.scala-lang.org. [Online; last accessed 08-February-2019].

[14] Twig - the flexible, fast, and secure template engine for php. https://twig.symfony.com/. [Online; last accessed 07-January-2019].

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

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

[17] XQuery 3.1: An XML Query Language. http://www.w3.org/TR/xquery/#EBNFNotation, 2017. [Online; last accessed 08-February-2019].

[18] Internet Engineering Task Force (IETF). The JavaScript Object Notation (JSON) Data Interchange Format. https://tools.ietf.org/html/rfc7159, 3 2014. [Online; last accessed 08-February-2019].

[19] Object Management Group (OMG). About the Unified Modeling Language Specification Version 2.5.1. https://www.omg.org/spec/UML/2.5.1/, dec 2017. [online; last accessed 15-March-2018].

Appendix A

Large Figures and Listings

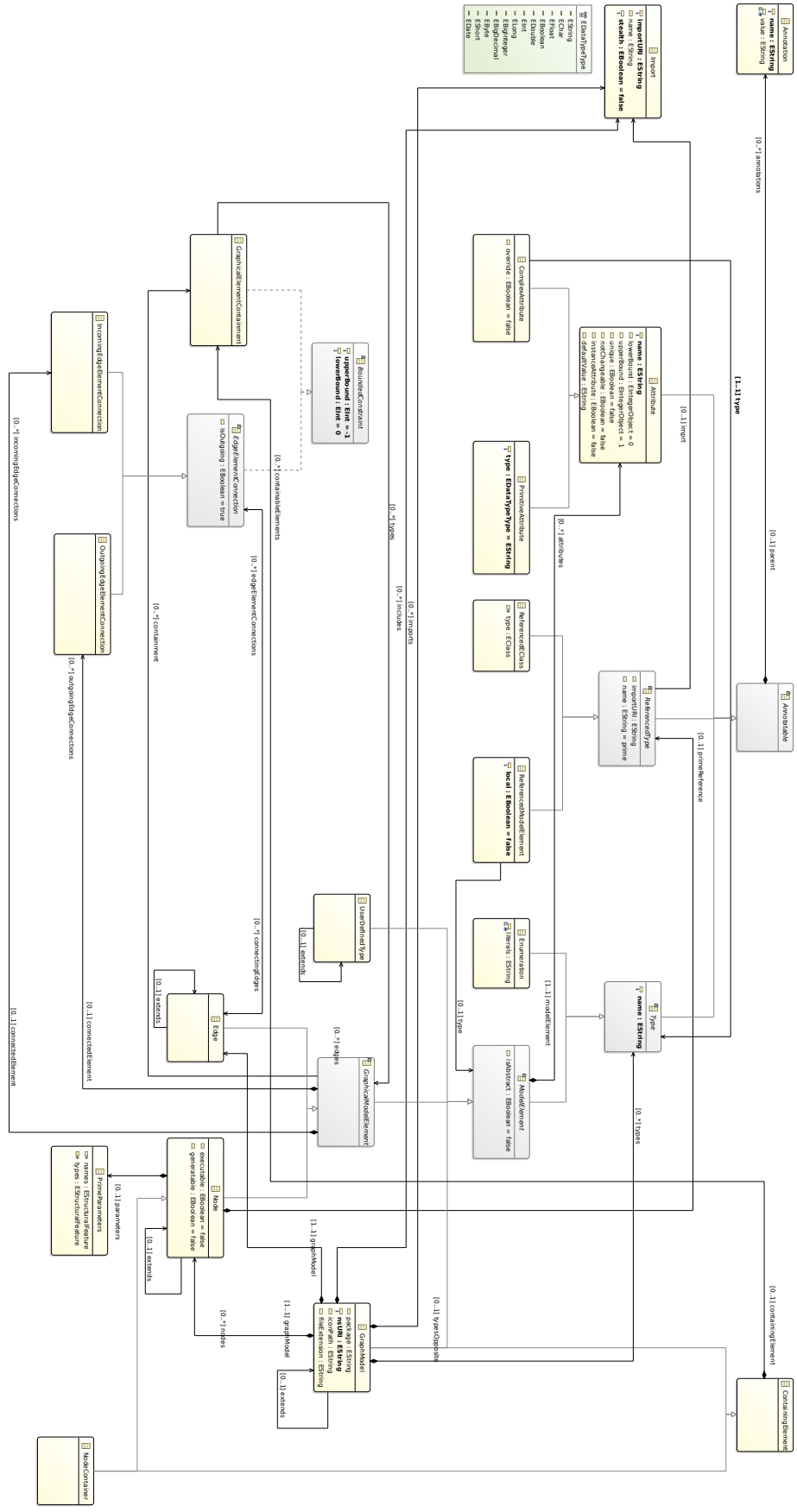


Figure A.1: Class Diagram of the Meta Graph Language

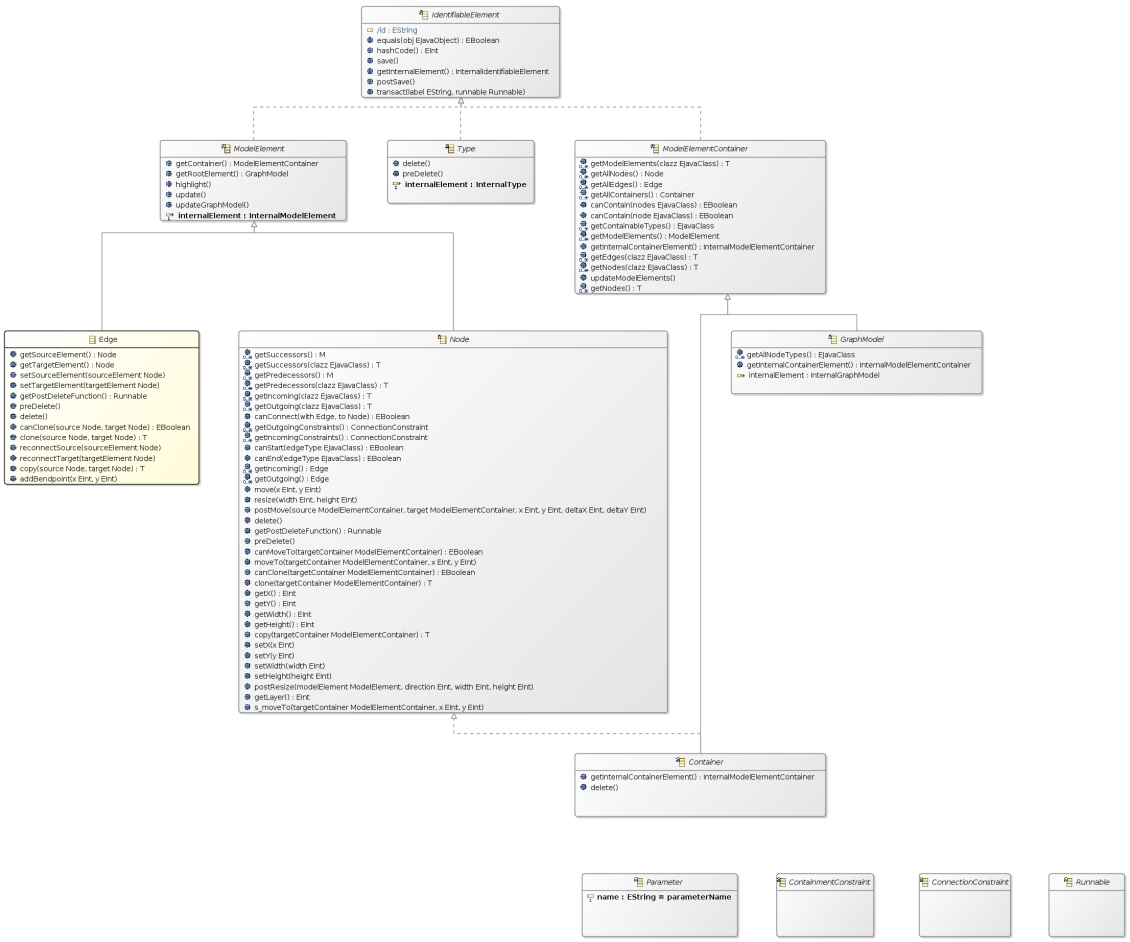


Figure A.2: Class Diagram of the Basic GraphModel

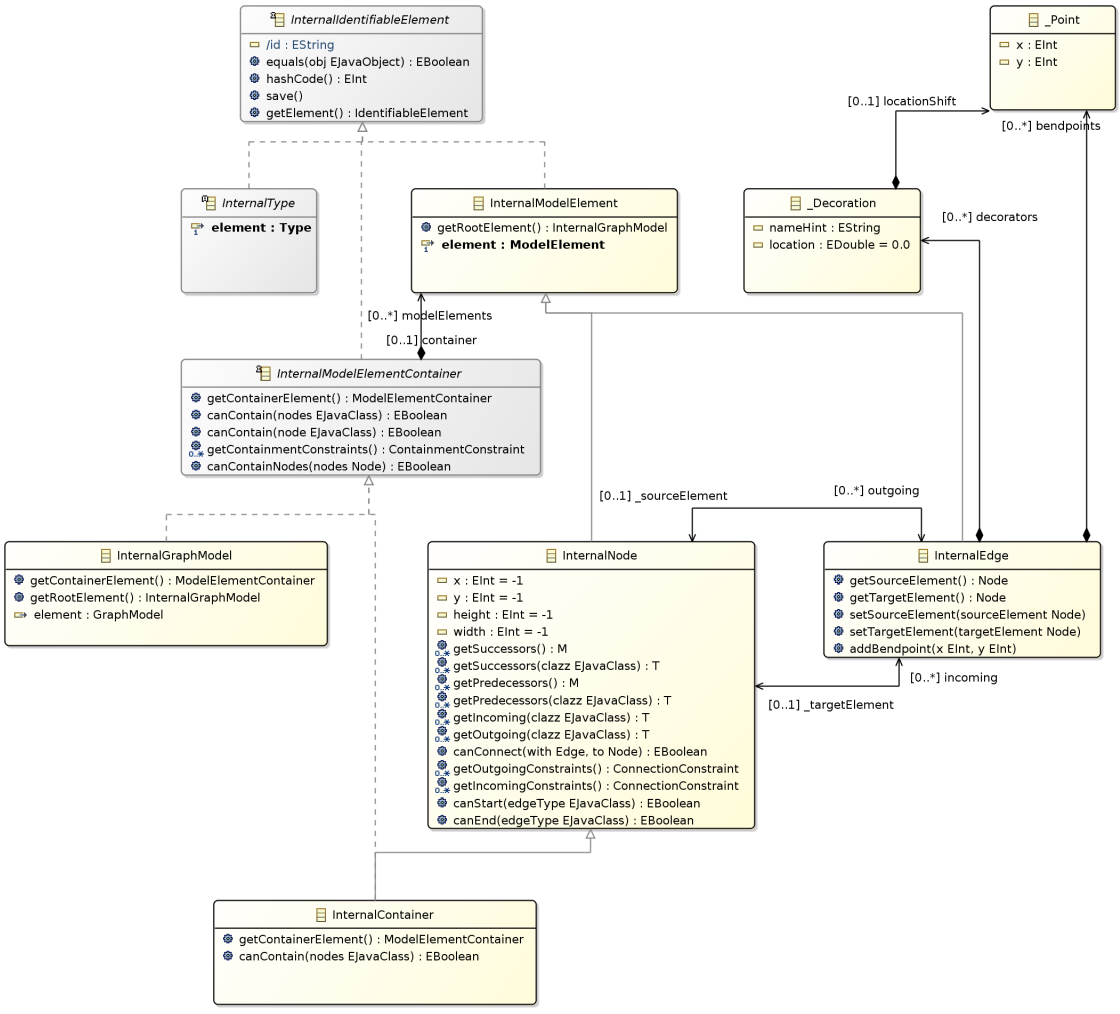


Figure A.3: Class Diagram of the Internal GraphModel

```
1  GraphModel ::=
2    Import*
3    Annotation*
4    'graphModel' EString ( 'extends' EString )?  '{'
5      ('package' QName )?
6      'nsURI' URI
7      ( 'iconPath' EString )?
8      'diagramExtension'  EString
9      ( 'containableElements' '(' GraphicalElementContainment ( ','
           GraphicalElementContainment )* ')' )?
10     Attribute*
11     ( Node | Edge | NodeContainer | Type )*
12   '}'
13 Attribute ::= PrimitiveAttribute | ComplexAttribute
14 PrimitiveAttribute ::=
15   Annotation*
16   'final'? 'unique'? 'attr' EDataTypeType 'as'  EString ( '[' EInt ( ',' BoundValue )
         ? ']' )? ( ':=' EString )?
17 ComplexAttribute ::=
18   Annotation*
19   'final'? 'unique'? 'override'? 'attr' EString 'as' EString ( '[' EInt ( ','
         BoundValue )? ']' )? ( ':=' EString )?
20 GraphicalModelElement ::= Edge | Node | NodeContainer
21 Edge ::=
22   Annotation*
23   'abstract'? 'edge' EString ( 'extends' EString )? ( '{' Attribute* '}' )?
24 Node ::=
25   Annotation* 'abstract'?
26   'node' EString ( 'extends' EString )? '{'
27     ( Attribute*
28       | ReferencedEClass
29       | ReferencedModelElement
30       | ( 'incomingEdges' '(' IncomingEdgeElementConnection ( ','
              IncomingEdgeElementConnection )*
31       |  'outgoingEdges' '(' OutgoingEdgeElementConnection ( ','
              OutgoingEdgeElementConnection )* ) ')' )*
32   '}'
33 Annotation ::= '@' EString ( '(' EString ( ',' EString )* ')' )?
34 NodeContainer ::=
35   Annotation*
36   'abstract'? 'container' EString ( 'extends' EString )? '{'
37     ( Attribute*
38     | ReferencedEClass
39     | ReferencedModelElement
40     | ( 'containableElements' '(' GraphicalElementContainment ( ','
           GraphicalElementContainment )*
41     | 'incomingEdges' '(' IncomingEdgeElementConnection ( ','
           IncomingEdgeElementConnection )*
42     | 'outgoingEdges' '(' OutgoingEdgeElementConnection ( ','
           OutgoingEdgeElementConnection )* ) ')' )*
43   '}'
```

Listing A.1: Extended Bacchus Naur Form of the MGL

```
44 OutgoingEdgeElementConnection ::= ( QName | '{' QName ( ',' QName )* '}' | '*' ) ( '[
       ' EInt ',' BoundValue ']' )?
45 IncomingEdgeElementConnection ::= ( QName | '{' QName ( ',' QName )* '}' | '*' ) ( '[
       ' EInt ',' BoundValue ']' )?
46 GraphicalElementContainment ::= ( QName | '{' QName ( ',' QName )* '}' | '*' ) ( '['
       EInt ',' BoundValue ']' )?
47 Import      ::= 'stealth'? 'import' STRING 'as' ID
48 BoundValue  ::= '*' | EInt
49 Enum ::= Annotation* 'enum' EString '{' EString+ '}'
50 Type ::= Enum | UserDefinedType
51 UserDefinedType ::= Annotation* 'abstract'? 'type' EString ( 'extends' EString )? '{'
        Attribute* '}'
52 ReferencedType ::= ReferencedEClass | ReferencedModelElement
53 ReferencedEClass ::= Annotation* 'prime' ID '.' ID 'as' EString
54 ReferencedModelElement ::= Annotation* 'prime' ( 'this' | QName ) '::' QName 'as'
       EString URI ::= EString
55 QName ::= ( ID | ANY_OTHER )+ ( '.' ( ID | ANY_OTHER )+ )*
56 EString ::= STRING | ID
57 EInt ::= '-'? INT
```

Listing A.2: Extended Bacchus Naur Form of the MGL cont.

Appendix B

Attached Papers

Parts portion of this dissertation have been published in cooperation with a number of co-authors. This section lists the publications with a classification of my contribution for each.

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

The presented concepts and technologies were discussed among all authors. Stefan Naujokat was main authors of all sections. The Cinco implementations were primarily done by Dawid Kopetzki and myself.

II Lybecait, Michael, Dawid Kopetzki, Philip Zweihoff, Annika Fuhge, Stefan Naujokat and Bernhard Steffen. **A Tutorial Introduction to Graphical Modeling and Meta-modeling 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 am the main author of sections 2 and 4. Implementation of the GCS tool was primarily done by Annika Fuhge and implementation of Pyro was done by Philip Zweihoff, both building on concepts and technologies developed by Stefan Naujokat, Dawid Kopetzki and myself.

III Lybecait, Michael, 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.

The presented concepts were discussed among all authors. I co-authored all sections and I am main author of sections 3, 4 and 6.

IV Bosselmann, Steve, 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.

The presented concepts and technologies were discussed among all authors. Implementations of Dime have been done primarily by Steve Boßelmann, Markus Frohme, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner and Philip Zweihoff. Enhancements in Cinco required for the implementation of Dime have been done primarily by Dawid Kopetzki and myself.