technische universität
dortmund

Master Thesis

Cinco Products for the web

Philip Zweihoff
November 2015

Supervisor:
Prof. Dr. Bernhard Steffen
Dipl.-Inf. Stefan Naujokat

Technical University of Dortmund
Department of Computer Science
Chair for Programming Systems (LS-5)
http://ls5-www.cs.tu-dortmund.de

Contents

Chapter 1

Introduction

The development of today's software systems, which use several different technologies, is a difficult and complex task. It is necessary to define a software with the close collaboration of a domain expert, who has the idea of the software and it's behavior and the developer, who tries to analyze, interpret the earned information and finally plan and realize it. This translation of domain specific ideas of a software to an abstract model, which can be understood by a developer, is a difficult matter to deal with.

To handle this problem, model-based development paradigms were introduced. These paradigms are based on the specification of an abstract model, which hides conceptual insignificant details from the developer. Depending on the manufactured model the software can be implemented. As an advantage of the model-based development and it's degree of abstraction the requirements are more precise, simple described and become less redundant. This leads to an accelerated realization and well understood domain specific concepts to the developers. To enhance the paradigms, modeling tools were established, which provide abilities to generate code from a given model. On the other hand of these model-based paradigms there are drawbacks like the high effort for the initial specification of a domain-model [10]. In addition to this it is necessary to implement further features and modifications for specialized requirements, which have to be reestablished to the superior model, so that the model and its code are synchronized. This additional effort leads to a disregarding of the model after its initial definition. Furthermore the tools for the model-driven development paradigms pursue a generic approach, what causes a complex usability and time-consuming development. As a result to the more important model-based approaches, it has become a crucial aspect for the development, that the domain-specific tools are easy to use with a large scale support.

A framework to realize this approach is Cinco , which most recent advantages are represented by the full generation of graphical modeling tools from high-level specifications with the Cinco *Service Centered Continuous Engineering* (SCCE) meta tooling suite [15]. Following the simplicity paradigm [12], it allows for the easy specification of a tool's model

and its semantics in terms of code generation as well as model transformation, evaluation and interpretation. This tool tries to facilitate the development of modeling tools, called CINCO Products, as easy as possible instead of complete generality. It is based on technologies like the *Graphiti diagram editor framework* [6] and combines the simplicity of the *jABC* [19] with the *Eclipse Modeling Project* [7]. As a design criterion CINCO restricts itself on domain models which can be represented as graphs, to be a simplicity-oriented framework without the ambition to be as generic as possible with enormous complexity. The model and its representation are specified in the *Meta Graph Language* (MGL) and the *Meta Style Language* (MSL) [9], which form the foundation of the generated domain specific tools.

Since CINCO provides the ability of creating CINCO Products it has become a platform to develop and run domain specific models. According to research report by *MarketsAndMarkets*, the Platform as a Service (PaaS) market is expected to raise [13]. The major forces driving the PaaS market are the key factors in application development process such as agility, scalability, limited needs for expertise, easy deployment, and reduced costs and development time [2]. For this reason this thesis concentrates on the development of a generator, which creates a *Rich Internet Application* (RIA) [5] called Pyro depending on a CINCO Product, so that it will be enabled as a platform as a service in the category of cloud computing services. The realization of the generator is based on a CINCO Meta Plugin which reads out the specified MGL and MSL of the meta model and transforms it into a rich internet application. Instead of generating a complete and native web application, Pyro is realized as a plugin for the *Dynamic Web-Application* (DyWA) [16], to benefit from its support for domain specific data and business process modeling. The DyWA accompanies the prototype-drive web-application development from the domain modeling through the development and deployment phase to the actual runtime and later product evolution [16].

The remainder of this thesis is structured as follows. The technologies used for the Pyro generator are described in chapter 2. Here, more relevant CINCO SCCE and DyWA as the source and the target of the generation are described in more detail than the underlying frameworks such as *JointJS* [4]. In chapter 3 the concept and development of the Pyro generator and its output the *Pyro Modeling Environment* (PME) are described. This includes the *Pyro Meta Plugin* (PMP) for CINCO which transforms the defined meta models of the MGL and MSL and generates the mandatory files for the Pyro Modeling Environment (PME). Chapter 4 gives an overview of the existing web-based modeling frameworks in the category of PaaS and Chapter 5 concludes this thesis.

Chapter 2

Preliminaries

This chapter conveys the necessary fundamentals to retrace the ideas, concepts and chosen realization.

First, this comprises the explanation of the two meta layers which are provided by the CINCO SCCE meta tooling suite and its core features based on the relevance for the Pyro Modeling Environment. The core features will be explained to get an idea of the utilization of CINCO and to distinguish which functionalities need to be represented by the PME. In addition to this it will be on display which information will be read out of the CINCO high-level specifications for the generation. This includes the explanation of a model transformation, which emanates from the Model Graph Language and the Model Style Language to a ready-to-run modeling tool. CINCO provides much more features and is still under development, so that a complete summary of all technical details and facilities go beyond the scope of this thesis.

The second section of this chapter describes and pictures the dynamic Web-Application DyWA which will be used as a container to run the PME. As a result of this the generated PME has to be capable of being adapted to the DyWA and its interfaces. The interfaces will be illustrated and explained using the example of the data structure representation of a hotel administration system. With the aid of this example the abilities of the DyWA, relating to the meta and instance data management and domain-specific code generation, will be pointed out.

2.1 CINCO SCCE meta tooling suite

This section shows how to specify a CINCO *Product* (CP) with the given high level specification Meta Graph Language (MGL) and Meta Style Language (MSL). To ensure the comprehensibility of these languages, a *Mealy Machine* [14] CINCO Product will be exemplified. Depending on the MGL and MSL the domain specific CINCO Product and multiple

features will be generated. In addition to this Cinco provides the abilities to annotate the defined components of the MGL to enable Meta Plugins and expand functionalities.

First, the two high level specification languages will be explained to illustrate the scope of a domain expert. Afterwards the generated features like the transformation API and the main meta plugins of Cinco will be described. At Last the Cinco Product itself will be presented and amplified to demonstrate the usability.

2.1.1 Meta Graph Language

The Meta Graph Language defines the meta model of the tool a developer wants to create. This means, that the MGL model defines which modeling components will be available in the generated Cinco Product tool. To define a meta model, Cinco facilitates multiple types based on the underlying graph structured modeling types. The dependencies, relationships and the inheritance hierarchies of the MGL metamodel are represented in figure 2.1. The types of the MGL metamodel represent all supported components which can be used by the developer. During the creation of a MGL, Cinco instantiates the underlying meta model types. This instances will be used for successive model transformations. The relation between the MGL metamodel and its instances, which are addicted to a developed MGL, will be explained later using an example. The definition of a MGL begins with the `GraphModel` and the components which can be contained. The supported types which can be contained are `Node`, `Edge` and `Container`. Each component and the graphmodel itself has to obtain a name.

Every component which is a specializations of the `ModelElement`, including the graphmodel itself, can be extended by additional attributes. The attribute declaration is similar to known programming languages with a name and a data type. Cinco supports the common primitive data types of the *Eclipse Modeling Framework* (EMF) [20] like integers, floats, enumerations, etc. and unsorted lists of these types. EMF serves the developer with the chance to define a cardinality, which restricts the minimal and maximal size of a list. In addition to this Cinco provides the ability to the developers to define their own user defined types, which will be instances of the `UserDefinedType` of the MGL metamodel. A user defined type resembles the complex data type `struct` established in *C* [21], which allows to combine many complex and simple data types and access them together. The user defined types can be aggregated to a component by an attribute. As a further feature to support the *DRY-Principle* [22], code reuse and according to the *OOP programming paradigm* [18], the components can be specialized by inheritance with abstract components.

Cinco provides fundamental possibilities to define constraints on the model, to ensure no misuse of the generated Cinco Product tool and support the end user. This constraints can be used to codify which nodes may be connected, which edges are suitable and how many links can be established. To describe this constraints, the node components are

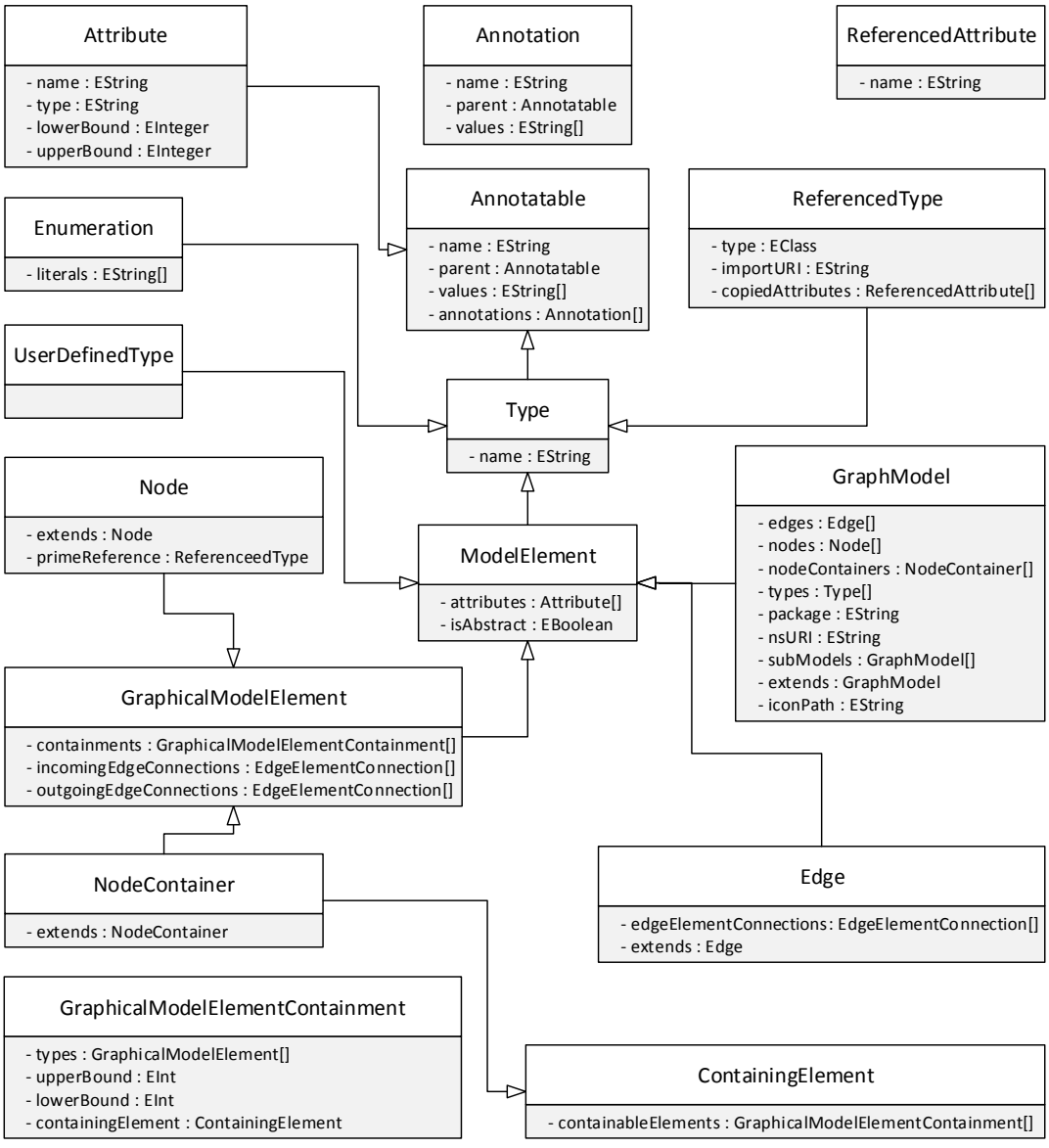


Figure 2.1: The meta model for the Model Graph Language. `MGL.ecore`

capable of being extended by a list of possible incoming and outgoing edges with cardinalities over subsets of these edges. Analogue to the connecting constraints, CINCO offers the facility to assess embedding constraints, which specify the components and its quantity of being embedded in a container or the graphmodel.

To extend to functionality and add further meta data to the components and its attributes CINCO provides annotations. The annotations, which are identifiable by a preceding at-sign, can be added to predefined types of the MGL metamodel like attributes, nodes, edges, containers and all other types which extend the `Annotatable` type. Furthermore arguments can be passed to an annotation to parametrize its behavior. The only required annotation is the `@style` annotation. The style annotation binds a defined style type, contained in a referenced MSL, by its name to the annotated component. As a result of that the visual representation of this component is defined. Additional arguments can be added to the annotation, to pass attributes of the component to the style. Afterwards the values of the attributes can be displayed. The other annotations are optional like the `@customFeature`, which adds a context menu entry to the related component in the CINCO Product. This entry triggers a method call of the predefined class referenced in the annotation's argument. This so-called *CustomFeatures* enable the developer to manipulate the model.

To illustrate the utilization of CINCO and its MGL a simple mealy machine will be exemplified. The associated MGL file is viewable in listing 2.1. First, the graphmodel is defined as `MealyMachine` with additional attributes like the `package`, an `URI` and further optional properties. A mealy machine requires an input string and depending on that generates an output char sequence. This two sequences and a reference to the current state of the automaton are stored in attributes contained in the mealy machine graphmodel. In this example the graphmodel contains no embedding constraints, so that every defined component can added often at will to the *MealyMachine*. The semantics determine that every node is capable of unlimited incoming and outgoing edges, including the self connection. As a result of this both connection constraints are determined to the `Transition` edge type with no limitations on the cardinalities. This MGL describes the basic model to realize simple mealy machines. The state nodes are distinguishable by its label, which is defined in the value of the `name` attribute of the component. The state diagram for a mealy machine associates an output value with each transition edge. As a result of this, the edges are extended with two attributes to define the input character and the produced output when the edge gets triggered. The created component of the `mealyMachine.mgl` will be realized as instances of the MGL metamodel, defined in the `MGL.ecore`. Depending on the created domain specific MGL file of the *MealyMachine* an instance of the MGL metamodel is initialized. The visual representation of this object composition is shown in figure 2.2. This object structure reproduces the defined types in the MGL which are instances of the `MGL.ecore` shown in figure 2.1. The code generators, which will realize

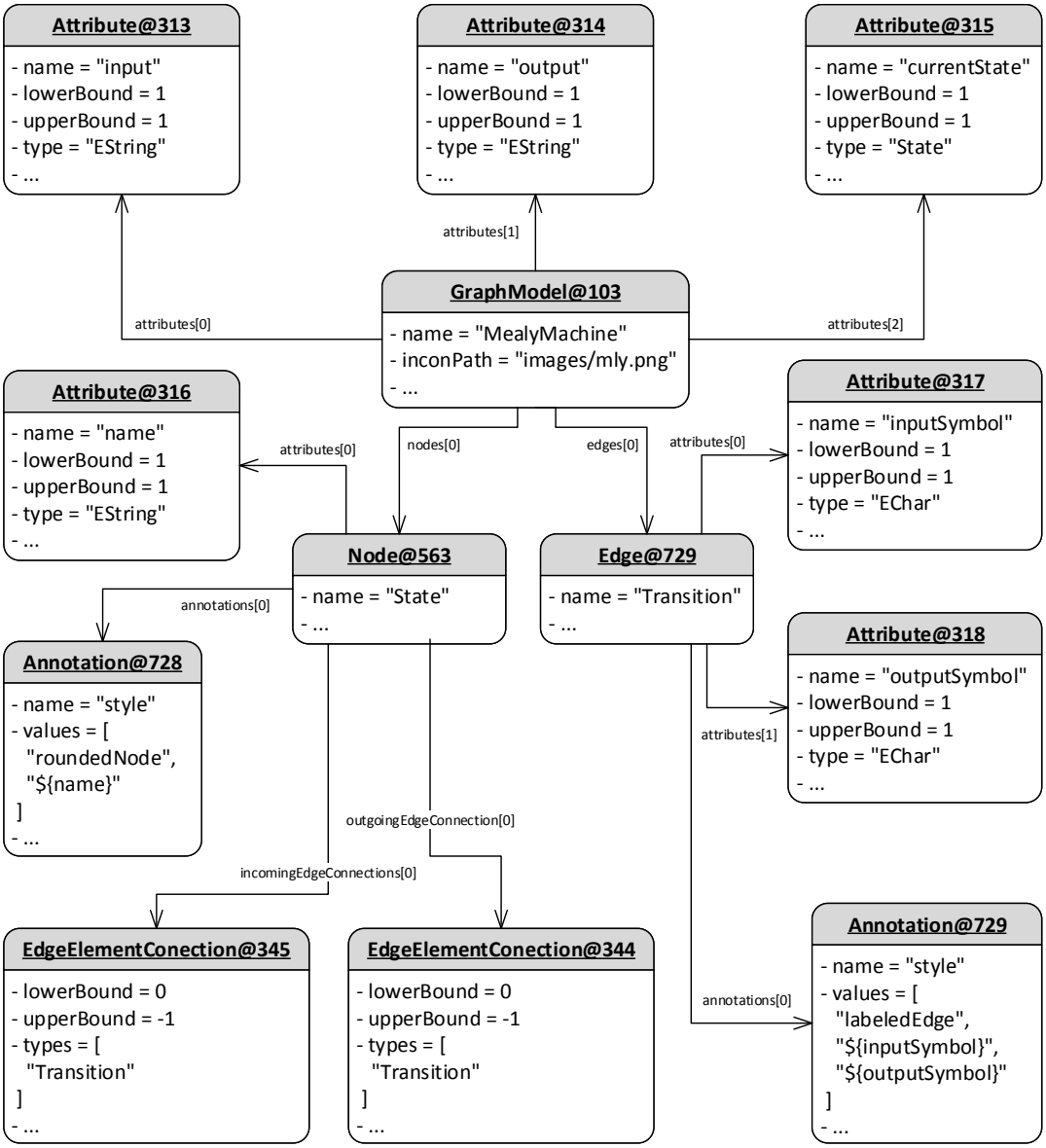


Figure 2.2: The object structure of the `MealyMachine` MGL instance.

```
graphModel MealyMachine {

  package info.scce.cinco.product.MealyMachine
  nsURI "http://cinco.scce.info/product/mealyMachine"
  iconPath "images/mly.png"

  attr EString as input
  attr EString as output
  attr State   as currentState

  @style(roundNode, "${name}")
  node State {
    attr EString as name
    outgoingEdges (Transition)
    incomingEdges (Transition)
  }

  @style(labeledEdge, "${inputSymbol}", "${outputSymbol}")
  edge Transition {
    attr EChar as inputSymbol
    attr EChar as outputSymbol
  }

}
```

Listing 2.1: An exemplary mealy machine MGL.

the Cinco Product, reads out the object instances to establish the domain specific model and the additional features. The object structure is accessible by the graphmodel instance, which contains the references to every declared component. The name, the package and other specifications are stored in the equally named attribute values of the graphmodel instance. In addition to this, the defined attributes like the input and output strings are realized as instances of the attribute type, associated to the graphmodel. The attribute's lower and upper bound are set to one, since they are not specified as lists with cardinalities. The state node is presented as a node type instance which, references its attribute as well. The style annotation is instantiated and extended with the parameter list, defined in the mealy machine MGL. Similar to the state node, the transition edge is represented in an edge instance, which references its attributes and the style annotation.

Next, it is necessary to set the visual representations of the components. This requires to determine the associated MSL file, which contains the style types to illustrate the states and the transition. Every component gets visualized by its corresponding style type. The definition of the style types will be explained in the next section 2.1.2.

2.1.2 Meta Style Language

The Meta Style Language (MSL) [9] enables the developer to specify the visual representation of the components in the generated CINCO Product, described in the MGL. The MSL is constructed to be as dynamic as possible to not limit the creativity of the developer.

The style components differ in their relation to an MGL component. As a result of this, the supported style components limit themself to an `edgeStyle` and a `nodeStyle`. A style declaration is initiated with the keyword of the style component and an unique name for the style, which will be referenced in the MGL style annotation. To show attribute values of the model in the visual representation of the component, parameters can be passed to the style by setting the required amount. The metamodel of the MSL, which is specified in the `MSL.ecore`, is visualized in figure 2.3. The MSL primarily differentiates between node styles and edge styles, although they can be accessed together in the `styles` List of the `Style` type.

The styling of an edge is characterized by a list of connection decorators and an appearance. The edge connection decorators afford the developer to define various decorators on the edge. Every decorator is specified with a location relative to the source, which has not to be mandatory in the CINCO Product tool by approving the movability. On the one hand the MSL provides the possibilities of creating individual decorator shapes by defining a polygon or a polyline and on the other hand predefined shapes can be used like triangles, rectangles, rounded rectangles and ellipses. In addition to this, images and textareas can be used as a decorator shape. Over and above these potentialities, every shape can be extended by an arbitrary value of child shapes. For this case every shape, which extends the `ContainerShape`, can be defined as absolute positioned or relative aligned with an optional offset. Furthermore the shapes are characterized by their size, position and color. The MSL is constructed to strongly support code reuse.

In contrast to the edge style, the node style consists of one main shape which can be extended and overlain by multiple other shapes like described before. To permit the full scope of the developers creativity every shape is capable of being assigned to an appearance. An appearance defines the visual representation of a component which references it, with properties like colors, fonts, etc.. The entire listing of all properties and possibilities of customization is shown in figure 2.3. Another feature of the MSL concentrates on the dynamic variation of the visual representation. To meet this demand, an *AppearanceProvider* can be assigned to a style, to modify the predefined component at runtime. This permits the style with a behavior extension to visual react on model transformations or attribute changes.

At this point a concrete example of a MSL will be explained references to the mealy machine MGL in the last section 2.1.1. First the necessary styles, which are referenced in

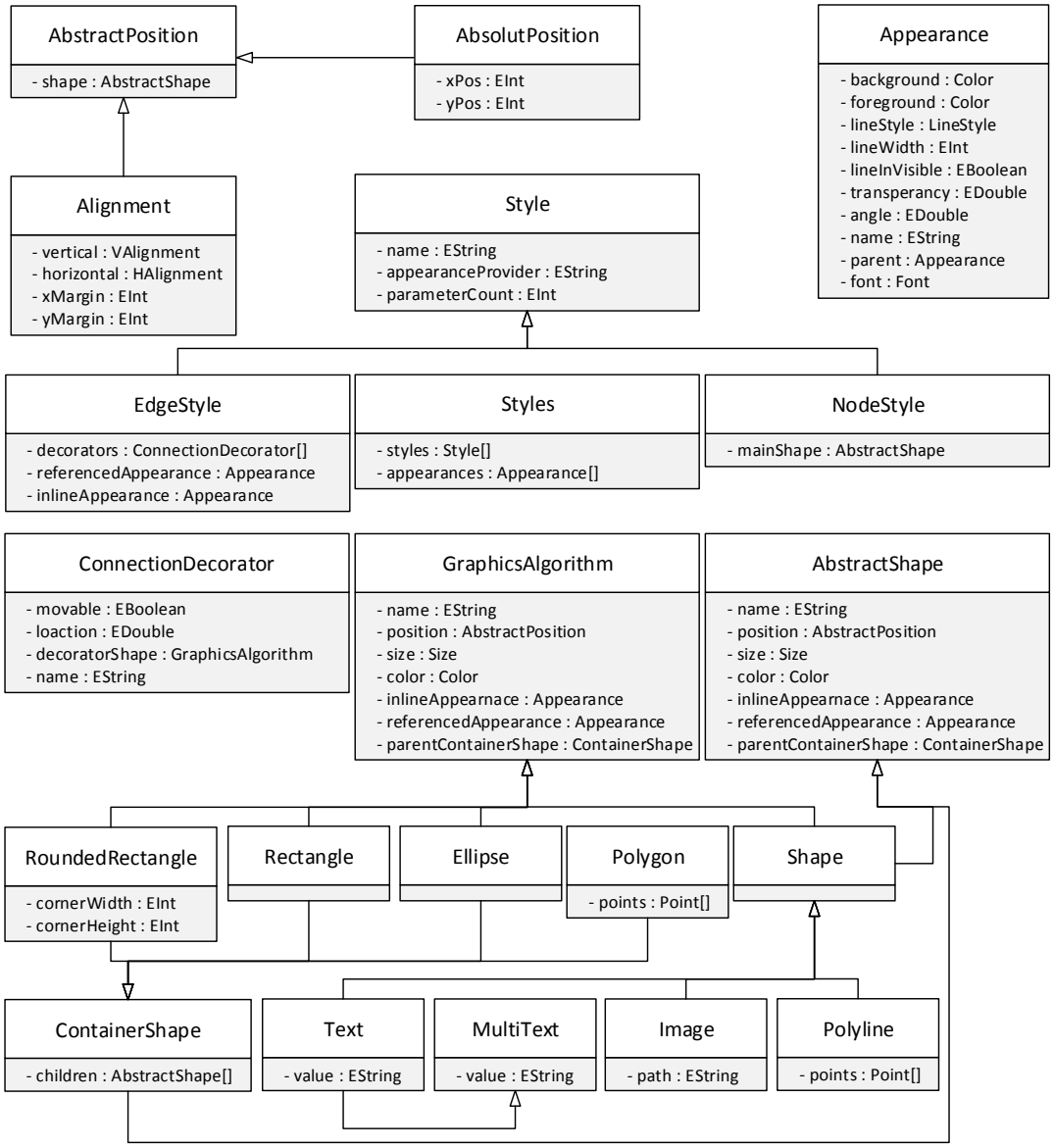


Figure 2.3: The meta model of the Model Style Language. `MSL.ecore`

```
appearance default {
  lineWidth 2
  background (229,229,229)
  foreground (0,0,0)
}

nodeStyle roundedNode (1){
  ellipse {
    appearance default
    size(40,40)
    text {
      position ( CENTER, MIDDLE )
      value "%s"
    }
  }
}

edgeStyle labeledEdge (2) {
  appearance default

  decorator {
    location (1.0)
    ARROW
    appearance default
  }
  decorator {
    location (0.5)
    text {
      appearance default
      value "%s / %s"
    }
  }
}
```

Listing 2.2: An exemplary mealy machine MSL file.

the `mealyMachine.mgl` (see listing 2.1), will be created. The state node refers to the node style `roundedNode`. The edge style `labeledEdge` is referenced by the transition edge.

In a classic mealy machine, the states are represented by circles, which contain the state number as label. To enable the dependence of the style to the label of the state, the `roundedNode` style will be parametrized by one parameter, which will be passed from the state node MGL model in the generated CINCO Product tool. As mentioned before a node style contains one main shape, which will be set to an ellipse for the state. To modify the default visualization of the ellipse an appearance is defined, which determines the line width and a size to specialize the ellipse as a circle. To show the value of the corresponding `name` attribute, a child *text* shape which will be positioned in the middle of the circle by setting up a relative positioning to the *ellipse* in the vertical and horizontal center. The value to be displayed, is passed by a notation known from the Java `String.format` method [8],

which replaces every place holder in order of appearance by the assigned parameters. As an additional customization, the appearance of the text shape is modified with a specialized font, size and emphasis.

The edge style `labeledEdge` is used for the transition edge, which should be visualized as a black directed edge with an arrowhead and the label for the input and output character. The color and linewidth of the edge itself is described in the `default` appearance which is declared in the global scope of the style and can be referenced all over. A connection decorator is defined and shifted to the end of the edge with the default appearance to look uniform. The arrowhead is available as the predefined decorator shape `Arrow`, so that a polyline is unnecessary. As mentioned before the mealy machine associates an input symbol with an output symbol, which is produced when the defined input is read. The input and output characters for the transition edge will be stored as values of the attributes and should be illustrated on the edge as well. The presentation is realized with a text decorator located in the middle of the edge. The value of the text decorator is created as a string template which consists of the input symbol followed by a slash and the output symbol. This labeling matches the known representation of the mealy machine.

At this point the development of the exemplary mealy machine CINCO Product tool is completed. The domain specific tool is generated without any additional configuration or set up and can be used directly to model. The generation and essential model transformation to facilitate the CINCO Product on a given MGL and MSL will be described in section 2.1.4.

2.1.3 Prime References

The idea behind CINCO pursues the goal to enable a developer to create any imaginable modeling tool. The possibilities to comply with these exigencies are extended with the explained high level modeling languages and multiple plugins. Another point of matter is the reuse, importing and execution of external libraries. In the subject of CINCO these references to other external components are called *Prime References*. This crucial feature approves CINCO to realize external dependencies to other models and its instances. To meet this goal a `prime node` can be declared in an MGL. The visual representation of the prime node is defined as usual in a `nodeStyle` of the assigned MSL. An exemplified code snippet is shown in listing 2.3. The prime node indicates one reference to an external EMF `EClass`. This referenced external type is defined after the `for` statement of the prime node declaration. To enable the reference, the necessary file, which contains the type to be referenced has to be imported in the MGL. Usually the required types are contained in another Ecore meta model which in the majority of cases is the model of another CINCO product as well. Therefore CINCO provides special support for an Ecore meta model import, including auto-completion for the prime reference to select the desired

```
1  @primeViewer
2  import ".../subModel.Ecore"
3
4  ...
5
6  @pvLabel("name")
7  @pvFileExtension("sm")
8  prime node ContainsSubModel for SubModel {
9    prime attr SubModel.internal as readInternal
10
11   prime parameters {
12     SubModel.inputName
13     SubModel.inputType
14   }
15 }
16
17 ...
```

Listing 2.3: An exemplary prime node.

type. In the generated CINCO product, the reference of the prime node will be solved, when it is dragged to the canvas. As a result of this, the CINCO product seeks for every instance of the predefined referenced type of the prime reference. To prevent the browsing of all files in the current directory, a file extension can be preassigned to filter the files before their content is examined. The filter function can be activated by the `@pvFileExtension` annotation, which is assigned to the prime node. The detected instances of the type are collected to be listed in the *primeViewer*. The prime viewer is a meta plugin for CINCO , which creates a tree view for all available prime nodes. It is enabled with an equally named annotation, added to the graphmodel in the MGL. Each prime node, which is defined in the MGL is displayed in its entry. The collected instances of the referenced types grouped depending on the prime node type and listed as sub-entries in the tree view. As a result of this the tree viewer, attempt to simplify the selection of the instance, which has to be associated to the prime node. To specify the displayed identifier of the collected instances in the sub-entries, a label can be specified in the prime node's declaration. This labeling is facilitated by the `@pvLabel` annotation, provided by the prime viewer meta plugin. The pvLabel annotation accepts one argument, which specifies an attribute of the referenced type. Later in the generated CINCO product, the value of the defined attribute is read out and displayed in the tree view as the identifier. To display other attribute values of the referenced types, a prime attribute can be defined (line 9 of listing 2.3), which enables the read out. The prime attributes are mirrored as a read only attribute in the properties view of the prime node instance.

Another relevant use case for prime references is the composition of multiple types to be referenced and especially the insertion of values for parameters. In contrast to the

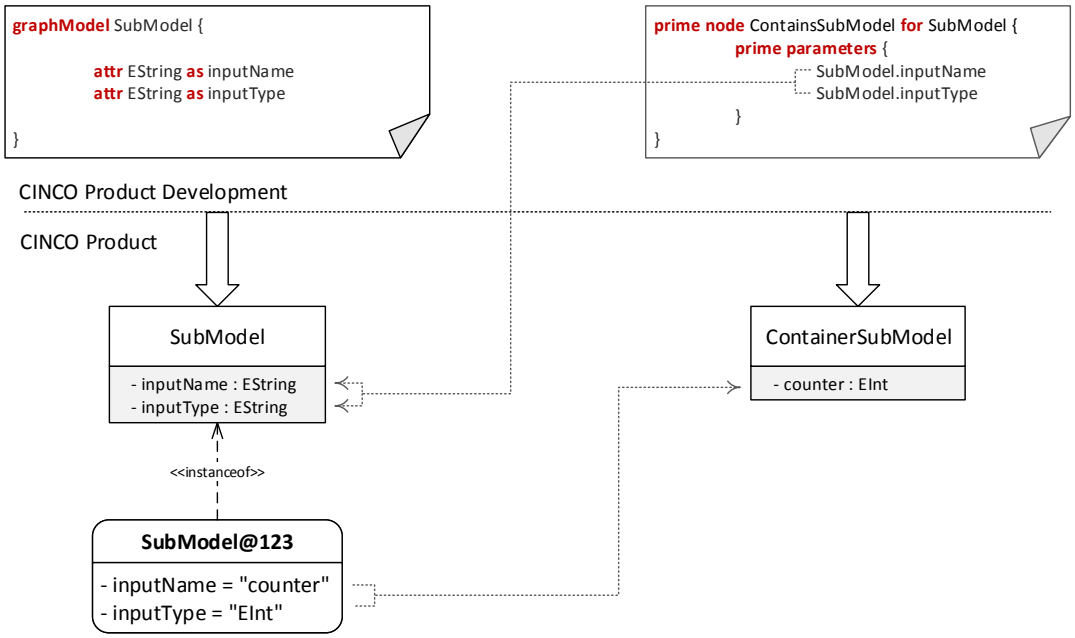


Figure 2.4: Exemplification of the prime parameters.

attributes, which can be determined in the time of the MGL creation, the parameters of a composed type arise from the modeled elements in the CINCO product. To enable this parameter discovery, the prime node is extended with `prime parameters`, shown in listing 2.3 line 10 to 13. A prime parameter holds references to specific attributes of the referenced type, whose values will describe the type and the name of the parameters to be passed. The relationship between the prime parameters and the attributes of the referenced type is exemplified in figure 2.4. The referenced graphmodel `SubModel` declaration is shown on the left and the prime node `ContainsSubModel` on the right side. When a sub model is created, the values of the attributes of the graphmodel specify the required parameters, which has to be defined when the submodel is referenced. To meet this goal, the prime parameters of the prime node are referencing to the attributes of the `SubModel` graphmodel. The first entry in the prime parameter declaration describes the attribute of the referenced type, which contains the name of the parameter, whereas the second entry defines the data type. This approach enables the developer to define the location, where the parameter definition can be read out, when the referenced type is instantiated. Afterwards, the properties view of the prime node displays the necessary input fields depending on the registered prime parameter values.

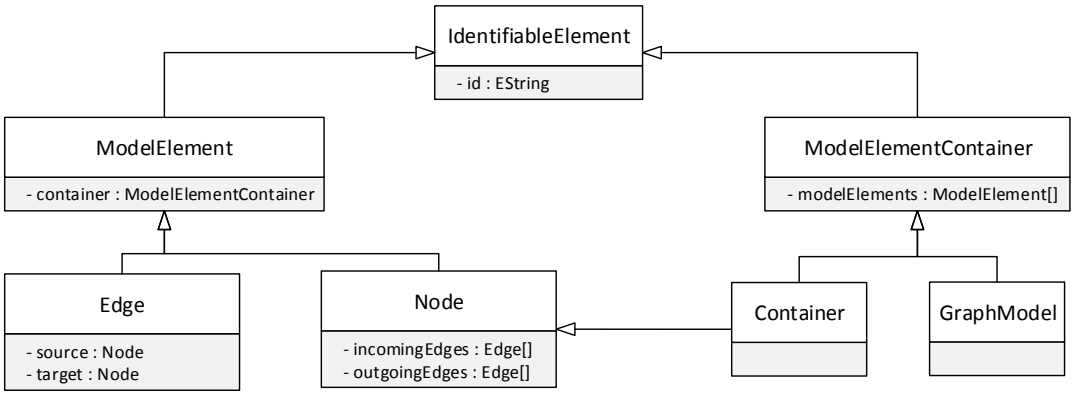


Figure 2.5: The graph model meta model of any CINCO Product. `GraphModel.ecore`

2.1.4 CINCO Product

Once a CINCO Product is specified by the MGL and MSL, unlimited different instances of the domain specific model can be realized. The CINCO Product itself requires its own meta model to store the data. The meta model is described in the MGL and MSL instances and is transformed into a domain specific meta model. Since CINCO limits itself to graph models including nodes, edges and containers, the generated meta model can be prescind from the `GraphModel.ecore`, which is presented in figure 2.5. The graphmodel meta model consists of the necessary types to provide the inheritance of the generated domain specific specializations. The presented types are declared to represent the invariable parts of the graphmodel. For this reason the underlying semantics of graph structures, like the incoming and outgoing edges of nodes, are already mapped in the `GraphModel.ecore`. The foundation of all types is the `IdentifiableElement`, which is specializing in the `ModelElement` and the `ModelElementContainer`. The ModelElement is containable and represents the opposite of the ModelElementContainer, which contains ModelElements. The `Node` and `Edge` types are children of the ModelElement in the inheritance hierarchy and inherit the ability to be contained. Between the Node and Edge types exist a similar relation, which implies the connection of nodes by an edge. The nodes are stored as a source and target Element, whereas the edges are located in lists, distinguished between incoming and outgoing edges. The `Container` extends the Node as well as the ModelElementContainer type, which leads to combined capabilities. As a result of this a Container can contain elements and be contained in another ModelElementContainer. In addition to this, Nodes and Containers can both be connected by edges. The `GraphModel` type only inherits of the ModelElementContainer, which results in the situation, that no element can be connected but contained. The `GraphModel.ecore` offers the entire foundation for the data structure of every graphmodel.

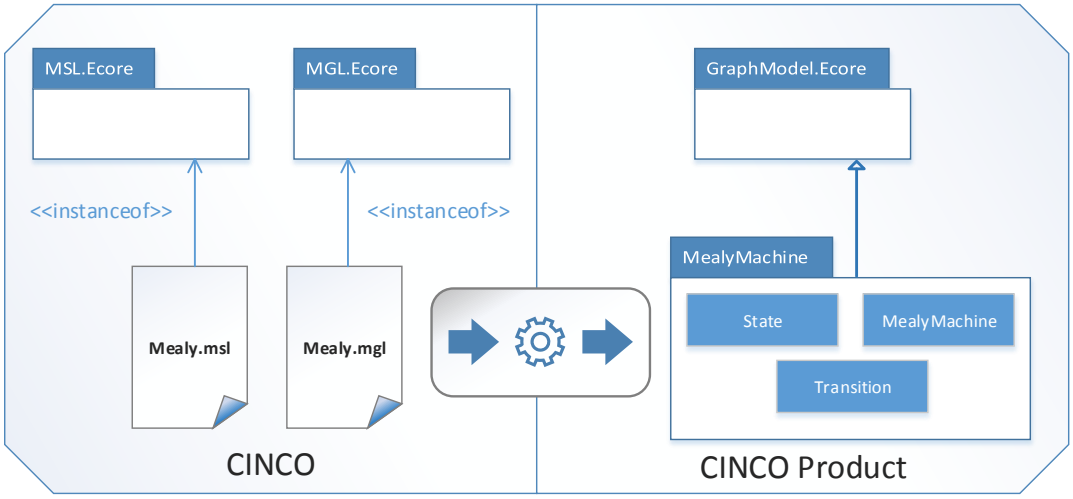


Figure 2.6: The MGL to GraphModel tranformation.

The transformation and generation process for the mealy machine example, which has been introduced in the last sections is pictured in figure 2.6. The MSL transformation is analogous to the MGL transformation and will be omitted to describe the MGL transformation in detail. The mealy machine is specified in the limitations of the `MGL.ecore` and its representing components of the MGL. Every defined Type in the written MGL is indicated by an instances of `MGL.ecore` types as mentioned in section 2.1.1. The instances are the source of the transformation and generation process of the CINCO Product. The information sustained in the `MGL.ecore` instance is used to generate new specialized types of the `GraphModel.ecore` meta model to guarantee type safe modeling. The generated types will be used as business model to manage the data and structure of the realized model in a CINCO Product.

Separate from the specialized types, the CINCO Product requires a graphical user interface to offer intuitive and rapid modeling to the developer. The exemplary integrated development environment (IDE) for the mealy machine CINCO Product is shown in figure 2.7. The tool itself is developed as an *Eclipse Plugin* to take full advantage of the usability features and the accustomed environment for the developers. The main part where the action takes place is the modeling canvas, which represents the graphmodel instance. The specialized nodes and containers can be arranged and connected by edges. Edges are created from source to target and validated referring to the connection constraints, which are generated as well. CINCO follows up with the principle to disable actions instead of later fault indication as much as possible, to improve the user experience. To provide the possibility of a clear arrangement nodes can be resized and moved and edges can be bent by dragging. Concerning to the semantics, that every edge holds a defined source and target, the removing of a node or a container which is connected to other element causes

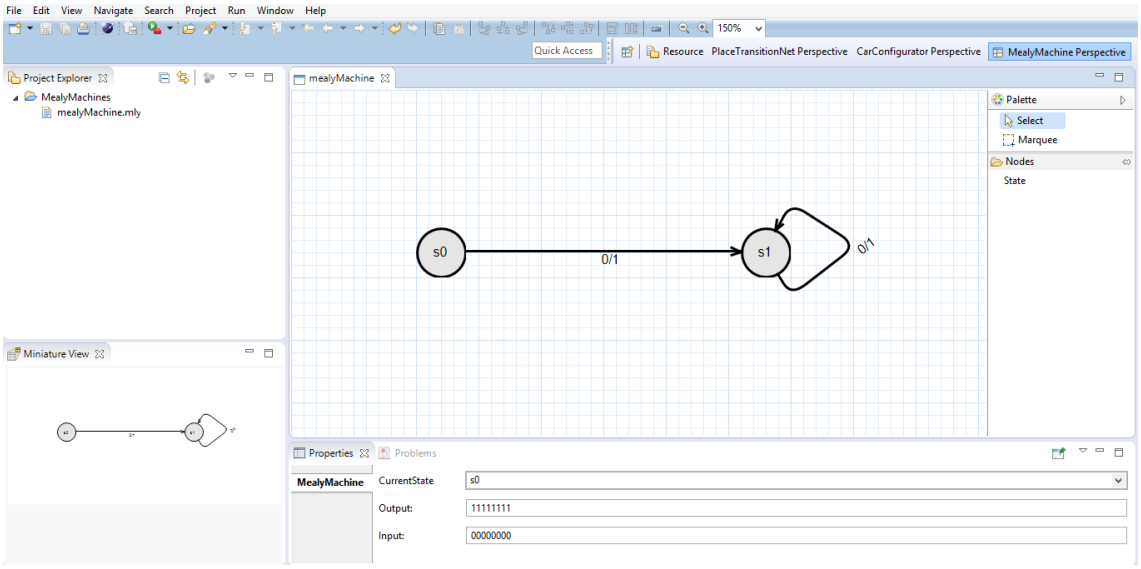


Figure 2.7: The CINCO Product IDE for the mealy machine.

the erasure of all connected edges. The context menu enables the developer with known fundamental operations like cut, copy and paste as well as the predefined *contextMenuActions* described in the MGL file, to perform model transformations and modifications. Despite the modeling canvas the attributes of the selected component are displayed in the *Properties View* of the IDE. Every Attribute is displayed with its label specified in the MGL and its current value stored in the `GraphModel.ecore` specialization instance. The input fields of the attributes form are type safe to the declared *EDataType* [20]. To create new nodes and containers and gather an overview over all available components a *Palette* as a grouped list is generated and displayed in the CINCO Products. The palette contains all nodes and containers, specified in the MGL which can be added to the canvas by drag and drop. Furthermore the components can be grouped individual by defining groups for every component in an annotation just as a preview image which will be shown.

The CINCO Product is a fully generated, domain specific, ready-to-run tool and can be deployed as a standalone eclipse instance. As a result of this CINCO is well on the way to be a one thing solution (in dependence on OTA [11]) for model driven development.

2.2 Dynamic Web-Application

The dynamic Web-Application (DyWA) [16] is designed to enable online prototype-driven development. The main constituent is the ability to define the meta data schema and constitute instances of the defined types with migration support for the complete application life cycle. This feature will be explained using the example of a simple hotel administration. First the meta modeling environment will be described, which enables the developer to model the required domain with types, fields and relations. In the second section, the

instance controlling is illustrated, using the same example as the section before to clarify the evolution of the project.

Next, the DyWA prepare the utilization of the defined types with full code generation of domain specific entities and controllers to enable basic specific CRUD [?] functionalities to the developers. The DyWA provides multiple additional features to support the developer by mastering his task without the necessity of knowing the entire tool stack, like process execution, which will not be mentioned at this point because of focusing on the meta modeling, which is significant for the rest of this thesis.

2.2.1 Meta Modeling

The ideas and concepts behind the DyWA are tightly focused upon the usability for domain experts, who are not necessarily familiar with the complete technology stack of implementing an application for the web. Thus, the DyWA tries to conquer the semantic gap between a technical and a domain expert. The realization should be possible without writing any line of code. To comply with these goals, the DyWA provides a form based domain drawn meta modeling, which allows the developer to define the meta model of the application to be created. In addition to this the DyWA offers further reliefs and support for the developer like type-safe data orientation, built-in dependency injection and integrated generators for domain specific code. The behavior of the domain meta modeling is similar to known object orientated modeling environments like UML class diagramms [?] or EMF [20], which support types, attributes and references. In addition to this the DyWA allows the developer to use inheritance and subtyping for reuse, polymorphism and extensibility. The meta modeling environment is part of the DyWA administration layer, which describes the underlying control plane.

A type, which is called *DBType* in the scope of the DyWA, is specified by a name which has to be unique for the entire application. All DBTypes are stored in one data base table with an identifier, its name and an optional description for further documentation. The entity-relation model (ER-model) [3] for the DyWA meta modeling environment is shown in figure 2.8. After the creation of a DBType, *DBFields*, which are analogous to attributes in OOP, can be assigned. The developer is able to define DBTypes determined as abstract, which cannot be instantiated but specialized in multiple ways. Thanks, to the prototype-driven development provided by the DyWA, it is possible to modify the meta model at any time in the development life cycle. Thus, DBFields can be added, removed, renamed or redefined with another type. Every DBField is defined by its name and is dedicated to one of two veins, which differentiate between a primitive and a complex attribute. The primitive attributes are restricted to simple data types like integers, binaries, floats, strings and dates as well as lists of strings. The complex attributes instead represent references between types and can either be a single reference or a list of references. The DBFields

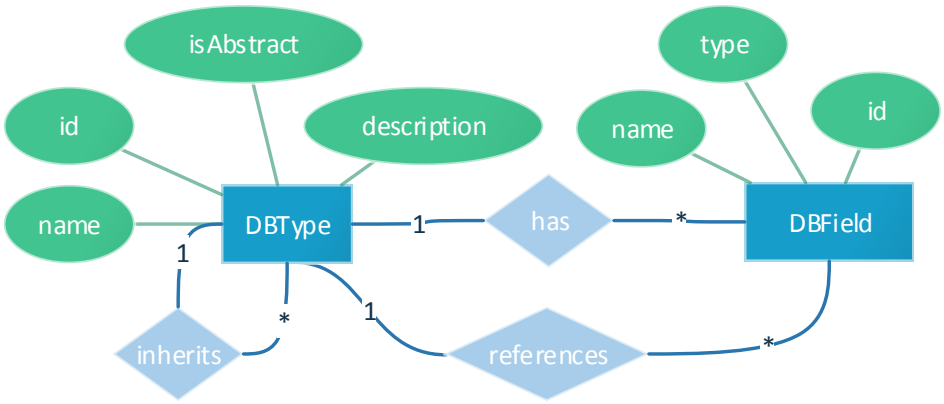


Figure 2.8: The DyWA meta model ER-Model.

are stored in its own table with a reference to its assigned DBType, name, data type and an identifier. To assemble the entire type defined by the developer, the entry of the DBType table and every associated DBField is loaded. For the complex attributes the same mechanism is used to realize the references between the DBTypes. This capabilities enable the developer to accomplish the ideas of a web applications meta model by just fill in a form.

To amplify the comprehension of the DyWA meta modeling, a hotel administration web application will be defined as an example. The hotel administration deals with the representation of the hotels rooms, its status and the received visitors. To pose an additional challenge the application is supposed to differ between three varying room categories which are characterized by different properties. First, the hotel itself is specified with a name, location and its stars. Every room, which is assigned to the hotel, should declare its floor, its price and if it's occupied. The apartments are additional specified by the number of beds, whereas the bridal suite is extended with a list of optional requests. The data base schema in relation to the UML class diagram of the hotel administration application is displayed in figure 2.9. This demands are met with a new `Hotel` type, which is extended with three primitive attributes. The location and name of the hotel are textual attribute which are specified as strings. The amount of stars which are bestowed to the hotel are represented with a numeric attribute of the integer data type. The DBType of the hotel which is, shown on the right in figure 2.9, is referenced by the three DBFields described. The rooms of the hotel resemble each other in the attributes of the floor, price and booking. In this case an inheritance is used to extend the `SimpleRoom` with the `Apartment` and the `BrideSuite` type. This affords the referencing of all rooms in one list of the hotel

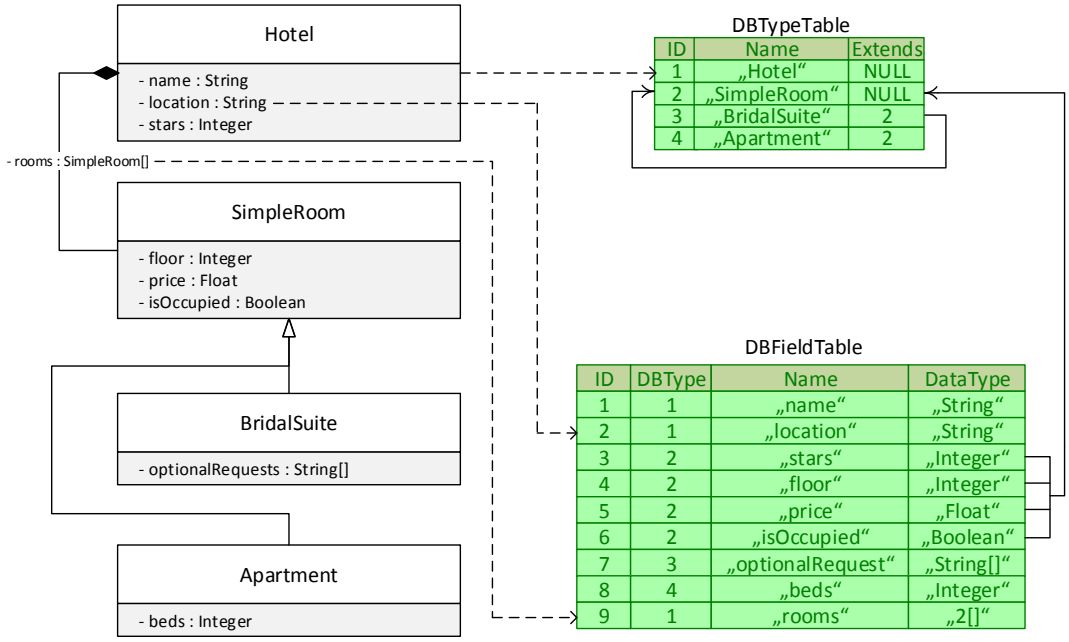


Figure 2.9: The meta model of the hotel administration application with regard to the DyWA database schema representation.

and in addition to this avoid redundancy. Thus, the simple room and its specializations are enhanced with the predefined attributes. As a result of this the DBType's `extends` columns of the Apartment and BrideSuite is specified with a reference to the SimpleRoom DBType. When this inheritance is solved, the Apartment and BrideSuite are extended with the attributes of the SimpleRoom. At this point the meta modeling of the domain is finished. The next section illustrates the creation of instances of the defined DBTypes.

2.2.2 Instance controlling

The DyWA provides also the possibilities to create, modify and delete instances of the predefined DBTypes, which are called *DBObjects*. Instead of the requirement of writing lines of code to instantiate and use DBObjects the DyWA offers an user interface on its administration layer. The user interface is also form based and covered for quick access and intuitive handling. To create an instance of a DBType, first the type and a name has to be chosen. The name is used for the determination of multiple instances of the same type and applies analogously to the `toString`-Method implementation known from programming languages like Java [17]. It is used for displaying and searching all DBObjects in one list in the user interface. After the creation of a DBObject, the attributes can be accessed and determined. This includes complex attributes which are realized as a single or multiple choice list, containing all the instances of the referenced type.

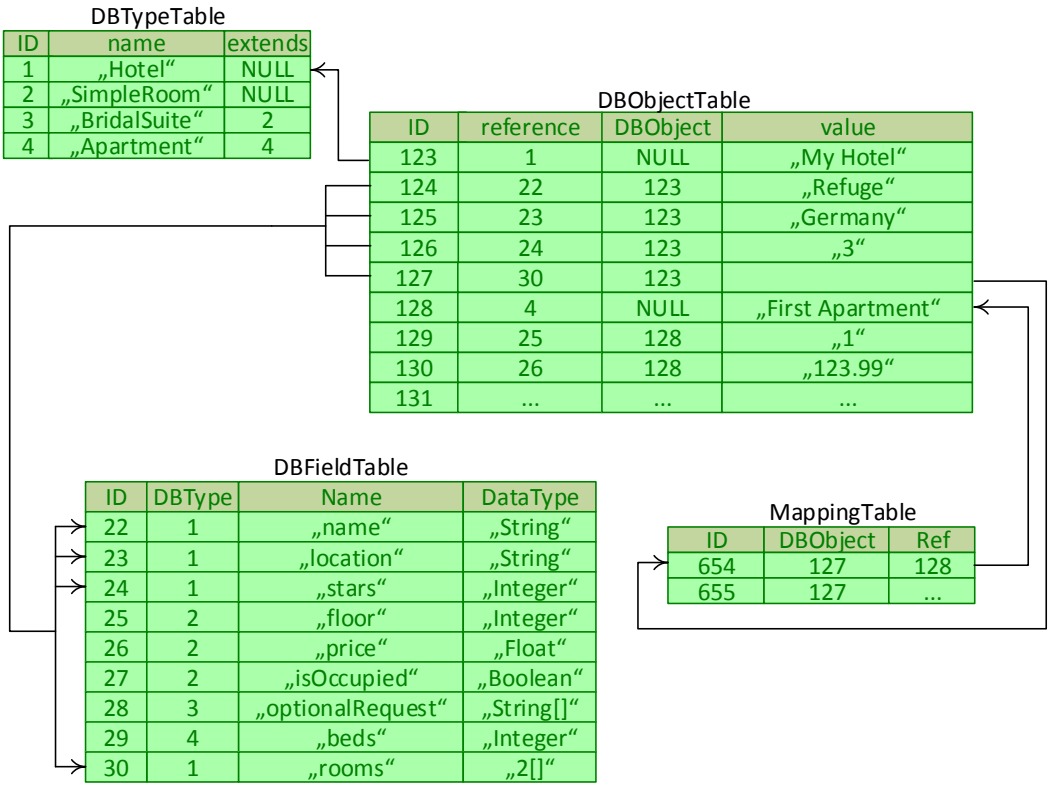


Figure 2.10: DBTypes and its instances of the hotel administration example.

The storing of the DBObjects is realized in its own database table, which is overall distinguished by a key value map. This means that a DBObject only consists of instances of DBFields which are represented by the value of the attribute and references to the incidental DBType and DBField including the default name attribute. This interrelationship becomes apparent when the hotel administration, which were mentioned before, is created. Figure 2.10 shows the hotel administration DBTypes and its instances. The hotel itself is created and named „Refuge“located in „Germany“. The result of this are three new entries with values in the table of DBObjects, which are referencing the hotel's DBType and the associated DBFields. The complete information of the DBObject is fetched and reconstructed on demand. The rooms for the hotel are created in the same way with a type and a name, except the properties which are inherited from the SimpleRoom DBType. The values of the inherited DBFields are referenced to the DBField of the SimpleRoom, but associating the specialized DBField for the Apartment or BridalSuite. To set the association between the hotel and its rooms, the hotels `rooms` attribute is edited. The attribute value is reconstructed using the entries of the `MappingTable`, which shows all containing references in the list, which includes Apartments and BridalSuite It becomes apparent that the management of the DBObjects is very difficult to handle and causes a lot overhead. This

conduction is simplified with the help of domain specific controllers and entities generated by the DyWA, which will be explained in the next section.

2.2.3 Domain Generation

After the modeling of the domain specific data types, attributes and relations, code generators can be used to obtain specialized entities and the depending controllers. The generated *Entity Beans* are server-side Java EE components that represents persistent data maintained in a database [**?**]. Every modeled type is transformed into a Java Interface to enable multiple inheritance and its implementing class. Thanks to the generated entities, the access to the DBTypes and its instance information is mapped to an domain specific entity and can be modified analogous to known java objects with get and set methods. The generated entities offer the possibility to handle the modeled domain as anticipated. To facilitate the migration support, the identifier of the incidental DBType and its DBFields are written into the generated entity. As a result of this, the renaming of a DBType is even possible after instances of the DBType already exist.

The entities itself do not suffice to comply with the requirements of the CRUD functionalities, as they only afford the modification and readout of the stored instance data of one entity. To fulfill the demands to easily create a new instance of a specific type and fetch or remove instances, domain specific controllers are generated depending on the modeled types. The controllers provide multiple ways to receive instances of the associated type, which embrace single reads by a given identifier as well as a complete selection of all instances. The selection methods are extended to optional fetching the instances including the subtypes which can be filtered before. In addition to this, the creation of a new instance can be performed by the controller, which requires a given name for the object to be distinguishable from the other instances. The erasure of an instance is feasible by just passing the reference to the controllers delete method. Certainly the developer takes responsibility to detach all associations between the instance to be removed and other instances, before deleting it. This constraint forces the developer to proper abrogate all associations.

Over and above the controller and entity generation, the DyWA provides domain specific hooks, which are realized by an prioritized event handling. The hooks are triggered on predefined events which are lean on the CRUD functionalities like creation, modification and deletion. Every hook is available in a pre and post variant, to ensure the opportunity to execute procedures before and after an appeared event. The hooks are created as a plugin for the DyWA, which provides interfaces for extending the functionalities. The plugins are implemented as generators, which will be executed in the domain generation of the controllers and entities, to tender specialized functionalities.

In summery the DyWA provides an approach to enable an domain expert with no previous expertise in web engineering with the power of creating a targeted domain by modeling the data types and controlling the instances. The modeling environment is created in a form based design to offer an intuitive and familiar management. Considering the evolution of an application, the DyWA features a migration support for domain changes after the inception of using it and the creation of instances. Furthermore, the advanced usage of the DyWA is supported by domain specific generated entities for direct and well-typed access as well as controllers for the administration. The generator plugin interface provides the possibility to extend the capacity of the DyWA with further domain specific features.

Chapter 3

Pyro

This chapter embraces the conception, design and development of Pyro. The idea behind Pyro is the shifting of Cinco Products to the web to combine the given benefits with the advantages of an internet platform. To fulfill these goals, Pyro has to provide at least the same usability and appliance as well as additional requirements like multiuser support and interoperability to already existing Cinco Products. An outstanding demand is the unrestricted employment of Pyro to every Cinco Product, which implies the necessity of the complete reproduction of all Cinco features like custom actions and hooks of the MGL (2.1.1) as well as all possible shapes and graphic algorithms of the MSL (2.1.2). As a cladding the DyWA and its meta modeling abilities are used to embed the transformed model of the Cinco Product. Besides the DyWA domain generation encouraged the realization of the server side of Pyro, so that a bridge is build between Cinco and the DyWA.

The concept to achieve this requirements is separated into two parts. The first part (section 3.1) describes the Pyro Meta Plugin (PMP), which reads and transforms a given MGL and MSL to afterwards enable a complete code generation of the Pyro Modeling Environment (PME). The second part concentrates on the modeling environment (section 3.2) and its functionalities for the modeler as well as for the more advanced developers, who need to maintain various interfaces to exert influence. To permit the same user experience as given in Cinco Products, the PME has to behave similar to a desktop application, which requires advanced asynchronous server client interaction, command handling for redo and undo functions and usability features of a rich internet application like context menus.

3.1 Meta Plugin

The meta model collection, transformation and generation of the Pyro Modeling Environment is performed by the Pyro Meta Plugin. This plugin is designed for the Cinco Meta Plugin interface, which enables Cinco developers to extend the functionalities of Cinco on

the meta layer. Every meta plugin is brought into action by annotations, which can be assigned in the MGL to the components, attributes or the graphmodel itself. To provide additional parametrization, the annotations accept a list of string parameters, which can be individually evaluated. Upon the completion of the component specification and its annotation in the MGL, the developer is able the trigger the product generation. At this point the CINCO generation process iterates over every assigned annotation and invokes the related meta plugins. When a meta plugin is activated, it receives the parameters which are specified in the MGL and the actual eclipse context, which includes the entire CINCO Product project and especially the attended MGL, MSL and CPD. This files can be loaded and transformed into object structures like the mealy machine MGL instance example shown in figure 2.2. By traversing the present structures, the complete CINCO Product specification can be read out and processed in every meta plugin.

The @*pyro* annotation is available to be assigned to a graphmodel like the @`primeViewer` annotation shown in listing 2.3. The developers has to pass two parameters to the annotation, which define the file path to the location where the generated DyWA App should be included and the path to the application server. With regard to the requirement, that Pyro is suitable for every CINCO Product without broader modification of the MGL, no further annotations, conventions or requirements have to be considered.

To enable the porting of the specified component's meta model and its underlying data layer, defined in the GraphModel.ecore, a mapping to the DyWA DBTypes and DBFields has to be created. This model to model transformation which forms the foundation of Pyro is described in the first section. Depending on the generated entities and controllers based on the injected meta model, the necessary functionalities of Pyro are generated and written to the DyWA App. The code generation which manufactures the required files is done by the Pyro Meta Plugin as well and is explained in detail in section 3.2. This process covers the generation of multiple layers and its files contained in the Pyro Modeling Environment, shown in figure 3.1. The lowermost layers, located on the left, are realized by the DBTypes and DBObjects and the generated domain specific entities and controllers provided by the DyWA, followed by the layer of *Commands* and the *Transformation API*, which symbolize the entry point for extensions. The Transformation API is implemented in CINCO as well, so that the negotiation of already existent features is given in the majority of cases. To provide multiuser support and redo and undo actions, the command pattern [?] is used, which saves encoded user interactions in the Pyro application for synchronization and versioning. The upper layers *Message Parsing* and *Page* take responsibility to render the web page, send, receive and parse messages from and back to the clients. The received messages will be interpreted, validated and executed on the Transformation API, before an response is created and returned back to the sender.

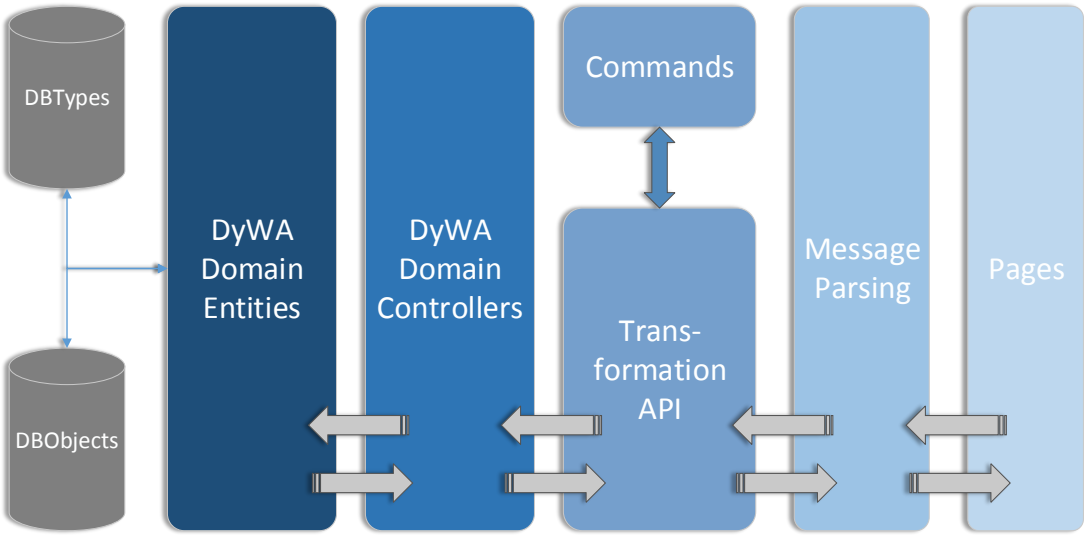


Figure 3.1: The layer structure of the Pyro Modeling Environment.

```
1 DBType country = this.typeController.createType("Country");
2 country.setAbstractType(false);
3 typeController.addPrimitiveFieldToType(country,"residents",PropertyType.INTEGER);
4 typeController.addPrimitiveFieldToType(country,"foundation",PropertyType.DATE);
```

Listing 3.1: An exemplary DyWA DBType creation.

3.1.1 Model Transformation

The model transformation of a CINCO Product is based on the Model Graph and Model Style Language files, which for the most part define the components and its visual representation. As shown in figure 2.6 the instances of a given MGL and MSL are read out and interpreted as a description of the meta model of the CINCO Product. This meta model is generated afterwards into new types, which are specializations of the `GraphModel.ecore` (shown in 2.5), since every CINCO Product is predicated on the edge, node, container and graphmodel components. As a result of the fact that every meta model of a CINCO Product is based on the `GraphModel.ecore`, it is necessary to reproduce the according types in the DyWA.

GraphModel.ecore reproduction

The actual DBType creation is solved by the execution of Java-Code, which is generated into a DyWA generator plugin. An exemplary DBType construction of a simple `country` DBType is shown in listing 3.1. The `TypeController` offers multiple methods to create, delete and extend the DBTypes. This methods are rested upon the possibilities of the DyWA administration user interface, so that the inheritance, the attributes and the ab-

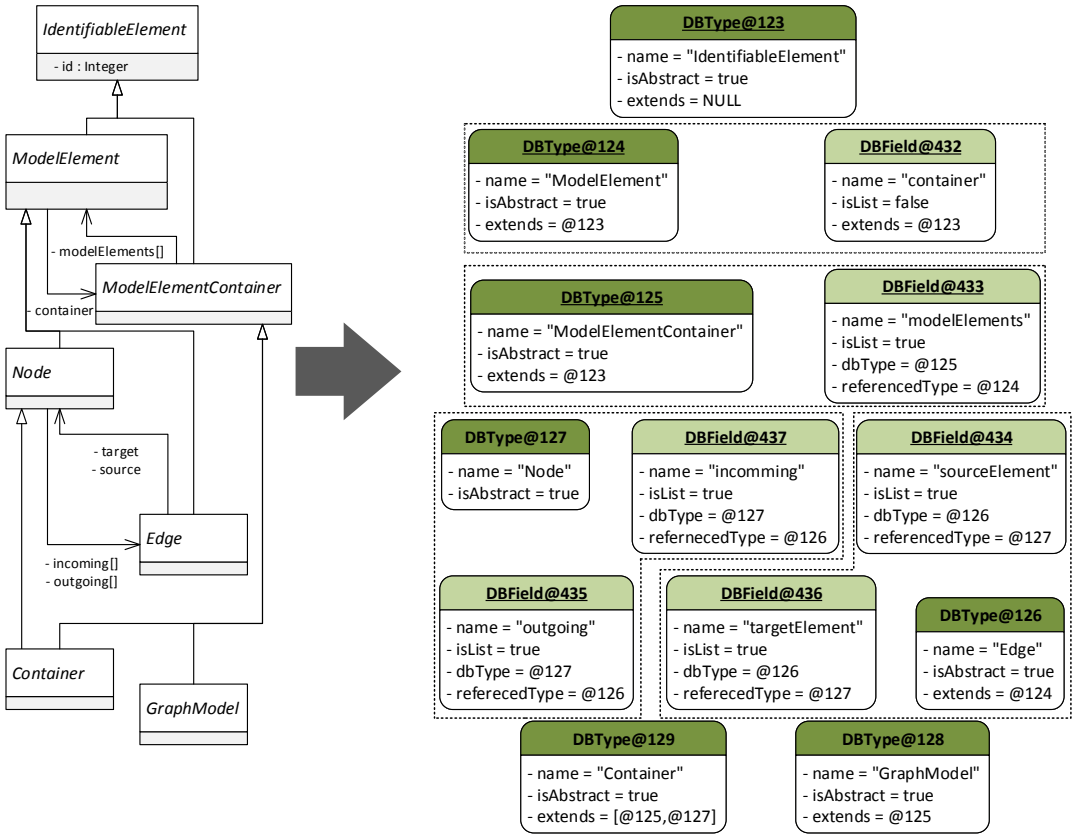


Figure 3.2: Reproduction of the `GraphModel.ecore` in the DyWA.

straction status are definable after the initial type declaration. The appropriate method calls on the `TypeController` are illustrated in listing 3.1. The type is created with a given name, no description and is declared as not abstract, so that it can be initialized. After this creation two attributes are assigned to the country DBType, which specify the number of residents in the country and its founding date. Based on this way of creating DBTypes the model transformation is conducted.

The development of the `GraphModel.ecore` porting is realized in a one-to-one mapping, which is pictured in figure 3.2. The exposure shows an object orientated reproduction of the DyWA DBTypes and DBFields, since to actual database layer is hidden from the DyWA app developer. The rebuild of the `GraphModel.ecore` in the DyWA guarantees the re-usability of already implemented CINCO Product features and the equality for further employment. The reproduction is executed top down to assemble the underlying abstract types at first until the superior specialized types, which represents the known CINCO modeling components. This results in *Level-order Traversing* of the hierarchical tree representation of the `GraphModel.ecore`. The level-order traversing is required, since the inheritance as well as the relation between different types in the DyWA is only feasible for already defined types. The first type to be reproduced is the IdententifiableElement,

which is the root element to all following types. To map this type to the DyWA a new DBType is initialized, with the name of the type, the abstract flag and no inheritance. Since every DBType is already distinguishable by a predefined identifier, the `id` attribute of the IdentifiableElement is ignored. The next two types ModelElement and ModelElementContainer are the super-types for the components, which differ in the ability to be contained in a ModelElementContainer or to contain other ModelElements. Each of these types is represented by new DBType with a corresponding name, the abstract flag and an inheritance relation to the IdentifiableElement as the super-type. At this point a difficulty occurs, when the referencing attributes are about to be assigned to the DBType, because of the *Circular Dependency* between the relation of the attributes. This problem is not met within the level-order traversing, since the referenced type depends on a not defined type at this moment. One approach to solve the circular dependency is *Forward Declaration*, which is particularly useful for one-pass compilers and separate compilation as well as the analogous linear execution. In the DyWA context this approach is translated, so that the initial declaration of a type is separated from the further actual definition of the attributes. As a result of this all attributes of every given type are defined after the initial declaration of all types to enable circular dependencies for an one-pass execution. The edge type of the `GraphModel.ecore` is transformed to the equally named DBType, which is extended by two DBFields for a source and target reference to instances of the node type. On the reverse side the node DBType contains two lists of incoming and outgoing edges, which are both complex DBFields to store a list of references to edges. Exactly like the containment to containing relation between the ModelElement and the ModelElementContainer, the edge and node relations are bidirectionalizations to each other. The container and graphmodel type are specializations of the ModelElementContainer, which enables the ability to contain other nodes, edges or containers. The container is additionally extending the node type, so that it can be contained itself. Since every of this referencing attributes is inherited, no further DBFields are required for the graphmodel and the container.

In Cinco the visual information and the business model are differently stored for the separation of concerns like semantics and visualization. Since the visual representation is once defined in the front end of Pyro, it is necessary to store the other changeable attributes like positions and sizes in the DyWA too. As a matter of fact, the non abstract DBTypes, which are directed to a visual representation like edges, nodes and containers, have to be extended with this additional information. Therefore the `Point` DBType is created to store the information of a two dimensional position in the canvas, including two decimal coordinates. The point DBType is referenced by the node and its succeeding DBType to store the position. To store the geometrical information of an edge, a list of points is needed to represent the different bending-points.

DyWA DBField data type Enum	EMF EDataTypes
PropertyType.STRING	EString, EByte, EChar
PropertyType.LONG	EInt, ELong, EShort, EBigInteger
PropertyType.DOUBLE	EDouble, EFloat, EBigDecimal
PropertyType.BOOLEAN	EBoolean
PropertyType.TIMESTAMP	EDate
PropertyType.STRING_LIST	EByteArray

Table 3.1: The relation between the EMF EDataTypes and DyWA DBField property types.

MGL.ecore instance transformation

The different components, which should be available in the CINCO Product, are defined in the MGL. The possibilities of the developer to define those components are described in section 2.1.1 and are based on the MGL meta model the `MGL.ecore`, which is shown in figure 2.1. In the generation process of the CINCO Product, when the Pyro Meta Plugin is triggered, it obtains an instance of the `MGL.ecore` contained in the context. An example of this object structure is shown in figure 2.2. The transformation of the specified components is executed after the initial creation of the necessary DBTypes for the `GraphModel.ecore`, so that the inheritance relations can be defined.

The graphmodel, which is extracted from the injected context, is analyzed to its nodes, edges, containers and user defined types. The transformation process is executed in multiple iteration cycles over the object structure, which are located on different abstraction layers, since the needed information is variable accessible. A graphical illustration of the transformation is shown in figure 3.3. Every negotiation begins with the DBType creation, by inscribing the defined name of the MGL ModelElement to the DBType and whether it is abstract. Since this step only requires the name of the defined type the transformation process iterates over each instance of the ModelElement type of the MGL. This includes every node, edge nodecontainer and the graphmodel as well as the user defined types but excludes the enumerations which are not needed to be stored in the database. The second iteration cycle regards to the ModelElement instances as well, to transfer the defined attributes of the currently created types. This attribute declaration has to be done consecutively in a discrete step, since the relations between different types can cause circular dependencies, which are explained in the previous section 3.1.1. Each attribute of a met ModelElement is transformed into a new DBField. The defined name of the MGL's Attribute is set as the DBFields name. The data type of the defined attribute is based on the textual representation of the EMF `EDataType` and supports more specifications as the DyWA. The mapping table for the EMF EDataType to the DyWA `Property` assembly is shown in table 3.1, which demonstrates the necessary simplification of the data types. In addition the attributes are separated on their complexity. The DyWA differentiates be-

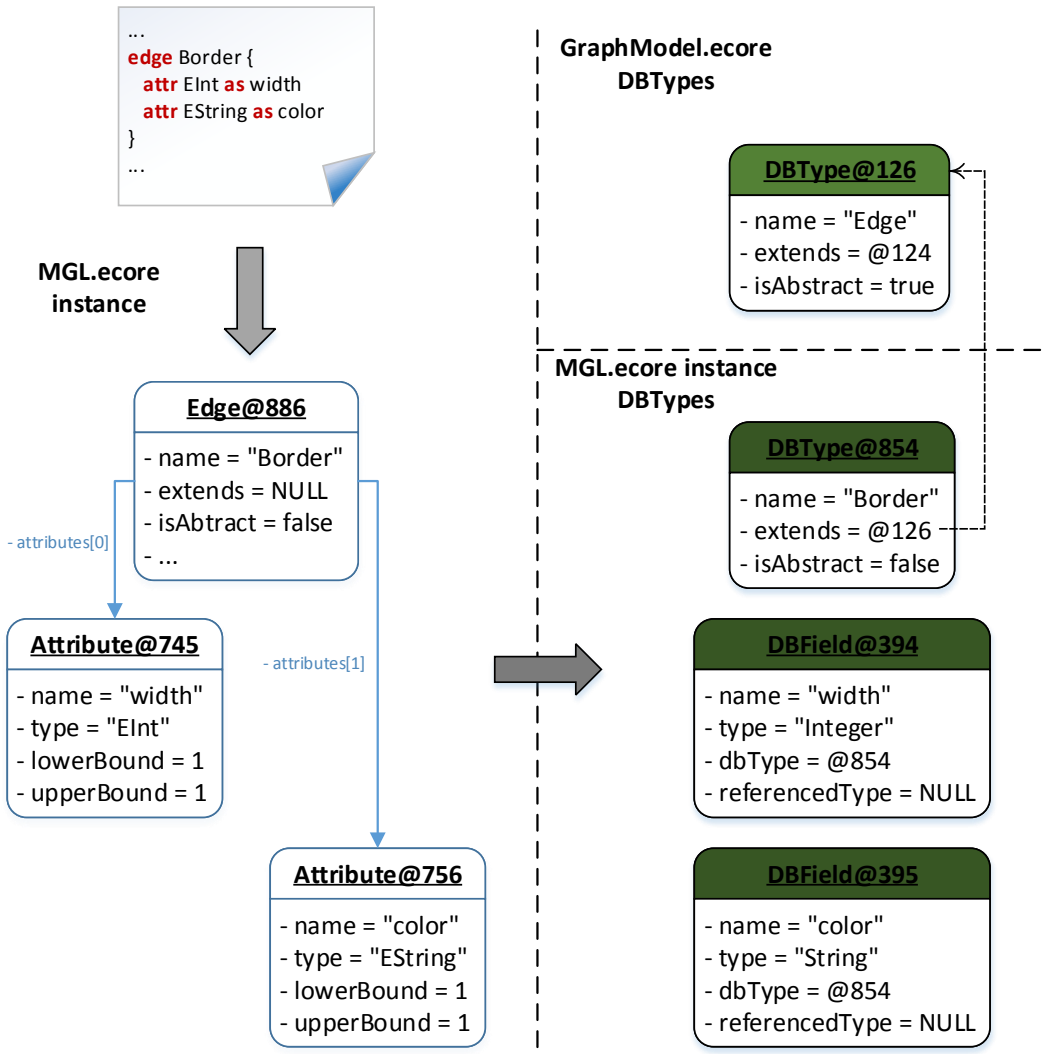


Figure 3.3: `MGL.ecore` instance model transformation.

tween the primitive DBField, which constitutes only the shown DyWA properties and the complex DBField to realize references between different DBTypes as well as multiplicity of references. Another important point is the observation of the `lowerBound` and `upperBound` attributes of the MGL Attribute instances, which identify the attribute as a list or single value. Since the DyWA only provides lists of strings as a primitive DBField, the accuracy of the attribute values is decreased. In the alternative to the primitive lists is the definition of user defined types, which enables lists of complex types which are validated type-safe. The last iteration cycle differentiates every component on the lowest abstraction layer and reproduces the inheritance relations between the predeclared DBTypes. This step is located on the definite components in the CINCO `MGL.ecore` to guarantee the inheritance only between components of the same type. To enable CINCO features like prime refer-

ences, the imported meta models are transformed to DyWA DBTypes as well, but without any further functions to handle the instances of this types.

At this point the transformation of the `MGL.ecore` instance into DyWA DBTypes is finished. The depending calls on the TypeController, which were mentioned before, are executed before the domain generator of the DyWA is triggered. This guarantees that the domain specific DBTypes are read out and generated to specialized entities and controllers, which are used in the generated code of the Pyro Modeling Environment. In addition to the DBType creation it is necessary to port the meta model to the front-end as well, to establish the properties view. This is realized by a transformation of the MGL components to a *JSON* [**?**] codification, which represents the defined meta model as well. The detailed explanation of the process to encode the JSON objects is skipped, since the process is almost analogous.

MSL.ecore instance reproduction

The `MSL.ecore` describes the possibilities of the visual representation of a component, which is placed on the canvas. The different shapes, graphics algorithms and appearances are explained and exemplified in section 2.1.2. CINCO is transforming and generating a meta model for the defined styles as well as the default initialization, which represents the defined appearances defined in the MSL instance. This approach enables CINCO to change and modify the appearances of a component after its initialization, but requires to store the entire style information in parallel to the business model. Pyro instead pursuits the approach to store the style information only in the front-end, which signifies that no further DBTypes are needed to describe the appearance of the components. Since the Pyro Modeling Environment is based on *JointJS*, the defined shapes has to be transformed to a *SVG* [**?**] markup and the appearances to SVG-Attributes. The detailed transformation is shown in figure 3.4. As shown in section 2.1.2, each component defined in the MGL is dedicated to a style in the MSL. Before the related style definitions are assigned, the MSL instances are loaded from the context, based on the import statement located in the graphmodel of the MGL instance. The different node and edge styles are read out in order of the defined components in the MGL. The transformation process iterates all components, gropes about the style annotation and loads the style from the imported MSL file. The style itself is transformed recursive in level-order of the appearing shapes, since the shapes of a style can be arranged in a tree similar illustration, to enable hierarchical shaping. This process is run twice to firstly generate the SVG markup and secondly the SVG attributes, which has to be defined separately.

The markup, which represents a complete style definition is based on the SVG tags, which are used to represent a given shape of the defined style in the MSL. To enable the scaling, rotation and movement of a defined in JointJS the different tags are embedded into

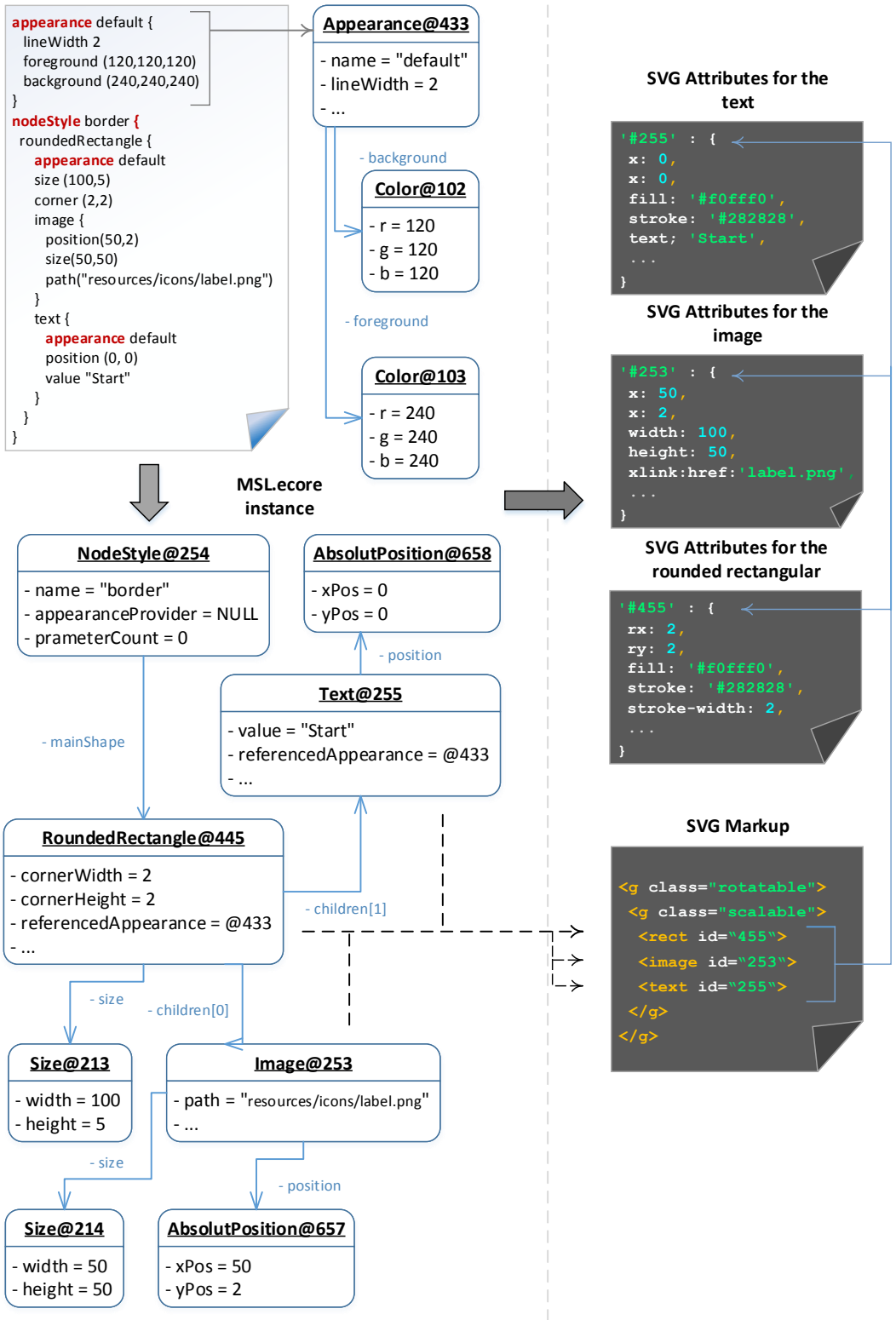


Figure 3.4: `MSL.ecore` instance model reproduction.

two `g` tags, which are container to group the other tags. Attributes applied to the `g` tag are inherited by child tags. The outer `g` tag is extended by the attribute `rotatable`, which enables the rotation of all child elements in the canvas by a specifying angle. The second SVG container enables the smooth and relative scaling of all child tags. The movability, which is provided by JointJS, is enabled by the additional attribute `movable`. Every SVG tag and its children, which are extended by this attribute are movable by dragging in the canvas. The inner tags of the markup describe the single shapes which are read out from the style. Every shape is translated into a SVG tag and extended with an unique identifier to bind the separated appearance description in the second iteration. The available MSL shapes and the translation into SVG tags is shown in figure 3.4 as well.

In the second iteration, the defined appearances of every shape are read out. CINCO provides two facilities to define an appearance, which has to be separately picked. On the one hand the *inline appearance*, which is defined directly in the shape and on the other hand the *referenced appearance*, which is related to an appearance defined in the global scope of the MSL file. Independent of this difference, appearances are extendable by inheritance. As a result of this the second transformation process of the MSL reproduction rebuilds every appearance of the discovered shapes, by solving the path in the hierarchy of inheritance. The fully reassembled appearance is transformed into SVG attributes, which are shown in the upper right part of the 3.4 figure. A special handling is needed for the `text` shape, since the value can be specified with placeholders. In this case the content of the SVG text tag, which is defined in the SVG attributes as well, is linked to a label provider, which reads the attribute values and writes them to the text tag's content.

The `MSL.ecore` of CINCO is quite equivalent to the SVG markup creation, so that the model transformation nearly becomes a one-to-one mapping. The appearances and SVG at tributes both offer similar possibilities of element styling. Afterwards the transformed styles of the elements are converted into a JSON representation, which extends the prototype of its depending business model, previously generated from the MGL instance.

Pyro-specific meta model

CINCO uses the file system, provided by Eclipse, to handle the different projects, folders and files in a CINCO Product. The Pyro Modeling Environment has to be extended with additional DBTypes shown in figure 3.5 to rudimentary cover the features of a desktop application like the CINCO products. The DyWA provides the ability to store user information, so that the different developers can be assigned to different graphmodels to work on. Similar to Eclipse the files can be combined in a project which is dedicates to an user, who creates the project and models. The project DBType is created and extended with a name, the date of creation and a complex attribute to store the reference to the user and the list of contained graphmodels. In spite of the multiuser abilities of Pyro an additional

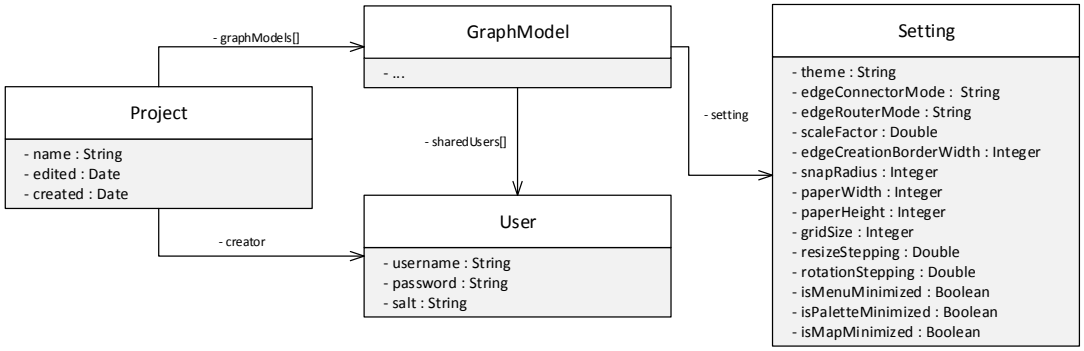


Figure 3.5: The additional DBTypes for the Pyro Modeling Environment.

list of references to share the project with other users is realized. To enable modifications of the user interface, which are persistent after a refresh, the settings of the Pyro Modeling Environment are stored in the database. The settings embrace multiple different look and feel modifications like themes and the individualization of the entire surface by element minimization and maximization. In addition, the adjustability of more advanced settings are provided like sizes of the clickable areas. Every setting is added to the `Setting` DBType to enable complete individualized modes of operation.

The multiuser support as well as the redo and undo functionalities are based on the command pattern, which is realized in the Pyro modeling environment. The command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters [**?**]. In the scope of Pyro the different methods are dedicated to create, update and delete based on the modeling components. Every command is parameterized with an contextual information like the position, the changed value and further information of the used method. This includes additional declarations to determine an embedded creation or which kind of update action is used. The available commands, which are mapped to DBTypes are shown in figure 3.6.

To store every command in one list aggregated to the graphmodel, the `PyroCmd` is defined, which represents the super type for all further commands. Every command depends on the affected element and an index, to determine the order of execution as well as combined commands. When a node or a container is dragged to the canvas a `CreateNodeCmd` is created. The position, where the element is placed and the underlying container are recorded in the command's attributes. This information is needed to roll back the action as well as to re-execute the action. For the container, no particular command is required, since the container differs in the ability to contain elements, which cannot be existent on creation. The move and remove commands for the nodes and containers require the

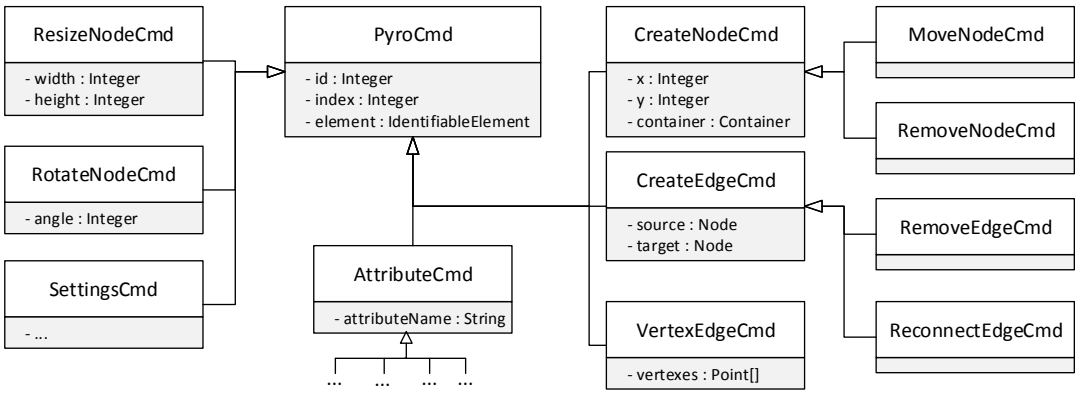


Figure 3.6: The different Pyro DBTypes for the command pattern realization.

same attributes to store the changed values of the element. The moving of an element can change the position and eventually the underlying container, so that both information are necessary. The remove command required this information as well, if it is inverted and rolled back. In this case the last position of the element and its container for the re-creation can be read out. The edge depending commands are structured similar to the node and container commands, despite the attributes of the `CreateEdgeCmd`. When an edge is created the source and target of the edge are set. The same attributes are changed, if the edge is reconnected. By the same token like the `RemoveNodeCmd`, the `RemoveEdgeCmd` is extending the `CreateEdgeCmd`. Since the move and remove command are not introducing new attributes, they are only marker interface to differ between the command types for further interpretation. The `AttributeCmd` is extended for each attribute of the defined components of the MGL to record every change of the attribute values. The attribute command extends the Pyro command as well and stores the attribute's name and its current value. To realize the redo and undo functions without redundant data, the previous attribute command has to be determined in case of a undo. For the resize, rotate and vertex commands, the same mechanism is used to reconstruct the history of an element's modifications. In further versions of Pyro the commands should be combined in *burst actions*, which are considered atomic for the developer.

The detailed mechanism of the redo and undo functions as well as the user synchronization are explained in section 3.1.2. At this point the entire meta model of Pyro is defined and reproduced in DBTypes. The DyWA domain generators are triggered to generate the domain specific entities and controllers. Both are describing the first lowest layers in figure 3.1, which are the base of the further code, generated by the Pyro Meta Plugin.

3.1.2 Code Generation

As illustrated in figure 3.1, the third layer contains the Transformation API and the Commands. This layer and its succeeding are completely generated by the Pyro Meta Plugin and permit the core functionalities and interfaces for extension points of Pyro. Every generated implementation is domain specific and totally typified, threw the defined components.

Transformation API

The Transformation API offers domain specific interfaces to simply control the generated entities. This API is implemented in CINCO as well, which provides the abilities to port already existing code to the platform of Pyro. The API includes well typed methods to modify the values of the defined attributes as well as special methods, which are developed based on the embedding and connection constraints. The idea of the Transformation API goes back to the wrapping of the entities, to enable the possibility of hooks and the command handling. To facilitate the wrapping of each domain specific entity, the given foundation of the entities is wrapped as well. This results in the generation of the interfaces for the types of the `GraphModel.ecore` as well. The generation process consists of one iteration, which creates an interface and the corresponding implementation class for each entity. The created classes are named equally to the types with a preceding C to differ between the generated entities and for the symbolic connection to CINCO . The generation of one entity wrapping includes multiple methods which depend on the given type. First, the get and set Methods for the attributes and the wrapped element are generated for every defined component. Additional methods for the determination of the root element and the enclosing container are generated as well. The implementation of this methods is generated as well and considers the correct typification with regard to the embedding constraints. Depending on the type, different methods are appended. Edges are extend with methods to directly reconnect the source or target and add bending points. The methods are depending on the given connection constraints, to prevent misuse and undesired linking. For example a generated Transformation API class for an edge, is only extended with a method to set the source with the parameter type of a node, which holds the edge type in its outgoing edges list. The nodes and the containers of the Transformation API, offer methods to access the successive and preceding nodes, which can be connected. The generated methods are restricted to the determined types, defined in the connection constraints. The same limitations are kept for the containers and graphmodels, which are extended by methods to obtain, set and create the containable elements. In addition to this, the interfaces of the Transformation API provide the abilities to execute the primitive operations on the elements, like moving, re-sizing and rotating. In summery

the Transformation API provides all necessary methods to create, read, update or remove elements with coherent typification to the defined elements and the constraints.

Commands

To offer the possibility of redo and undo functions as well as the multiuser support, Pyro is employing the command pattern to meet the goal. Every influence of a user, who models with the defined types is serialized depending on the kind of the modification and logged in a command. The commands are stored sequential in relation to the graphmodel which is affected. In addition to this the graphmodel is extended with an index of the actual position in the command list, according to undo actions which decrease the index. The meta model of the Pyro commands is shown in figure 3.6. The commands are created at the end of the corresponding methods in the Transformation API. When a new node is created as an example, because the developer dragged it to the canvas, the according method gets called. At the end of the execution of the method, the matching create command for the node type is produced and stored, including the significant information, like the position and the underlying container. The list of commands symbolize a log of all actions done by the developers or any plugin using the Transformation API like hooks. To redo and undo the commands, a command handler is realized, which deserializes and inverts the commands. The command handler is part of the Transformation API and offers multiple methods in the graphmodel interface. As a result of this, the commands can be re-executed or inverted to realize undo functions.

Figure 3.7 shows the sequential processes of the redo and undo functions, depending on the layer structure of Pyro. The undo function is triggered on the Transformation API graphmodel interface (`CGraphModel`). The command handler determines the last command, which was executed and converts it to its opposite. The converted information describes the state before the command was executed. After the reversal the converted command is executed directly on the graphmodel to not create further commands in the undo function. The index of the commands is decreased and the affected element is returned. The last step is necessary to possible synchronization with the element on the canvas. When a command has been undone, the next action, which is not the redo action, causes the deletion of all successional commands which has been undone before. This linear undo offers the possibility to return to the beginning of the development. The redo function works quite similar to the undo function. Instead of the actual command, which can be determined by the actual index stored in the graphmodel, the succeeding command is identified if it is available. Afterwards the command is executed and the command index is increased.

The presented realization of the command pattern of Pyro offers the possibilities of the multiuser support. The synchronization of the developers graphmodel on the client side and

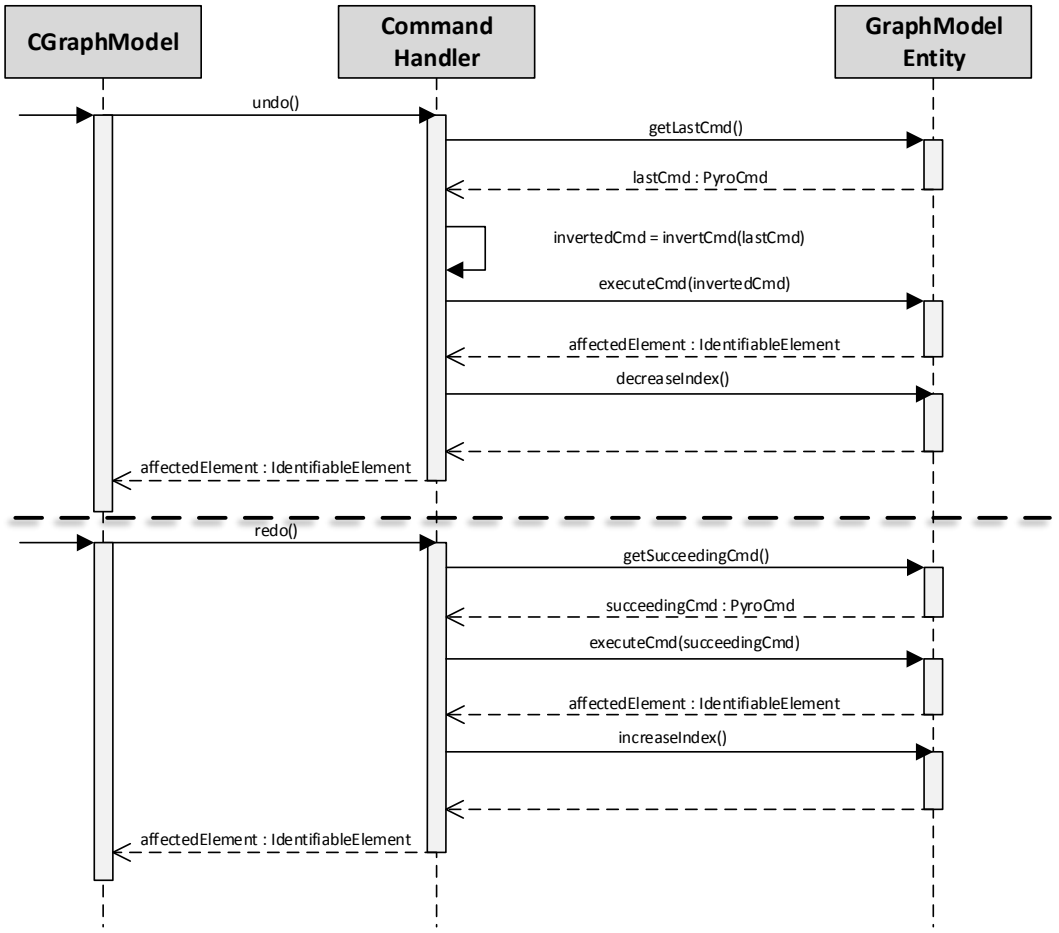


Figure 3.7: Sequential processes of the redo and undo functions.

the graphmodel stored on the server, which is modified by all involved developers, is shown in figure 3.8. Every user which views a graphmodel receives the actual command index, when the model is loaded. The synchronization is done chronology in the background. The front-end sends a request, including the actual index of the viewable model to the server. After the deserialization of the received message, the graphmodel interface of the Transformation API is called to synchronize the client. This causes a read out of all commands, which has been missed by the client depending on the transmitted index. The command handler determines the affected commands, serializes and returns them to the Transformation API. The responded list of commands is executed on the client side graphmodel to establish the same state on the client side. Last, the transferred index is actualized to be send in the next synchronization step.

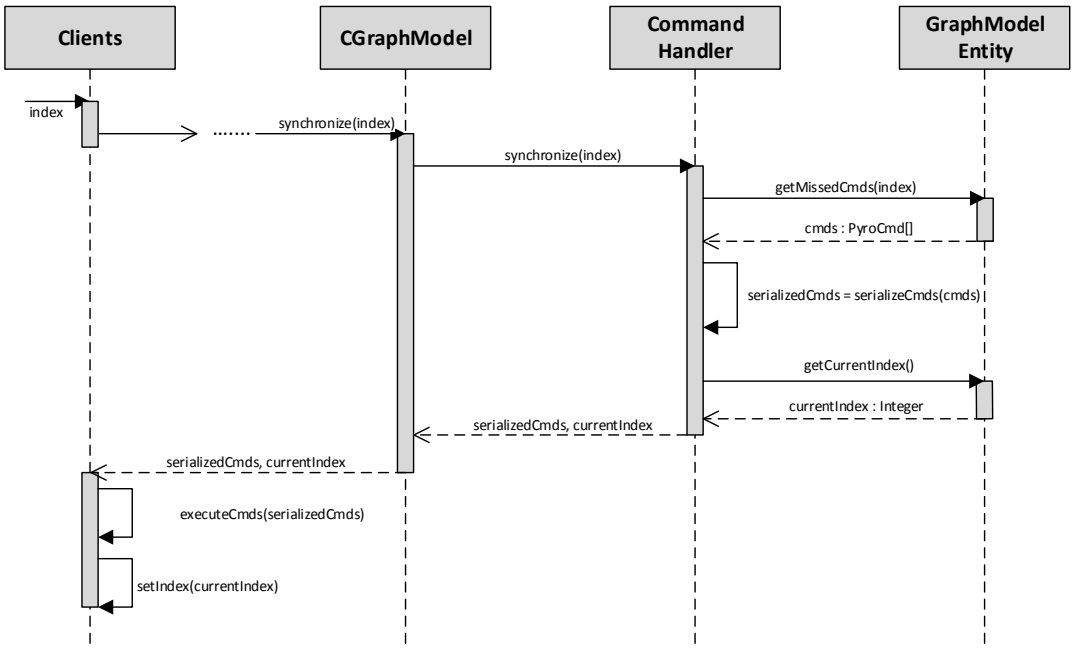


Figure 3.8: Sequential processes of the multiuser synchronization.

3.2 Modeling Environment

The Pyro Modeling Environment is designed in according to the CINCO Product's abilities. The DyWA application of Pyro, which is generated by the PMP, consist of two pages. The project explorer shows all projects of the registered developer in a sortable table and offers the possibilities to create, edit or view a project. The main page of the PME is the editor page including the canvas. The available parts of the editor are related to the CINCO product. The detailed explanation of the pages is shown in section 3.2.1, especially the available areas of influence for the developer.

The second section describes the communication between the client and the server during the development of a model. First the different simple actions will be declared, which comprehend the CRUD functionalities as well as actions depending on the graph model domain. In particular the live and delayed validation of the editor will be pointed, using a concrete example.

3.2.1 User Interface

The first page shown to the developer is the projects explorer shown in figure 3.9. The projects are listed in a table for a clear arrangement. The table consist of multiple columns, which display the defined name of the project, the date of creation, the last edit and a

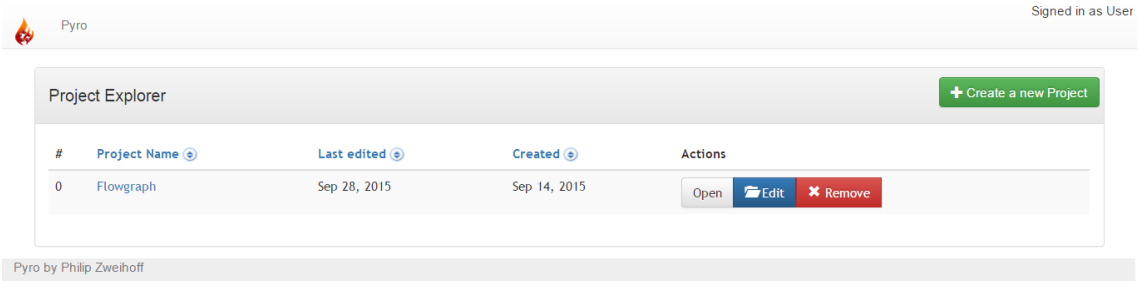


Figure 3.9: The Pyro Modeling Environment Project Explorer.

button group. Every columns despite the *Actions* columns is sortable. The button group contains three buttons with basic functions to open the project, edit the name or remove it. If the project is shared with other developers, the color of the row changes to blue. The project explorer represents a simple overview over the involved projects and is not overcharged with unnecessary functions.

The editor page (shown in figure 3.10) is loaded when the open button of a project is pressed. The editor is structured into four main parts. Every part can be modified by the minimize and maximize buttons in the upper right corners of each panel. This customization posses individual employment for every developer, since every setting is stored for the current user.

Menu

The menu of the editor offers the main functionalities, which are known from multiple desktop applications. The *File* button opens a dropdown menu for the project administration. The developer can create a new project or edit the name of the current one. In this menu, the return link back to the project explorer is located as well as the sharing options. The sharing menu, which is opened in a modal, provides a table of each developer, available to share the project with. The list of developers can be extended by typing an email address, which leads to an invitation of the corresponding developer to join the project. The creator of the developer can decide, which other developers to collaborate with on the opened project. Afterthought, the modal can be used to amend the prior decisions and remove a contributor from the sharing list.

The next two buttons showing the current modeling mode, which can be on the one hand the default *Move* and on the other hand the *Select* mode. The selection enables the developer to mark all elements in the drawn boundary of the selection. This feature enables copying, removing, cutting and moving of multiple elements at once. The *move* mode, induces the default behavior of the cursor. Elements can be moved by dragging and edges can be sustained. To visual differ between the two modes, the cursor changes its representation in the canvas depending on the selected mode.

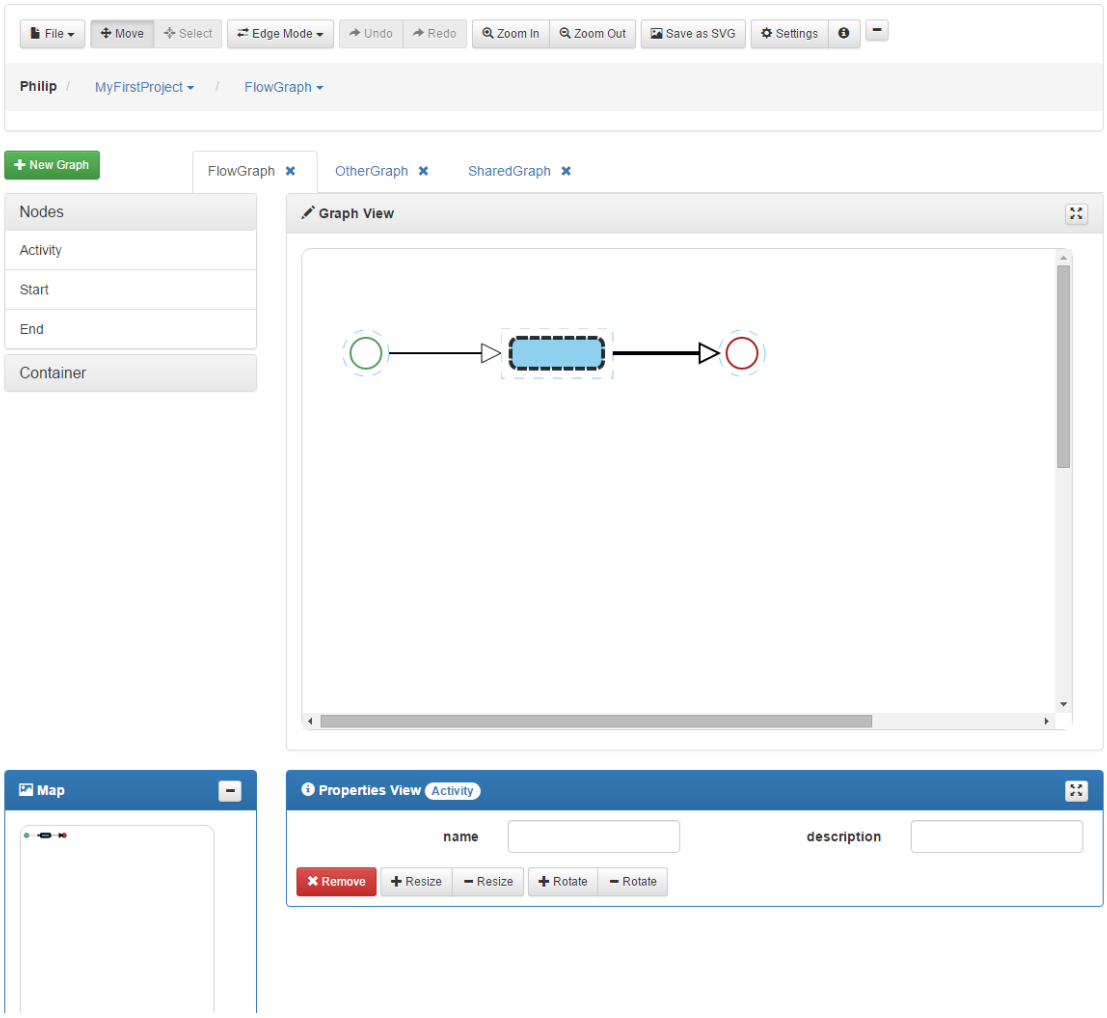


Figure 3.10: The Pyro Modeling Environment Editor

The *Edge Mode* offers nine different variants of the visual representation and the behavior of the existing edges on the canvas. Every edge mode consists of a routing algorithm and a vertex design. The vertex design determines, how the bendingpoints created by the developer are rendered. The default *normal* design causes an angled vertex, exactly at the position of the bendingpoint. The *rouded* vertex design renders the edge with rounded vertexes instead of the angled ones, but the bendingpoint position is still on the edge. The third vertex design *smooth* changes the exiting bendingpoint positions and unclenches the edge itself. The resulting representation is similar to a *Fourier Transformation*. Based on the normal and rounded vertex designs, three routing algorithms are combined. The routing algorithms try to find a path between the connected nodes without colliding with other elements on its way. The difference between the given algorithms is the resulting edge behavior. The *orthogonal* routing provides only rectangular edge-courses. The *manhattan* routing is similar to the previous, but tries to size the edges in a way that two lines

connected by a vertex have potential differences in size. Instead the *metro* routing builds an edge with only angle bisecting vertexes.

The *Undo* and *Redo* buttons are located analogous to known editors and enable a linear traveling in the user action history. In further releases, the history will be viewable in another panel as well, to offer the possibilities to directly jump to a desired version of the graphmodel. The remaining buttons offer the possibility to change the scaling of the elements in the canvas, to get an overview or a very detailed insight of the graphmodel. As a result of the SVG structure behind Pyro, an export function is provided to embed the model as a graphic in other documents. The *Settings* button opens a modal dialog with the core settings of the editor, like the size of the paper, the grid resolution and the theme for personalization.

Under the menu bar a breadcrumb navigation is located. This navigation enables the user to retrace the structure of the project as well as the listing of all the created models. The breadcrumb is a combination of a displayed path known from file management systems to show the current opened file location and the tree view of a project, which is used in nearly every IDE and file system as well. The breadcrumb navigation offers the ability to switch between the different projects and the contained models in a compacted way, in contrast to an outsized tree view. In addition to, the models in the project are displayed as tabs above the canvas as well, to quickly change the model to edit. Next to the tabs on the right side, a button is located which offers the ability to create new graphmodel instances in the opened project. The opening modal dialog requires a name for the model and the type, which has to be instantiated. After the model is created, the depending tab is opened.

Palette

The palette is located on the left side of the editor. The composition and the utilization are similar to a Cinco product. Every defined group in the `@palette` annotation of a component in the MGL, leads to a group in the palette. For non grouped components, the overall *Nodes* and *Containers* groups are provided. Every group contains the components with the suitable annotation value. When a group is clicked, all components are displayed by their name and can be dragged to the canvas. Besides the create feature is disabled for the component, then it is not displayed in the palette. As an exception the prime references are included in the palette as well, but every prime reference type is constructed in its own group. The displayed identifier of the prime reference is not the corresponding component name, but the value of the defined prime label. In contrast to Cinco , the Pyro palette is restricted to one expanded group at a time, to increase the general survey. When another group is opened, the other one is automatically closed.

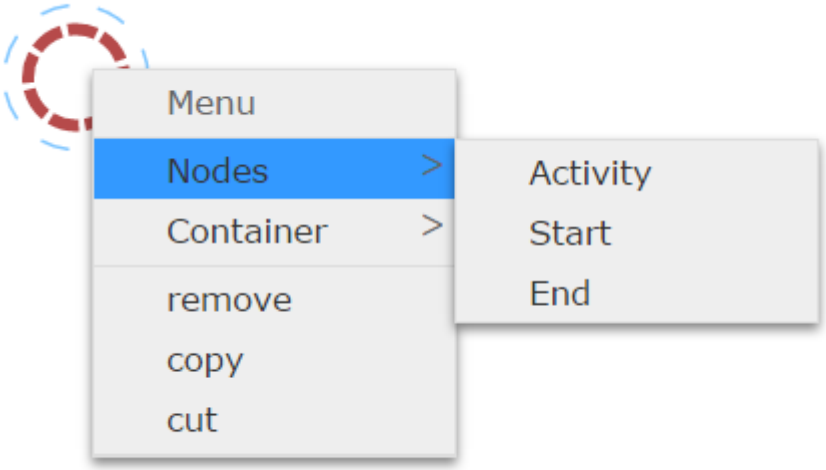


Figure 3.11: The Pyro Modeling Environment context menu.

Canvas

The canvas constitutes the major part of the editor. It is designed to simplify matters and support the developer to quickly and intuitively create models. This goal is met with multiple menus and visual reacting items to increase the usability. The components can be created by drag and dropping the palette entries on the canvas. The element is immediately rendered on the cursors position. By click and hold the visible shapes, the element can be moved on the canvas, besides the movement feature is disabled in the MGL instance. Another way to create an element in position is the context menu (shown in figure 3.11), which can be enabled by right clicking. The context menu contains multiple actions like the known cut, copy and paste options, but also the same groups as the palette on the left. Every defined group pictures a entry, which is the title for a sub menu. The sub menu holds every component by its name in a sub menu entry. When a menu entry is clicked, a new element is instantiated in position of the initial right click. Besides, Pyro offers a hover menu (see figure 3.12) for every node and container in the canvas, which shows up when the cursor is located above the shapes of the element. The hover menu provides actions to scale, rotate, remove the element from the canvas or display the depending properties view.

To provide the ability to quickly and easily create edges between elements, every element is encircled by an small blue few transparent border shown in figure 3.12 as well. This border represents the clickable area to draw an edge to another element. At the moment of the edge creation, all connectable elements are highlighted, to simplify the linking. In addition to this, the edge snaps to the connectable element when the cursor is close to the target. This feature provides the ability of connecting elements that are quite small

Figure 3.12: The node and container hover menu.

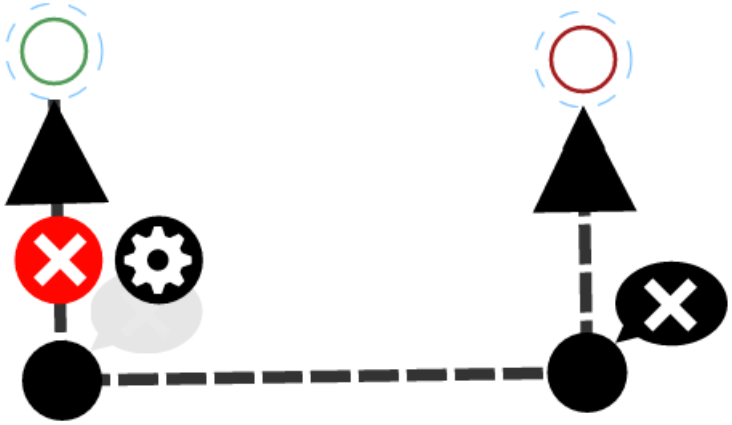
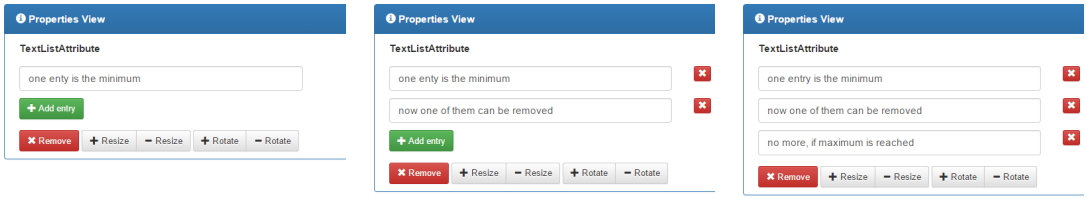


Figure 3.13: An edge in the canvas, displaying all user assistance.

or zoomed out. According to the edge creation, the developer can add bendingpoints by just clicking the edge at any location and stretching it into position. When the cursor is located over an already existing bendingpoint, a big dragable point is displayed to simplify the repositioning as well as a speech bubble containing an remove sign. By clicking the speech bubble, the developer removes bedingpoints to tighten the edge again. A similar behavior is shown at the source and target connectors of the edge, which also enlarge to be simply dragable for reconnecting. To remove the entire edge from the canvas, another red speech bubble is shown when the cursor is located above the edge. Next to the removing speech bubble, a gear sign offers the trigger to display the properties of the edge and highlights it. All user assistants available for an edge are pictured in figure 3.13.

Properties View and Map

The properties view of the editor is located beneath the canvas. When a element is selected and highlighted, the properties view for this element is created, depending on the elements



(a) Least entries **(b)** Amount in between **(c)** Most entries

Figure 3.14: Dynamic form to display list attributes.

attributes. The Pyro Modeling Environment creates a form, which consist of input fields and labels, displaying the attributes. Character devised attributes like strings, floats or integers are displayed in text fields, but extended with type-safe validation. The non textual attributes like booleans are rendered as checkbox input fields. To display an enumerating attribute, a choice field is used, which shows all possible and the selected literal as well. Since CINCO provides the ability to specify every attribute as a list of values with cardinalities, Pyro supports dynamic forms (see figure 3.14). Dynamic forms are extendable by clicking the plus button, to get another input field. Every input is removable by the red buttons on the right. The number of fields is validated at every time the amount changes, so that the add button disappears when the upper bound is reached (see figure 3.14c), similar the remove buttons when the lower bound is reached (see figure 3.14a). Any modification of the input fields causes an event, which is send to the server for synchronization. This synchronization is realized asynchronous to not be an encumbrance to the developer.

On the right to the properties view the map is located. The map displays a minimized overview of the entire canvas. When the zoom function of the menu is used the map is updated as well. To display the properties view in full width with three columns, the map can be hidden. In addition to the input fields for reading or updating attribute values, the properties view is extended with a button group at the bottom. The buttons reflect the functionalities of the hover menu, like remove, resize and rotate to enable modifications in every place.

3.2.2 Validation and Synchronization

The Pyro Modeling Environment is based on an event handling mechanism, which notices all actions done by the user. This includes clicking, dragging, moving and profound events triggered by Pyro itself. The events are caught by multiple event handlers, so that Pyro reacts depending on the performed action. Independent of the type of the event, an asynchronous communication between the client, who comes up with the action and the server is fulfilled. This information exchange, conduces the synchronization between the graphmodel displayed on the client and the one on the server side stored in the database.

Before the data on the server is updated and before the action is performed, the action is validated and in case of a constraint violation rolled back.

The constraints which has to be considered are defined in the MGL instance and generated into ruling methods for validating. The actions which has to be validated are limited to the create action, the move action and the reconnect action for edges. The create and move actions are constricted to the embedding constraints, which define how many elements of a type can be contained in the parent element depending on their types. As presented in section 2.1.1, the constraints can be defined in sets of element types combined with a upper and lower bound. This boundaries depending on the contained types are validated before the element is created in a container or moved into it. As an exception, the creation of an edge is not validated depending by the embedding constraints but on the connection constraints. The connection constraints are defined similar to the embedding constraints, including sets and cardinalities. The validation is realized in a different way, since the possible edge types depend on the chosen type of the source and target elements and their current already existing connections. As a result of this, the edge creation or reconnection are restricted to the constraints. A known problem, which is also present in CINCO products, is posed by the fact that the lower boundaries of the embedding and connection constraints cannot be complied but only observed. As an example, this problem is raised when a container with lower embedding constraint boundaries is created and no elements are contained. At this point the constraint violation is only displayed to the developer, but cannot be complied.

In spite of the possibilities of modifications by hooks or other plugins, the validation is executed on the server side as well, generated into the relevant methods of the Transformation API. The sequential process from the create action of a node and an edge, performed by the developer to the validation and the response is shown in figure 3.15. The node creation event is triggered when a node is dragged to the canvas form the palette, or the entry of the context menu is clicked. The event handler receives the action, including the element to create, the position and the container underneath. Since the graphmodel is the foundation of every element, the container is always defined. The event handler triggers the validation, which checks the embedding constraints. This process is generated depending on the MGL instance and counts the containing elements grouped by their types to compare the calculated amounts to the defined constraint cardinalitities. When the validation succeeds, the action is propagated to the server, which validates the creation as well. Every request send to the server, results in a response, which displays the positive or negative outcome of the execution. In case of a invalid classified response, the performed action is rolled back to the prior state before the creation. This supposes the deletion of the currently added element.

The edge creation, which differs in a few points, is shown in the lower part of figure 3.15. The event is thrown when the developer draws an edge near to an element, just

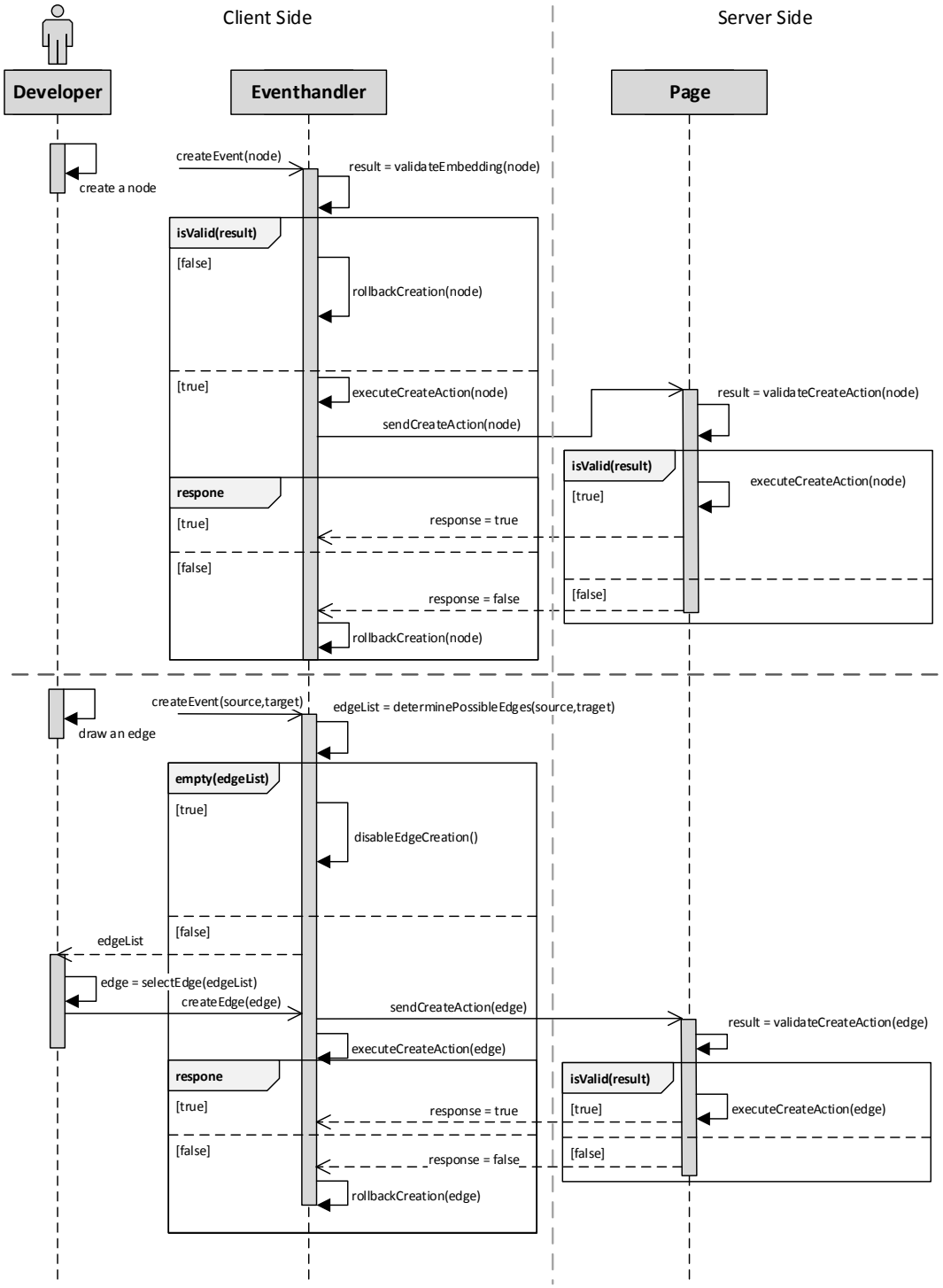


Figure 3.15: Client server interaction and validation of the create action.

before the edge would snap to the element. The listening handler obtains the source and the potential target element. Depending on the defined constraints and the current outgoing edged of the source node and the incoming of the target node, the possible edge types are determined. This list of edge types is displayed to the user in another context menu, if it is not empty. In this case the drawn edge is not snapped to the near by element and cannot be connected. When the developer selects an edge type of the list, the callback sends the edge create request to the server. At this point, during the waiting for the servers response the edge is already created. In the meantime, the request is handled and validated on the server as well. Similar to the node creation, the result of the server adjudicates on the edge be conserved. This two step validation is necessary, because of the influence of other plugins, hooks or other users, editing the same model. In this manner the consistence and correctness is guarantee.

3.3 Planned Features

The currently realized editor yields a graphical user interface with basic functionality to create, modify and remove elements including additional environmental features to individualize the behavior, the look and feel and a project explorer. Every element is rendered in a SVG as specified in the MSL instance. Furthermore, the generated Pyro Modeling Environment executes a two step modelvalidation and support multiuser editing.

Despite these features, the PME is not finished at all. To improve the useability of the editor the implementation of additional features is required. This section provides an overview of planned, but not yet realized features of the PME, which will be domain specifically generated by the Pyro Meta Plugin. The following collection consist of features that are on the one hand already realized for the CINCO products and need to be mapped, on the other hand new features that will take advantages of the rich internet platform of Pyro.

Problems View

The Pyro Modeling Environment is restricted to the constraints, defined in the MGL instance. This constraints specify the validation process, when the corresponding action is performed. As mentioned in section 3.2.2, the lower bounds are not considered, when they are not crossed once, which would then disable the remove feature. To display the lower boundaries constraint violation to the developer, a problems view should be created. All connection and embedding constraints are validated after an action of the developer takes place. The resulting violations are collected and used to update the problems view. This behavior is similar to the problems view present in Eclipse, which is already used by the CINCO products. The problems view itself will be structured like a table for a clear arrangement and to enable the developer to sorting and filtering functions. Every collected

incorrectness is displayed in its own row. The row is separated into multiple columns, which will at least display a short and a detailed message, the type of the assigned element and the constraint that was violated. When the row is clicked twice, the element which causes the problem is highlighted and focused on the canvas. As an example an embedding constraint, which defines that a container named `Bucket` has to contain at least one node of the type `Wastepaper` is validated. When a Bucket container is dragged to the canvas, the validation is triggered as shown in figure 3.15. But the process is extended with a succeeding validation, which considers the lower bounds of the embedding constraints, of the available containers on the canvas. The Bucket instance is discovered and validated, which causes an error to be displayed in the problems view. Afterwards, when a wastepaper is added to the bucket, the process is triggered again, but this time doesn't determine an error, so that the entry of the problems view is cleared.

Another use case for the problems view is the usability, of additional plugins located on the server. The periodic update cycle, which synchronizes the model on each side could also trigger an extended validation. The results can be responded to the server and then update the problems view. This provide an opportunity to validate other constraint like OCL [?] queries as an example and illustrate the results, so that even more verification can be realized.

Model import and export

Since CINCO is open source and CINCO products are already created and under development, the possibility to export already existing graphmodel instances to the Pyro Modeling Environment could be quite suitable. To meet this goal the Pyro Meta Plugin has to be extended to generate another Eclipse Plugin, which is included in the CINCO products. This Pyro Export Plugin is able to read out the present graphmodel instance located in the Eclipse instance of the CINCO product and export it to Pyro. It is imaginable that the exportation is realized in two different way, to offer great flexibility.

First, the existing graphmodel, which should be exported, is serialized into a certain file format and then uploaded to Pyro. The transferred file will be deserialized and then interpreted as an instance of a graphmodel, which is stored afterwards. The other concept is based on remote calls from the Pyro Export Plugin on the CINCO side to Pyro located in a server. This remote calls can by realized by a *Representational State Transfer* (ReST) [?] webservice, which has to be provided by Pyro. This webservice enables the external calls of methods placed behind HTTP requests. The necessary parameters can be passed by GET or in the POST content of the request. This way facilitates the transfer of an entire graphmodel instance without any file exchange done by the developer. In addition to this, the remote calls of the Pyro Export Plugin can be used to realize a unidirectional synchronization of the Pyro server and the CINCO product desktop application, which

results in a remote desktop application. Afterwards, the ReST webservice of Pyro could be used to pull the model information from the server, which would realize an import function back to the CINCO product as well.

Multiple MGL instances

At this point, only one MGL instance at a time can be read out, transformed and generated by the Pyro Meta Plugin. The resulting Pyro Modeling Environment is just capable of creating the one type of a graphmodel defined in the MGL instance. As a first step the imported Ecore files and its containing types are transformed as well, to be used in the PME. But to provide the possibility to read out multiple MGL instances at once, the Pyro Meta Plugin has to be attached to the Cinco Product Definition (CPD), which associates multiple MGL files. Presently CINCO doesn't offer an extension point for meta plugins to be assigned to the CPD files. But when this interface is realized, the Pyro Meta Plugin can be modified to obtain the CPD file instead of just one MGL instance. The following transformation and generation process has to be extended for the iteration above multiple MGL instances. This results in a separation of the generation templates into two groups of local templates for each graphmodel and global ones, which are used once for the PME generation. Another important point is the meta model transformation, especially with regards to the forward declaration described in section 3.1.1. For this reason the entire meta model of all associated MGL instances has to extracted, collected and ordered to prevent errors caused by circular dependencies. After the transfomration and generation process the Pyro Modeling Environment provides the ability to create each of the defined graphmodels of the CPD. The developer obtains multiple different graphmodels to work on and Prime References can be used more accurate.

Notes

When a software is implemented, the developers use comments and annotations to accomplish legible and durable marking of their code. This results in an understandable way of employment and assists the other developers with the explanation of the author's train of thought. This facilities can be proved beneficial in the scope of modeling, when a note can be assigned to an element on the canvas. At this point, CINCO provides the possibility to define a MGL instance that offers a node type, which can be connected at least to all other nodes and containers but not to edges. This node is presented as a rectangular, displaying the text inserted in the properties view. The MGL generation results in a CINCO product, which enables the developer to add the note node to arbitrary other nodes and containers to annotate them. But the real advantage is taken, when the notes are realized as a core feature of every modeling environment.

In the PME, the notes would also be realized as notes, but adding the ability to be connected to every element including edges and the graphmodel itself. Once an element is created and located in the canvas, the developer can access the note in the hover menu which comes up when the cursor is located above the element. The note node is rendered as a simple text input field to include text and commentaries. In addition to this the note can be extended with annotations known from the JavaDocs [17] as an example. This annotations could provide the indication of the author, pre- and postconditions or a reference to another user, who is shared in the model. As a result of this the comprehensibleness of the model can be increased. The user references permit the developers to interact and discuss directly inside of the modeling environment, without the necessity of other additional tools.

Migration support

The migration support is the most important feature to be realized for the CINCO to Pyro transformation. At the moment, every diversification of an existing MGL instance, as an example the adding of a new node type, causes the necessity of a re-generation of the Pyro Modeling Environment. This regeneration cleans up the entire DyWA application, all DBObjects and the existing meta schema to create an unaffected environment for the new generation process. As a result of this, all prior information is lost. The migration support feature should enable the generation process with consideration to the existing meta schema and already instantiated DBObjects. To meet this goal, the Pyro Meta Plugin has to remember or experience which types are present in the Pyro Modeling Environment. This deduction can be realized by two different mechanisms. The offline solution is based on the documentation of the MGL instances after every generation process, which results in a review to the last execution of the PMP. The second approach uses remote access to the DyWA to read out the available meta model before the generation is done. In both cases, the former meta model is retrieved to consider the already existing types. The migration process obtains the two sets for the prior model and the current model and establishes the migrated target model. In addition, the native DyWA migration support is used as well, to avoid the complete erasing of DBTypes and its instances. This implies that the instance of the model, displayed in the Pyro Modeling environment still exists after the migration. The process itself is divided into three stages:

1. Insert new types of the current model, which are not available in the prior model to the target. The comparison of the types, to create the associations between the prior and current model are realized by unique identifiers provided by CINCO .

2. Mark the completely excluded types of the prior model as deleted in the DyWA. This results in the continuity of the types and its instances but prevents further instantiation.

3. The remaining types, which are available in both models are observed iterative:

   (a) When the name of the type has changed, the DyWA migration is used to change the name. The DyWA adapt all created instances to ensure consistence and renames them.

   (b) When the inheritance of a type is altered, the DyWA provides the ability to migrate this modification as well. The execution of this step requires the creation of new attributes for all existing DBObjects and mark the unneeded as deleted, so that they will be considered no more.

   (c) The defined attributes of a type are migrated in the same way as the types. New attributes are directly created and removed ones are marked as deleted. The renaming of an attribute is propagated to all DBObjects. An exception is the modification of the data type of an attribute. This requires the adaptation of every DBObject depending on the type containing the attribute. Since the data types are not compatible like integers and dates and cannot be converted to each other, the value is set to the default of the new data type. This is the only phase in this approach, which will cause a loose of information. To prevent this loss, the adaption of the affected attribute is skipped. The developer will be informed, that the migration process has to be able to convert the attribute values.

Another important point is the migration of the defined constraints. Since the existing instance located in the PME will not be changed during the migration process, despite the attribute data types are incompatible modified, the defined constraints can be violated. To re-establish the correctness of the model instance a second process is executed, which validates the available nodes, edges and containers. Each element, which is not conform to the constraints is signed to the developer as an invalid migration. As a result of this the migration process constructs a meta model which consists of the prior and the current model, but offers only the possibility to create further instances of the latest MGL instance. The instances are carefully modified to avoid the loss of preexisting information.

Chapter 4

Related Approaches

The previous chapter describes the concept, realizations and planed features of Pyro. In this chapter, related approaches for web based modeling tools are shown. The presented tools vary from diagram drawing interfaces to model driven code generators. The described tools are already released products, so that the available features are quite sophisticated. The following sections show a compressed introduction to every tool from the user's point of view. Every modeling environment distinguishes itself by different characteristics. The way, how the different approaches try to simplify the acquaintance of the developers with model driven development is described. After the initial description of each tool, the comparison with Cinco and Pyro is figured.

4.1 Lucidchart

Lucidchart [?] is a web-based modeling tool and offers the possibility to create multiple different diagrams depending on predefined types. The supported diagram types are limited to the upper-level grouping of flowcharts, organizational charts, multiple UML designs, website wireframes, mind maps, software prototypes and many more. The tool is based on current web standards like HTML5 and JavaScript, which enable the utilization on the latest modern web browsers. The user interface is shown in figure 4.1, which illustrates the rich internet application. The web application provides a palette on the left, which presents all available components based on the chosen diagram type. The palette can be extended with other components of different types as well. On the top, the menu bar is located, which offers multiple dropdown menus, known from different desktop applications. The menu bar enables the developer with common operations like modifications of the layout of the entire application, file management and a help. In order to a modeling tool, multiple arrangement options can be used, to position the elements on the canvas automatically. On the top of the canvas, the page menu is located to enable all known features to configure the text layout as well as colors and settings depending on the edges. The available functions

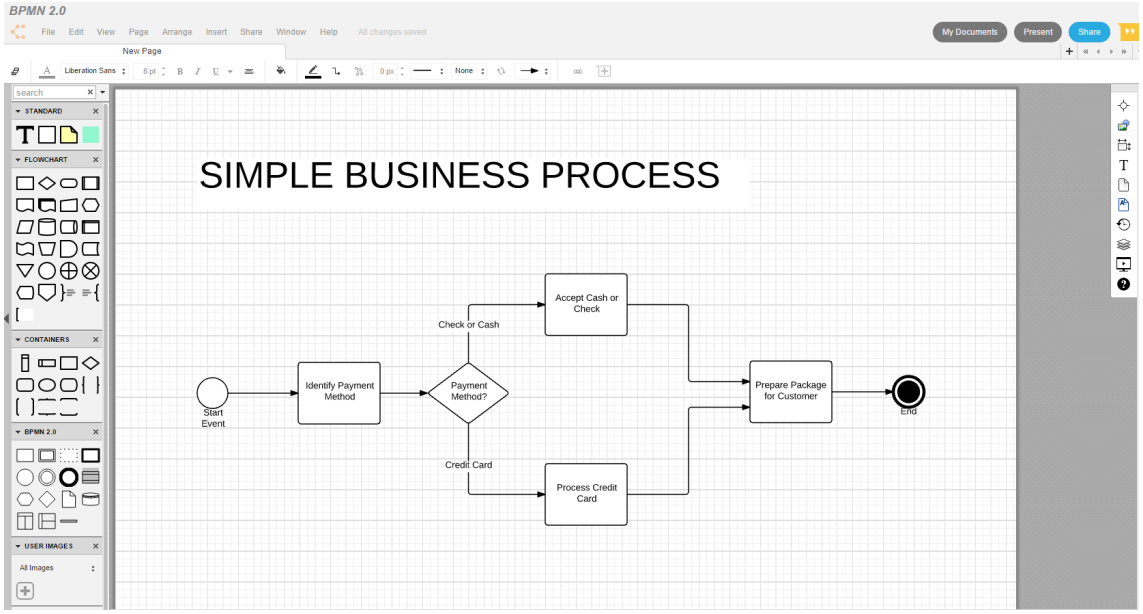


Figure 4.1: User interface of Lucidchart for BPMN diagrams.

in the page menu can be modified and customized, depending on the developers needs. In addition to the page menu bar, a compressed canvas menu is locate on the right to offer quick actions for the most common features.

Lucidchart is used for collaborative diagram modeling, so that a diagram can be shared with other users. To press home the advantage of the web based approach, other user can directly access and modify the diagrams in real time, since no further software has to be installed. In addition to this, applications for the major operating systems and mobile devices are proposed, to afford the developers to work offline and synchronize with the cloud at a later time. One of the main features is the drag and drop interface, which leads to a simple and intuitive utilization and short adjustment to the tool. To support the developers on their collaboration, Lucichart provides different communication services embedded in the web application. This includes an in-editor chat to instantly send and receive messages from other developers working on the diagram, comments on all viewable elements and a video chat for more personal contact.

In the majority of cases, the diagrams are created to visualize complex processes and relationships. As a matter of fact, the diagrams has to be presented or embedded in other media. To meet this goal, Lucichart offers the opportunity to export the designed diagrams in common file exchange formats like PDF and image formats like JPEG and PNG. In addition the diagrams can be accessed from other sides by a direct link, published to the web or send via mail.

Since diagram tools are particularly measure according to the number of supported shape libraries, Lucichart offers a large collection of industry-standard shapes. This libraries include shapes and templates for usual process models like flowcharts in general,

BPMN and UML state charts as well as feature diagrams to create a specific setting for different customizable IT infrastructure like networks or server racks. The most shape libraries are provided in nearly every other online diagramming tool as well. The special templates, which are available in Lucidchart are the wireframes and mockups. Both libraries enable the developer to create and design the user interface to demonstrate the visual representation to others and to convey an idea of the web page or mobile application. In addition to this, the work flow of the web page is modeled as well, since the components in the diagram can be assigned to different predefined behaviors. To show the designed wireframe or mockup, a demo mode can be executed to demonstrate the possibilities, actions and paths the user can take. Despite the available shape libraries, the developers are free to defined their own shapes by importing SVG elements, or modify and extend already existing shapes. The imported SVG files can be created by hand or extracted from another *Microsoft Visio* [?] or Omnigraffle file. But the imported SVG files cannot be assigned to any newly defined semantics. They can just be set to an already existing type like an UML Activity to inherit it's behavior.

The Lucidchart diagramming tool provides multiple features for interactive utilization, collaborative creation and communication between the developers. It takes advantage of the cloud based data storage and the resulting calculation offloading, as well as the platform independence. The rich shape library support all common diagram types and additional user interface design tools. After the last release, the potentiality of Lucidchart was spotted by different magazines including *PC World*, which marked Lucidchart as „an online alternative to Microsoft Visio“ [?]. On closer inspection, it is visible that aside from the wireframes every shape library is based on a graphmodel structure consisting nodes, edges and containers. With regard to Pyro and CINCO , it is possible to recreate every shape library of Luicdchart in different CINCO products and with the use of Pyro directly transform them to run in a web browser. Besides, Lucidchart is capable of designing diagrams with predefined shapes, but without any possibility for the developer to define individual constraints, behaviors and especially no further transformation of the created models. In summery Lucidchart is only usable for visualization and demonstration but without any further processing. The different shape libraries for designing diagrams are very perfected but limited.

4.2 GenMyModel

GenMyModel [?] is an online collaborative tool for creating models of predefined types, including code generation. The available models are restricted to UML2 class, use case, sequence, activity and object diagrams as well as BPMN and ERD models. All models can be created and modified directly in the browser, with analogous features like a desktop application to enable ease of use. Like the previous tool, mentioned in the last section 4.1, the

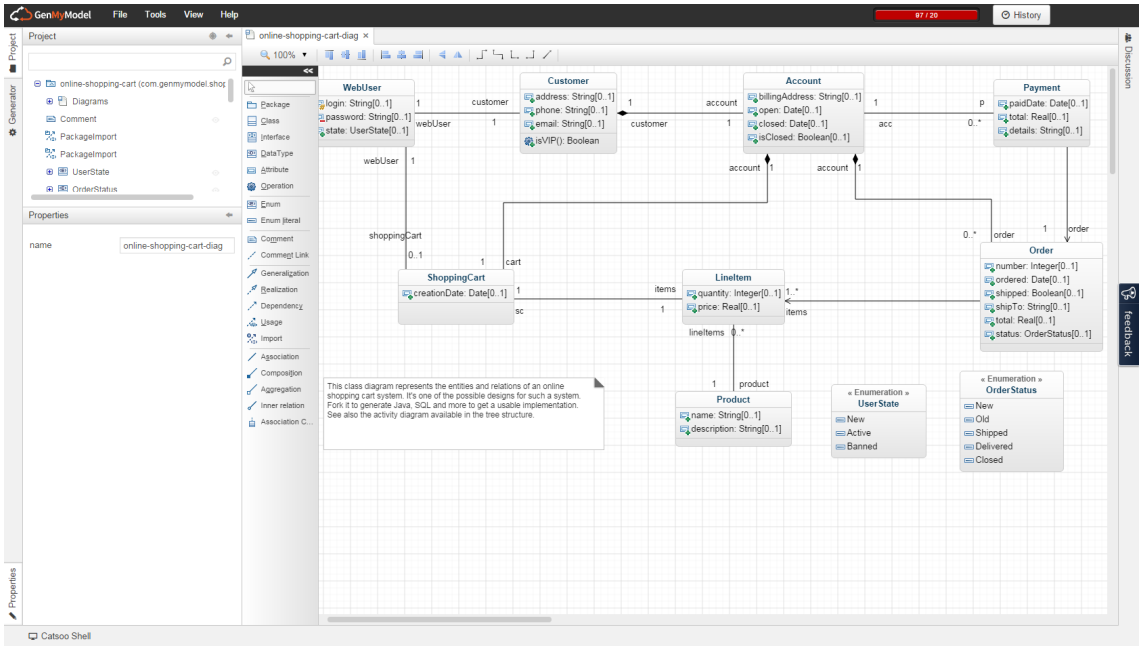


Figure 4.2: User interface of GenMyModel for a UML2 class diagrams.

editor of GenMyModel is a rich internat application with context menus, multiple palettes and different menus shown in figure 4.2. In contrast to the user interface of Lucidchart, a project and file explorer is located in the right, which permits the user with the possibility to manage the different models. The other menus are similar to Lucidchart and include a palette and a menu bar to modify the edges. The GenMyModel interfaces appears more cheerfull and minimalist than Lucidchart. Every menu is constructed especially for the different model types. In contrast to the Lucidchart user interface, a properties view is displayed in the lower left corner, which illustrates predefined input fields for the selected type. The properties view shows that the available types are created on restrictions and semantics, to fulfill the given standards. The diagrams are not just graphics for demonstration purpose, but instances of the predefined meta models of UML, BPMN and ERD, including the described semantics and syntax. As a result of this, it is not possible to combine unrelated types in one diagram, or connect two attributes in an entity relationship diagram as an example.

The GenMyModel online modeling environment concentrates on the usability and support for the developer in every phase of the realization. This includes the live validation of every model depending on its type and the prevailing constraints and invariants. In the example of an UML class diagram, the equally naming of two attributes causes an error, to guarantee a valid code generation. The code generation is only available for the data meta models, including UML class diagrams and the ERD. For these diagram types, the developer can choose between different target programming languages. As a result of this the modeled UML class structure can be generated to different Java classes or plenty of

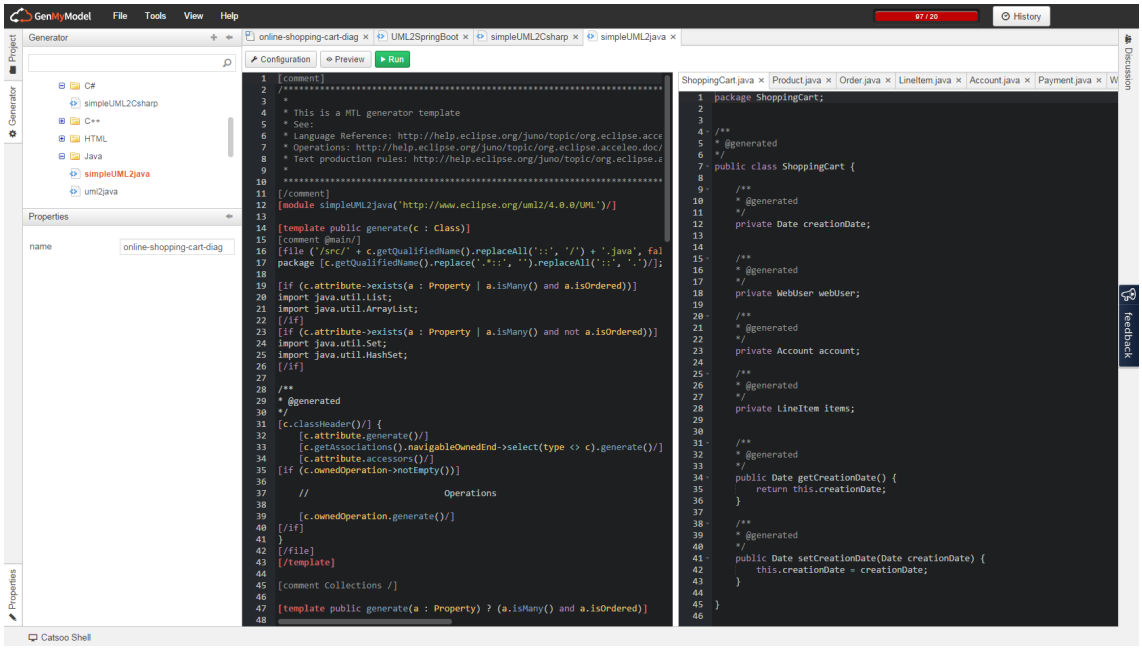


Figure 4.3: Code generator template of GenMyModel for a UML2 class diagrams to Java code.

SQL statements, which will create a database schema depending on the class diagram. The code generation itself is realized by *Acceleo*. Acceleo is a pragmatic implementation of the Object Management Group (OMG) MOF Model to Text Language (MTL) standard. Acceleo tries to help the developer to handle the lifecycle of its code generators. The template language is embedded in the static code similar to other Eclipse templating engines like XTend [**?**]. The language provides known features like loops, conditions multiple string operations. A disadvantage of this web based code generator is the fact, that all logical operations are located directly in the template and cannot be swapped to external programs. In addition to this, the templates of GenMyModel cannot be extended with different specializations or modularized. Besides the code generation, which is not capable for all model types, every diagram can be exported in different visualizing file formats to present and publish the diagrams to others.

The collaboration of all developers is enabled in real time and leads to the insignificance of further versioning. The synchronization is done on the server, which propagates every change on the model to every client similar to the Lucidchart tool. The communication between the different developers is realized with a chat between them, to dialog about the current or other joined diagrams. GenMyModel offers special features for business processes like the appropriated level of details in the BPMN diagram depending on the role of the editor. In addition to this the combination of the different diagram types in a single tool permits the process improvement by closing the gap between IT and business teams. The modeled BPMN can be injected to every standard execution engine to automate the business model workflow.

In contrast to the prior mentioned Lucidchart, the GenMyModel web application offers additional usage to the developer. The models which are created in the GenMyModel can be generated to executable code to build the modeled software. As a result of this, the diagrams are not just visualizations. The restriction to the low amount of model types by comparison to the Lucidchart, shows that the usability of GenMyModel could be extended, if further diagrams and generators are added to the tool. With the aid of CINCO , this further models and especial domain specific models, which are designed for one special purpose, can be designed and transferred to the web with Pyro. But the approach of code generation based on the online modeled diagram is not realizable with Pyro at this time, so that this displays an interesting feature for the future.

Chapter 5

Conclusion

This thesis describes the conception and realization of Pyro. The idea behind Pyro, is the possibility to sustain a ready to use modeling canvas for the web, depending on the definitions, done in a CINCO product development. Furthermore, an entire environment is created to enable exactly equal usability and extendability of a desktop CINCO product for Eclipse. To meet this goal, the Dynamic Web-Application is used as the underlying container, to benefit from the DyWA's abilities depending on dynamic data schema.

Since Pyro is addicted to a CINCO product, the CINCO meta modeling SCCE is described first. Every CINCO product is based on the two meta languages MGL and MSL. The developer describes the desired modeling components by writing a MGL file. For simplicity reasons, the supported components are limited to graph types like nodes, edges and containers. The components can be extended with attributes as well as constraints for connection and embedding. To define the visual representation of the predefined components, a MSL file is created. The defined styles, which can be associated to the MGL components, are extendable with multiple shapes. Every shape can be customized by additional appearances, so that the developer's creativity is not bounded. The created MGL and MSL files represent the foundation of the CINCO product and Pyro. As a result of this, the Ecore meta models for the MGL and MSL are illustrated, described and exemplified. In addition to this, the relations between the defined components and the generated types are explained.

As mentioned before, the generated Pyro Modeling Environment is embedded in the DyWA. This includes the necessity to create a DyWA specific database to store the information as well as the utilization of the DyWA framework. To show the affected functions and interfaces, the DyWA is described. The DyWA offers the possibility to create an entire meta model for a domain specific web application. The developer defines DBTypes and DBFields to realize a meta model similar to form based UML class diagram. After the creation, the available types can be instantiated in DBObjects. The DyWA's administration interface provides multiple actions to perform modifications as well as a migration

support, to change the domain after instances already exist. In addition, the DyWA is extendable with multiple apps and plugins, which uses the predefined meta model. One of this plugins is the generator plugin, which creates domain specific entities and controllers to provide well typed access to the domain objects.

Pyro has to realize a bride between the definitions in the MGL and MSL files of CINCO and the DyWA to enable the modeling environment for the web. To meet this goal, the Pyro Meta Plugin for the CINCO product development is created. This meta plugin is executed when the common CINCO product generation is triggered. The plugin obtain the created components and styles of the MGL and MSL. This information is used to perform a model transformation, which maps the collected component to DyWA DBTypes. This step is combined with additional code generation, based on the read out model, for the entire DyWA app. The generated Pyro Modeling Environment mocks and improves the CINCO product features with the special benefits of the web interface and cloud storage. To enable the requirements of the pursued rich internet application, the PME is constructed based on a command pattern, which provides the ability to redo and undo action as well as the collaboration of multiple developers on one graphmodel instance. The detailed realization, including the meta model and the sequential process, are illustrated in this thesis. Although Pyro represents multiple features of CINCO at this time, still many functions has to be realized. One of the major requests is the migration support to enable the domain modification extinguished by CINCO to the DyWA DBTypes and Pyro.

The next step for the evolution of Pyro is the porting of the CINCO product development itself to the web based on the DyWA. This includes the possibility to create MGL files by defining components and style them using the MSL shapes and appearances. Still a long time of development is needed to meet this goal, but with the already finished proof of concept by Pyro, the end of th way is in sight.

List of Figures

Bibliography

[1] Arlow, Jim and Ila Neustadt: *UML 2 and the unified process: practical object-oriented analysis and design*. Pearson Education, 2005.

[2] Bassett, L.: *Introduction to JavaScript Object Notation: A To-the-Point Guide to JSON*. O'Reilly Media, 2015.

[3] Boniface, Michael, Bassem Nasser, Juri Papay, Stephen C Phillips, Arturo Servin, Xiaoyu Yang, Zlatko Zlatev, Spyridon V Gogouvitis, Gregory Katsaros, Kleopatra Konstanteli et al.: *Platform-as-a-service architecture for real-time quality of service management in clouds*. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pages 155–160. IEEE, 2010.

[4] Chen, Peter Pin-Shan: *The Entity-relationship Model&Mdash;Toward a Unified View of Data*. ACM Trans. Database Syst., 1(1):9–36, 1976.

[5] Client.IO: *JointJS API*. `http://www.jointjs.com/api`, 2015. [Online; accessed October 9, 2015].

[6] "Eclipse.org": *XTend*. `http://www.eclipse.org/xtend/`, 2015. [Online; accessed October 9, 2015].

[7] Fraternali, Piero, Gustavo Rossi and Fernando Sánchez-Figueroa: *Rich internet applications*. Internet Computing, IEEE, 14(3):9–12, 2010.

[8] Gamma, E., R. Helm, R. Johnson and J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[9] "GenMyModel": *GenMyModel.com*. `https://www.genmymodel.com/`, 2015. [Online; accessed October 9, 2015].

[10] Gorman, Brian: *Learning Java EE 7*. InfiniteSkills, 2014.

[11] Graphiti: *a Graphical Tooling Infrastructure*. `http://www.eclipse.org/graphiti/`, 2015. [Online; accessed October 9, 2015].

[12] Gronback, Richard C: *Eclipse modeling project: a domain-specific language (DSL) toolkit.* Pearson Education, 2009.

[13] Inc.", "Lucid Software: *Lucidchart.* https://www.lucidchart.com/, 2015. [Online; accessed October 9, 2015].

[14] "Java": *Java Expression Language.* http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html, 2015. [Online; accessed October 9, 2015].

[15] Kopetzki, Dawid: *Model-based generation of graphical editors on the basis of abstract meta model specifications.* PhD thesis, Master's thesis, TU Dortmund, 2014.

[16] Margaria, Tiziana and Bernhard Steffen: *Continuous model-driven engineering.* Computer, 42(10):106–109, 2009.

[17] Margaria, Tiziana and Bernhard Steffen: *Continuous model-driven engineering.* Computer, 42:106–109, 2009.

[18] Margaria, Tiziana and Bernhard Steffen: *Simplicity as a driver for agile innovation.* Computer, 6:90–92, 2010.

[19] MarketsandMarkets: *Platform as a Service (PaaS) Market.* http://www.marketsandmarkets.com/PressReleases/platform-as-a-service-paas.asp, 2013. [Online; accessed October 9, 2015].

[20] Martin, J.: *Managing the Data Base Environment.* A James Martin book. Pearson Education, Limited, 1983.

[21] Mealy, George H.: *A Method for Synthesizing Sequential Circuits.* Bell System Technical Journal, 34(5):1045–1079, 1955.

[22] "Microsoft": *Visio.* https://products.office.com/de-de/visio/flowchart-software, 2015. [Online; accessed October 9, 2015].

[23] Naujokat, Stefan, Lybecait Michael, Bernhard Steffen, Dawid Kopetzki and Tiziana Margaria: *Full Generation of Domain-Specifc Graphical Modeling Tools: A Meta2modeling Approach.* under submission, 2015.

[24] Neubauer, Johannes, Markus Frohme, Bernhard Steffen and Tiziana Margaria: *Prototype-driven development of web applications with dywa.* In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 56–72. Springer Berlin Heidelberg, 2014.

[25] "OMG": *Object Constraint Language (OCL) 2.4.* http://www.omg.org/spec/OCL/2.4/, 2014. [Online; accessed October 9, 2015].

[26] "Oracle": *Java SE JDK.* `http://www.oracle.com/technetwork/java/javase/overview/index.html`, 2015. [Online; accessed October 9, 2015].

[27] Richardson, L. and S. Ruby: *RESTful Web Services.* O'Reilly Media, 2008.

[28] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, William E. Lorensen et al.: *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, 1991.

[29] Steffen, Bernhard, Tiziana Margaria, Ralf Nagel, Sven Jörges and Christian Kubczak: *Model-driven development with the jABC.* In *Hardware and Software, Verification and Testing*, pages 92–108. Springer, 2007.

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

[31] Strohmeyer, Robert: *LucidChart Steps Up Online Business Diagrams.* PCWorld, 2010.

[32] Stroustrup, Bjarne: *The C++ programming language.* Pearson Education, 2013.

[33] Thomas, David and Andy Hunt: *The pragmatic programmer*, 2000.

[34] "W3C": *Scalable Vector Graphics (SVG) 1.2.* `http://www.w3.org/TR/2002/WD-SVG12-20021115/#drawingorder`, 2002. [Online; accessed October 9, 2015].

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den October 9, 2015

Philip Zweihoff