MASTER IN COMPUTER SCIENCE

Extensible DSL for Specifying Editors in a MetaCASE Tool

Krzysztof, Magusiak

Award date:
2012

Awarding institution:
University of Namur

University of Namur

Extensible DSL for Specifying Editors in a MetaCASE Tool

Krzysztof Magusiak

Master thesis in Computer Science

2011-2012

Promotor: Vincent Englebert
Supervisor: Daniel Amyot

Abstract

Domain-specific languages (DSL's) become more and more popular for specifying and configuring existing systems. The gap between a DSL and the domain is smaller than the gap between a general-purpose language and the domain. So, these DSL's are simpler to understand and write; however, there is a need for proper tool support for presenting, editing, validating and transforming such models.

In this thesis, an approach for defining a customizable and extensible language for specifying user interfaces for meta-models will be presented. In other words, we will extend an already existing domain-specific language, Grasyla, so that it can specify the concrete syntax and be able to handle events. The model used for testing the presented approach will be the User Requirement Notation (URN). Then, a method for generating basic concrete syntax specifications, mainly for editors, from a meta-model will be presented.

Résumé

Les langages dédiés, *domain-specific languages (DSL's)* deviennent de plus en plus populaires pour spécifier et configurer des systèmes existants. L'écart conceptuel entre les langages dédies et le domaine est plus petit que l'écart entre un langage général et le domaine. Donc, ces langages dédiés sont plus simples à comprendre et à écrire ; cependant, nous avons besoin d'outils pour présenter, éditer, valider et transformer ces modèles.

Dans ce mémoire, une approche pour définir un langage personnalisable et extensible pour spécifier les interfaces utilisateur pour les méta-modèles sera présenté. En d'autres mots, nous étendrons un langage dédié existant, Grasyla, pour qu'il puisse spécifier les syntaxes concrètes et gérer les évènements. Le modèle utilisé pour tester l'approche présentée sera User Requirement Notation (URN). Puis, une méthode pour générer des des syntaxes concrètes de base, principalement pour les éditeurs, sera présenté.

Contents

Chapter 1

Introduction

1.1 Model Driven Engineering

Since the beginning of software development, a trend to create abstractions to alleviate the complexity of the business domains and the underlaying platforms is present. Moreover, the domain and user requirements change over time. This creates a need for easily evolving related software products. Indeed, modeling allows us to describe in an abstract way the system under study. So, we can forget about unnecessary details and technologies. Also, models can be easily modified, verified and source code can be generated from them.

Two categories of languages can be used to model a system. Firstly, general-purpose modeling languages (GPML's) are languages with a defined set of generic concepts. A well known example of a GPML is Unified Modeling Language (UML). Secondly, Model Driven Engineering (MDE) approach uses domain-specific modeling languages (DSML's), or in short domain-specific languages (DSL's), to specify directly the systems under study. When dealing with DSL's, different levels of abstraction need to be defined. Indeed, a model is an abstraction for the system under study. The meta-model allows to define separately the concepts of the domain and the used methodology. Hence, a meta-model represents the concepts of a language that is used to describe a model, in other words, the model is an instance of the meta-model.

In this thesis, we advocate that, when solving a particular type of problems, DSML's are superior to GPML's. They have many advantages: users and experts manipulate *instances of* concepts that are already known. Therefore, users would be more familiar with the objects they use and would be able to describe a system configuration that meet the requirements of the stakeholders. Also, a description of a system using a DSML should be more concise that a description made using a GPML because a DSML is closer to the domain.

Note that modeling languages are generally classified as graphical or textual. Graphical languages are structured with different shapes and symbols for different concepts whereas textual languages are structured by using keywords. By the way, some languages can be operational, that means executable by a machine. Others will simply describe a system and could be used as an input configuration for some software. Langlois, Jitia, and Jouenne [37] present a more complete classification of DSL's.

1.2 Domain-specific languages

As explained in the previous section, DSL's allow users to describe directly the system under study using the concepts that they know. However, it can be difficult or costly to develop a DSL. The drawbacks of using a DSL are the learning cost of the language, the cost of developing a tooling support and integrating it with the existing development environment.

Any language is composed of three components: the abstract syntax, the concrete syntax and the semantics. The abstract syntax is a structure defined by some meta-model, it is composed of the concepts manipulated in the language. The semantics denote the meaning associated with the concepts of the language. Finally, the concrete syntax is a particular encoding for the abstract syntax that indicates how different concepts are represented. Any model has an abstract syntax defined by its meta-model, but there is no standard way to describe concrete syntaxes [21]. Moreover, there can be several completely different concrete syntaxes for a single abstract syntax. For instance, state charts can be represented both graphically and textually.

When a user learns how to use a DSL, they must know the matching between the concrete syntax and the abstract syntax and the semantics; which in case of DSL's are at least partially known. There are tools for general-purpose languages, however for domain-specific purposes new tools have to be created. It is often an expensive and resource-consuming task.

Examples of DSL's Domain-specific languages exist for already some decades. In the Unix world, commands such as `awk` or `grep` use their own language respectively for transforming and filtering text files. *Regular expressions* that can be used by these programs are themselves languages that describe regular expressions. Also, *graphviz* is a graph visualization software, these graphs are described using a defined DSL. A more recent example would be the usage of a python framework for describing mobile phone applications by Nokia [33].

Internal and external DSL To be complete, domain-specific languages can be classified in two categories: *internal* and *external* DSL's [37]. Internal (or embedded) DSL's are a particular way of using a host language, the language's concrete syntax is a subset of the syntax of the host language. This approach has been popularised by languages like Ruby, Groovy, Scala, etc. Similarly, stereotypes can be used in UML to annotate the model to define new concepts. The other category are external DSL's, they have their own syntax and may require a specialized tool for their edition. From this point, when the DSL notion will be used, we will refer to external DSL's if not stated otherwise.

Example of tools CASE (Computer Aided Software Engineering) tools provide an environment for a domain-specific language, or few related languages. An example of a CASE tool is DB-MAIN which is used as a data-architecture tool which supports models such as Entity-Relationship models or UML.

MetaCASE tools are environments that can be specialized into a CASE tool. In other words, it is possible to define new meta-models and work with them in the same tool. Some example of metaCASE tools are: Eclipse Modeling Framework, MetaDone, MetaEdit+, etc. They will be further described in the next chapter.

The problem Existing tools allow the user to specify a concrete syntax for a language, however the user interface is not personalizable nor easily extensible in the existing tools. Also, usually only one concrete syntax is permitted for one abstract syntax.

1.3 Overview

The goal of this thesis is to extend one of the mentioned tools, MetaDone, to be able to specify the user interface and interactions as a part of the concrete syntax of the manipulated model.

The rest of the thesis will be structured as follows. In the beginning, we will precise the context and the scope of the thesis and define the criteria to compare the usability of metaCASE tools. The problem will be clearly described and hypotheses will be defined. Also, we will introduce the URN, the language that will be used to guide and validate the approach by creating an editor. Then, we will present a solution and explain its architecture and implementation. To check if the goal has been achieved, we will validate the solution and show its limits. Finally, we will discuss some related work and extensions that could be made in the future.

Chapter 2

Context

First of all, we are going to define some commonly used types of representations and we are going to introduce some of the existing metaCASE environments. This way, we will know what are the important features that a metaCASE tool should have. Finally, we are going to compare the existing tools and add some remarks.

2.1 Types of representations

We may distinguish between textual and graphical languages. Each of these categories can be subdivided further: for textual languages, we may use Chomsky's hierarchy [9], for graphical languages, we may divide them into geometric-based (spatial) and connection-based languages.

Textual languages These representations are just sequences of characters. Editors of these languages may highlight words or format the text automatically to make them more user friendly.

Geometric-based languages In this kind of graphical representation, the relative positions of elements matter. There are relations between the represented elements such as *inside*, *under*, *rightOf*, *overlaps*, etc.

Connection-based languages This is the main representations for graphs, the important information is whether there is an edge between given elements. Such representations may be recursive, meaning that nodes of a graph can also be graphs; in this scenario, whether edges between graphs are permitted is often implementation specific.

Mixed These categories are not exclusive: for example, class diagrams are mainly connection-based, however they have some geometric-based properties such as a class contained in a package is represented *inside* that package.

Composability of representations Two representations are composable if one of these can be embedded into the other one. For example, a connection-based language may include elements that are represented by textual languages; on the other side, two textual languages may result in an ambiguous grammar when they are composed.

2.2 Usability of metaCASE tools

We are going to define some comparison criteria for metaCASE tools. This will allow us to spot the important features that a metaCASE tool should have. Most of criteria we are going to use were already defined in "Evaluation of Development Tools for Domain-Specific Modeling Languages" [2] which evaluates some tools for the GRL language.

- *Graphical completeness*: Can all notation elements be represented? Is it possible to define a new way of presenting elements inside the tool?
- *Editor usability*:
  - Ease of creating and editing elements
  - Undo/redo support
  - Import/export support
  - Automatic layout
  - etc.
- *Effort*: How much time is required to produce a functional model?
- *Language evolution*: How are older models handled when their meta-model changes?
- *Integration with other languages*:
  - Is it possible to use several models in combination?
  - Is it possible to integrate with other tools?
- *Analysis capabilities*: Can we easily analyze or transform models?
- *Version control*: Can the models be versioned by the tool? This is discussed in a survey by Altmanninger, Seidl, and Wimmer [1].

The tools must be helping the user writing the model even if they go though inconsistent states. In fact, the stored model does not need to be valid at all times. For example, in an entity-relationship model a relationship must have at least two roles, the user may have an entity and create a relationship, but that relationship would have only one role until another linked entity would be created.

2.3 Existing metaCASE tools

2.3.1 AToM^3

AToM^3 [5] is a tool for multi-formalism and meta-modeling. It can be used as a metaCASE tool [38]. It represents meta-models as abstract syntax graphs which makes possible the manipulation of the models by using graph-grammar models. As graph-grammars describe transformations between graphs, model transformations can be done easily in this tool.

The program is written in python, so are the models. Meaning if the user wants to extend the behavior of the model, they have to operate at a very low level. Even if this gives a really fine-grained control to the user, it may be a too low level API which may be verbose. Nevertheless, there is a graphical editor for creating new models and for mapping representations to meta-objects.

2.3.2 Eclipse

Eclipse [11] is an open-source IDE, mainly for Java, which is extensible through plug-ins. In fact, Eclipse runtime is based on the OSGi framework Equinox. Initially, it was developed by IBM, then given to The Eclipse Foundation.

EMF, GEF and GMF

Eclipse Modeling Framework (EMF) is an extension based on the OMG MOF standard [44] that allows to define models. EMF can save the models in the XMI format and generate Java classes for handling them.

MOF is a strict meta-modeling language composed of four layers. The M0 layer is the data layer representing the real-world objects which are described by the M1 layer that are the models. Meta-models are in the M2 layer such as the UML language which describe the M1 layer. Finally, the M3 layer is the meta-meta-model.

Even if EMF provides basic tree-based editors, Graphical Editor Framework (GEF) is used to provide graphical editors. It is based on a Model-View-Controller paradigm. The Graphical Modeling Framework (GMF) provides tools for easily creating graphical editors using EMF and GEF. A graph editor can be generated from an XML description of the mapping between the model elements and shapes. It provides also tools to check the consistency of the model using the Object Constraint Language (OCL). The typical workflow of creating a graphical editor is described in Figure 2.1.

1. Create an EMF domain model
2. Create the diagram models (graphical and tooling definitions)
3. Create a mapping model between the EMF model and the diagram models
4. Refine the graphical model, add constraints, etc.
5. Generate the editor from the created specifications; it can be further used as an Eclipse plug-in

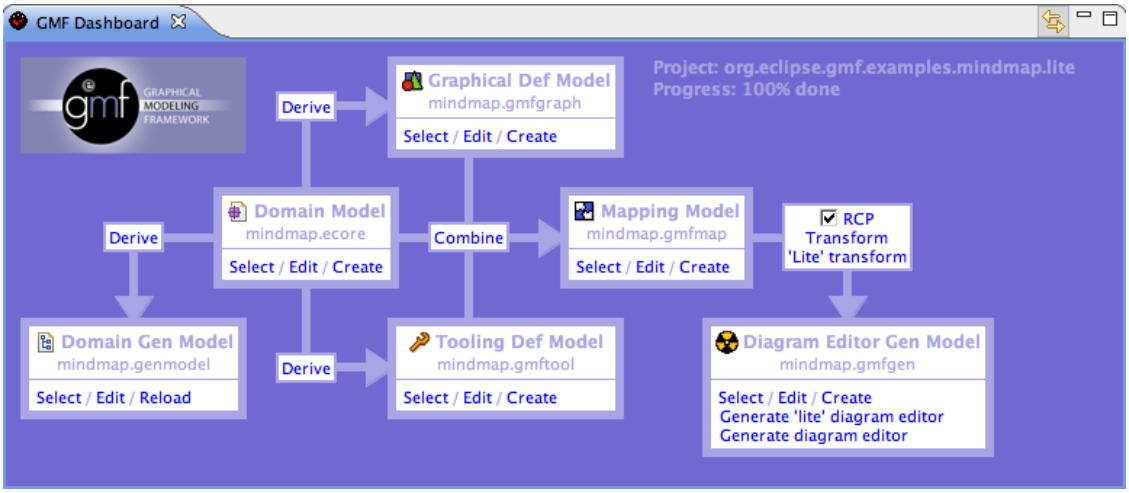

Figure 2.1: The GMF Dashboard (workflow)

Other plug-ins

Graphiti Similar to GMF, Graphiti allows the user to build the editor directly as an Eclipse plug-in.

EuGENia GMF is quite complex and has a steep learning curve. EuGENia is another project that simplifies the development of editors using GMF by simplifying the creation

of the diagram model and the establishment of the mapping. By adding annotations to the meta-model, the rest can be automatically generated.

Conclusion

Even if Eclipse is a usable and extensible tool, creating the editor is usually a hard and time-consuming task. When building an editor, error messages are often unclear and, still, the offered alternative plug-ins are not as expressive as GEF.

2.3.3 GME

GME [24] is a metaCASE tool running on Windows. Its meta-modeling language is based on the UML class diagrams. The models can be stored as XML or in a database. The Object Constraint Language (OCL) can also be used to check the model consistency.

The application is modular an extensible through the use of Microsoft COM technology. For example, the visualization is customizable using decorator interfaces. It is also possible to import existing projects into newly created ones to reuse already defined objects.

2.3.4 MetaDone

MetaDone is a metaCASE tool that is being developed at the University of Namur [12, 17, 15]. Currently, the project is not suited for production usage.

Models are represented using the Metal1 language [13] and are stored in a repository. Relations between objects are objects too, which allows to use them as a source or target of other relations. The core of the Metal1 language is presented in Figure 2.2.

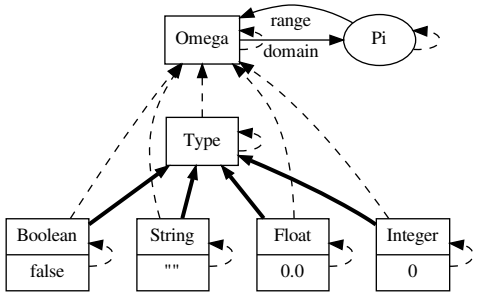

Figure 2.2: Metal1 core

The representation we are going to use for *metal models* is rectangles for vertices and ellipsis for edges. Simple arrows will indicate the domain and the range of an edge, in other words respectively their sources and targets. Bold arrows represent inheritance from sub-type to a super-type. Finally, dashed arrows represent instantiation, the source is an object and the target is the type.

The user manipulates Metal1 models through the use of Metal2 [14] which is a set of proxies for objects in Metal1 that are more user-friendly. There is exactly one Metal1 object for every Metal2 object. There is no restriction on what object is a sub-type or an instance of

another object type; it is a non-strict meta-modeling language. Moreover, models, roles and meta-objects are regular objects too. This allows a kind of reflexivity for the meta-models and an unlimited number of abstraction levels. Additionally, an object can be an instance of multiple meta-objects and so it can be used in multiple models.

In Metal2, `MetaObject` is an object defined in some `MetaModel` that can be instantiated. Two other types of meta-objects are meta-properties and meta-roles. A `MetaProperty` is a meta-object containing a value of a given type. A `MetaRole` is a directed link between two meta-objects: it has a *domain* and a *range* objects. Metal2 will be presented using an example of a state chart, it is described in Figure 2.3.

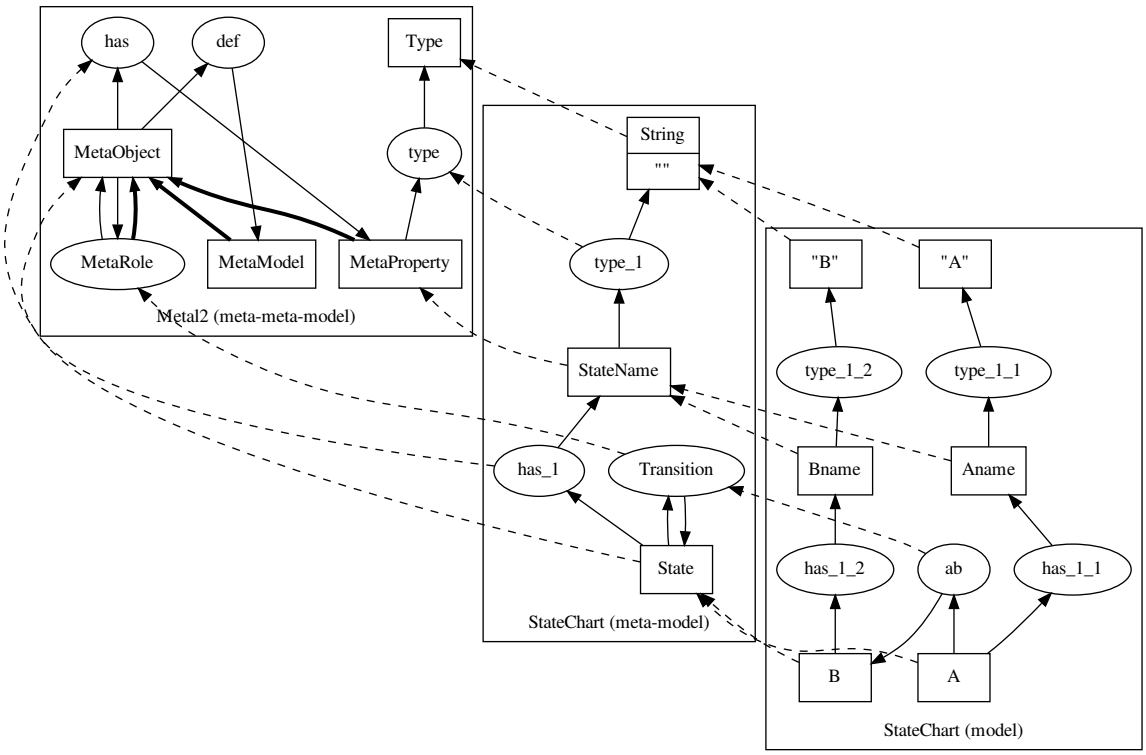

Figure 2.3: Metal2 - state chart example

Only the relevant parts of Metal2 were presented in Figure 2.3 inside the meta-meta-model. In that diagram, we can see that the meta-model for state charts contains named states and transitions between these states. Finally, that state chart is instantiated. It contains two states, `A` and `B` and a transition between them.

Using the provided Java API, the user can create plug-ins that manipulate the repository. Reachable objects are objects such that there exists a path composed of `def` roles from a root model. To find meta-objects, names are used; this works similarly to a file system.

Grasyla

Grasyla (graphical symbolic language) is a language for visual language specification, has been developed [16, 18]. In other words, it permits to define concrete notations for Metal2. Grasyla has a meta-model that is bootstrapped when MetaDone starts. In fact,

Grasyla scripts are Grasyla models. These consist of equations that map meta-objects to expressions that specify how an instance of that meta-object should be rendered.

A Grasyla script is a specification of how a *visualization* of some meta-model is built. A visualization is a representation of a whole model, a *view* is a visualization of a subset of a model.

The rendering is done by a Grasyla interpreter which takes a Grasyla script, a meta-model and an instance of that meta-model and creates a visualization. The editor is a window that has input elements to change the values of the object in the repository. It is generated automatically by the tool when an object is double-clicked; after validation, the whole screen must be redrawn.

A Grasyla script is composed of two parts.

The first part of the script will specify which meta-objects will be drawn on the screen. All the instances of meta-objects marked as "roots" will be drawn first, then the instances of meta-objects marked as "junks" will be drawn if the instance is not already represented.

The second part of the script contains the equations used to choose what representations are to be used. An equation is a quadruplet $(MetaObject, \{single, list\}, String, GExpr)$. The first element is a meta-object, the second tells whether the expression is applicable to a single object or a list of objects, the third is a *functor* which is used to better filter the equations and the last is the expression. The interpreter will choose the best matching equation and represent the object by instantiating an *engine* that will render a component using some given expression. The *GExpr* can be a box, a label, a reference to another object to represent, etc.

The interpreter stores the information specific to the visualization in the repository as a model. For example, it can store the position of the elements on the screen. In its last version, Grasyla scripts can contain programs written in some other scripting language that can be evaluated on initialization or when some part is drawn.

An example of a Grasyla script to represent a state chart is shown below. This script represents the states as rectangles with their names written inside the box. The transitions are represented as edges. The Figure 2.4 shows what could be rendered for a state chart. The Grasyla code for the state chart is written using the concrete syntax of Grasyla, however MetaDone has a meta-model of Grasyla that is used at runtime.

Grasyla 1.0: State chart

```
notation <Simple StateChart> for <StateChart>;
begin
    root each metaobject <State>;
    junk each metaobject <Transition>;

    $ one metaobject <State> = boxV {
        "State:" [ Italic ]
        $<StateName>
    }
    [ Border = "Line" [ StrokeWidth = 2 ],
      bitt1 [ place = "rectangular",
        bittfor = "Transition.domain",
        this = 0,
        maxedges = 10
        ],
      bitt2 [ place = "rectangular",
        bittfor = "Transition.range",
```

```
        this = 0,
        maxedges = 10
20      ]
    ]
    $ many metaobject <State> = group { head tail }

    % metarole <Transition> = edge
25      target shape = anchorTriangle(25, true, false, 0)
        stroke(3.0, rgb(0,0,0))
        at 50% TOP_CENTER: "transition"
end
```

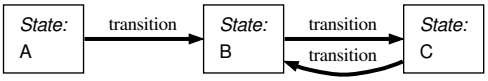

Figure 2.4: Grasyla 1.0: State chart visualization

Grasyla has quite interesting features:
- Declarative language to specify the visualization of objects of a single meta-model.
- Supports a rich set of predefined commands. Also if-then-else statements are included.
- Arguments indicating how to render expressions can be added. An example is the `Italic` option which indicates that the text style must be italic.
- The language supports user-scripting. It is not shown in the example, but it is possible to execute a script that would result in a boolean value that can be used in an if-then-else expression.

However, there are some *limitations*. A Grasyla script specifies a visualization for a single meta-model. For instance, it is not possible to represent different models in the same view. The equation resolution is based on the type of the object in the current meta-model, however an object may have different types in different meta-models. Also, an editor cannot be specified in this language. Moreover, the scripts cannot be composed even if they end up in the repository where links can be created. The language is not regular, meta-objects and meta-roles and handled in a different way. Also, every new feature makes the concrete syntax more complicated and, often, adds new keywords to the language. Attributes of the expressions such as colors, font sizes, etc. are constant values. Finally, the current implementation cannot refresh a part of the graph, the whole window must be redrawn after each modification.

Conclusion

So far, the tool has poor usability: no undo/redo, limited import/export, no shortcuts, etc. Moreover, there is no concrete syntax for the Metal2 language. To add a new meta-model, the user must write a plug-in which would create the meta-model using the provided API; it works for small models, but becomes tedious for models composed of more than 20 meta-objects.

However, the tool can handle multiple concrete syntaxes per meta-model. Moreover, Metal2 is a small and a non-strict meta-modeling language which makes it possible to have a

model that describe other existing models.

2.3.5 MetaEdit+

MetaEdit+ [42] is a popular commercial metaCASE tool which has won several awards. It is composed of three main components:

- The repository server which stores the data (models)
- The workbench for designing the domain-specific languages
- The modeler for using the created languages

One of the user-friendly aspects of this tool is the Symbol Editor which is a graphical editor for concrete syntaxes similar to a drawing tool. Apart from the graph-like view (Figure 2.5), MetaEdit+ also offers other representations such as a matrix editor. It is also possible to check the validity of the models and to generate reports from existing models.

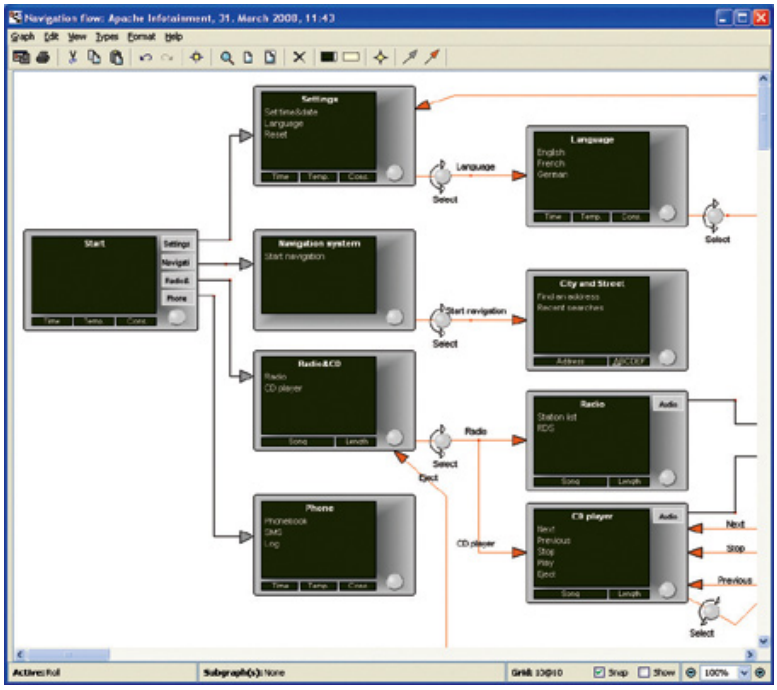

Figure 2.5: MetaEdit+: model example

The abstract syntax is described with GOPRR [25]. It consists mainly of graphs, objects, properties, relations and roles. Everything is saved in a repository.

Other features include a programmatic access to the repositories using an API, model importing and exporting as XML and command-line parameter support. For example, the exchanges between an instance of MetaEdit and a client program are done in SOAP.

2.3.6 MPS

Meta Programming System (MPS) is an open-source language workbench developed by JetBrains [41]. It implements the *language oriented programming* paradigm: programming style consisting of creating, using and extending domain-specific languages [10].

MPS is a projectional editor in which the model is edited in a text-like manner which is handled directly as an abstract syntax tree [52], this eliminates the need for code parsing. The manipulated tree can be directly stored in files (serialized as XML).

Creating a language (the meta-model) in MPS is done by defining the concepts, editors and renderers. Optionally, constraints, generators and a type-system can be defined.

2.3.7 Protégé

Protégé [50] is an ontology editor. Ontologies describe related concepts and relationships; they represent knowledge and define a shared vocabulary. Here, the formats used to store ontologies are OWL and RDF files.

Protégé comes with two different editors: Protégé-Frames and Protégé-OWL. In the first editor, an ontology consists of hierarchies of classes, properties, relationships and their instances. The second editor is used for building ontologies for the Semantic Web as defined by the W3C's Web Ontology Language (OWL), it is not possible to specify a personalized visual representation.

2.4 Comparison based on the usability points

In this section, we are going to compare these metaCASE tools and synthesize their common features.

Graphical completeness

Presented tools can handle all kinds of connection-based representations. The exception is MPS which is text-based and does not handle graphics at all. In Eclipse, the user can even write parts of the view using the Java 2D API.

Different tools prefer either connection-based or textual representations. However, support for geometric-based representations is rare. Few of the presented tools allow the user to write extensions to support new representations and if they do, it is not always straightforward.

One concrete syntax per abstract syntax For example, Eclipse or GME support just a single graphical representation per meta-model. Even if it is better to have just one notation from a cognitive point of view, we might still want to use the same specification language for reports too.
To create textual reports, the tools may have a generator or this can be done through the use of plug-ins. However, reports are another kind of representations of the model, but their creation is done using totally different tools than specifying the graphical representations.

Tools There are some differences between tools like the types of the shapes they support. Overall, all the tools handle connection-based or textual representations. However, only Eclipse and GME support adding new components programmed as plug-ins. All of this will be added to MetaDone. MPS does not support graphical representations of its models.

Editor Usability

Most of the tools support undo/redo, saving of models and automatic layout. The overall usability of the metaCASE tools is good, but from time to time, things like shortcuts are missing.

Editors are generated by the tool and look alike. Still, a specific editor could be more effective for some tasks. Adding such personalized editors for some models can rarely be done; it is however sometimes possible to do it by developing a plug-in. Nevertheless, it may still be difficult to integrate that component with the existing ones.

Tools All the tools generate a default editor. The most usable editors are based on Eclipse. Right now, MetaDone is the worst in this category, it does not have undo/redo, shortcuts or import/export capabilities; also, its generated editor is hard to use.

Effort

MetaEdit+ has an integrated visual editor for the presentation of the model which is intuitive to use compared to the other approaches. Here, Eclipse (GMF) is one of the hardest to learn and use.

Language evolution

Most tools can open old models when new attributes were added to the meta-model, but fail to do so when, for example, meta-classes were renamed or deleted. This point does not entirely apply to MetaDone or MetaEdit where meta-models are saved with the models in the same repository and the model, after modifications the models are also updated; other tools save the model separately from the meta-model.

Integration with other languages

MPS can include directly editors for languages embedded inside other languages. In general, for textual grammars, the resulting grammar might have conflicts. But, meta-models are often completely disjoint, meaning they do not share concepts, and thus it is not possible to display different models in the same view. Still, even if the meta-model would not be disjoint, an object could have different representations depending on the rendered meta-model; which implies that the editor should be aware of which components are rendered for which objects in what meta-model.

Tools Eclipse, MetaEdit+ and others can edit one model per editor; the exception is MPS where it is easy to embed a language inside another. None of the analyzed tools can handle multiple meta-models in the same editor. However, these tools have a common behavior whatever the model they manipulate.

Analysis capabilities

All of the presented tools can export models which can be analyzed by other tools or support plug-ins that can be written to explore their models directly. MetaEdit+ can even expose SOAP services to read the models. Also, some tools support internally constraint checking: for example, Eclipse and GME support OCL.

Version control

Most of these tools save the models as text files which can be versioned. However, model semantics are not versioned, resolving conflicts is difficult and the automatic merge of files can create inconsistent states [26]. Presented tools have no utilities to version a model.

Chapter 3

Problem statement

In this chapter, we are going to introduce the hypotheses, the problem and define exactly the scope of this thesis.

3.1 Problem statement

In section 2.4, we have presented some features that are not handled by the analyzed tools and pointed out some other problems.

As already written, many of the existing tools support the visualization of one concrete syntax per abstract syntax. Nevertheless, many methods need to present models from several view points with sometimes ad-hoc notations. Moreover, very often, models must co-exist with interdependencies between their elements. These relationships are of interest and must be represented. There is a need to show hierarchic structures with possibly alternate or different syntax. Handling both textual, connection-based and geometric-based graphical languages is needed to support different possible notations.

For instance, a concept such as an `Actor` may be represented by a class in a class diagram with a linked state chart for its behavior and an actor in an Use Case diagram. We might want to view the use case diagram with the actor rendered as a class with an embedded view of the state chart. The displayed view should be editable. Finally, we would like to generate some code for the class actor and we would like to use a single language for all of that.

All of this could be done using a single language which would reduce the effort of learning different tools for different representation types. In fact, a single specification language could be used for describing editors if we consider that they are also a concrete syntax. Note that even if existing tools provide ways to change the concrete syntaxes of models, there is no way to specify how the editor behaves. The only way of doing it is to write a plug-in that implements a new editor.

The main objective of this thesis is to show how to improve the usability of a metaCASE tool by improving mainly *the integration with other languages* and *the editor usability* while keeping the *effort* low and still providing *graphical completeness*.

Limiting the scope Other usability criteria presented in section 2.2 are not in the scope of this thesis. These are not directly related to the problem of visualizing a model.

– *Version control*: How to keep track of different versions of the repository.

- *Constraint checking*: This may be different from other approaches. We argue that checking the validity of the model is a concern different from representing or editing a model. It is not necessary to have a valid model at all times, however the user needs feedback from the tool when constraints are not respected.
- *Analysis capabilities*: The presented approach will focus on transforming a model stored in some repository into some representation. Performing only very basic analysis that is relevant for the user interface or the representation itself is sufficient when building an editor. We consider that other, more advanced, analysis can be done directly on the model and not on its representation.

3.2 Hypotheses and method

The tool which we are going to extend will be MetaDone (see 2.3.4). MetaDone has been chosen because it has a small meta-meta-model that does not have limitations inherent to other meta-meta-models. For example, there is no separation between abstraction levels, which allows to create a model with associations to both models and meta-models; as opposed to MOF-based meta-meta-models which are strict. Moreover, it is developed in Namur and it is still in the development phase, this means that we will be able to change it more easily than a completed project. The advantage of using a small meta-meta-model is that there are fewer concepts for which we have to make a mapping to the concrete syntax, which will result in a possibly simpler language than for a complicated meta-meta-model. However, MetaDone comes with some imperfections such as the lack of proper event handling. Also, MetaDone does not have a constraint checker, therefore we will not be able to use constraints for the representation.

By limiting the scope, we will focus on the user interface part. The language Grasyla will be extended to allow to specify user interfaces for stored models. Grasyla already supports hierarchical notations and connection-based languages; we will add support for modular definitions, handling multiple models, possibility to extend the language, refreshing automatically the view, etc. That extension will be developed to support URN described in chapter 4. Also a small Petri editor will be used for debugging and trying new features without changing all the time the developed URN editor.

How the result will be validated For each of the usability points, this section defines the measures that indicate whether the objective is achieved.

- *Integration with other languages*: The tool must be able to visualize two linked models in the same window. It must be possible to draw links between elements from one model and elements from another model. These models may be built using different notations and may be instances of different meta-models.
- *Effort*: The effort is measured by the time and the number of lines of code to write to produce some editor. It is also important to be able to identify rapidly errors if there are any. In other words, the tool must give at least some meaningful feedback when something goes wrong.
- *Graphical completeness*: The first measure that comes to mind is the number of graphical elements that can be represented in the tool. For a given language, we can count the number of features that could not be represented. We have chosen URN as the language for which the editor will be built, therefore we should be able to represent

nearly all the concepts in that language. The tool must support a kind of plug-in based extensions for graphical elements: it should be possible to write a plug-in that adds for example new shapes.
The validation should be pursued with other modeling languages than URN, but this is clearly outside of the scope of this thesis.

– *Editor usability*: The editor usability is evaluated quite subjectively, the user must be able to use a large set of predefined edition components which should also be easy to use. This criterion will be evaluated by summarizing the feedback from presentations of the tool. However, this validation should be done more formally, but this is not in the objectives of this work.

Chapter 4

Presentation of the URN

User Requirement Notation (URN) is a modeling language used to model goals, using the Goal Requirement Language (GRL), and scenarios, using the Use Case Map (UCM) language. It has been standardized by the International Telecommunication Union (ITU) [29, 28]. An overview of the language is presented in "User Requirements Notation: The First Ten Years, The Next Ten Years (Invited Paper)" [3].

This chapter will introduce both languages and an existing editor. For more information, the standard can be consulted. The meta-models are in the annex of *User Requirements Notation (URN) - Language Definition* (Z.151) [28].

4.1 GRL

GRL is a language used for representing stakeholders, their beliefs, goals, choices and required resources to accomplish their goals. It is based on the I* language and inherits a lot of concepts from it [28].

A GRL diagram is composed of *actors* that are represented by the ellipses and *intentional elements*. Actors may contain *intentional elements*:

- goals are represented by rectangles with oval sides
- soft-goals are represented by similar rectangles, but the top and the bottom sides are bent to the center
- tasks to accomplish are represented by hexagons
- resources are represented by rectangles

This language defines several *element links*. The first kind is the *decomposition* relationship that is used to break goals into smaller pieces. The second one is the *contribution* link which indicates how fulfilling some goals influences other goals; it has attributes such as the contribution type. Finally, the *dependency* link indicates what has to be done for a goal to be achieved. These are represented in Figure 4.1.

In Figure 4.2, we give a simple example of a user that wants to secure his data and, by the way, his system. We break the goal *Secure the data* into two sub-goals: ensuring the privacy of the data and backing it up. To ensure the privacy, they will encrypt the hard disks. Similarly, they will use a remote backup service that is provided by another actor.

Another example of a GRL diagram is presented in Figure 4.3, that diagram describes a goal that is the *security*. That goal is achieved by using an *authentication* mechanism that is the card key. The numbers represent the quantitative importance level of a goal. An

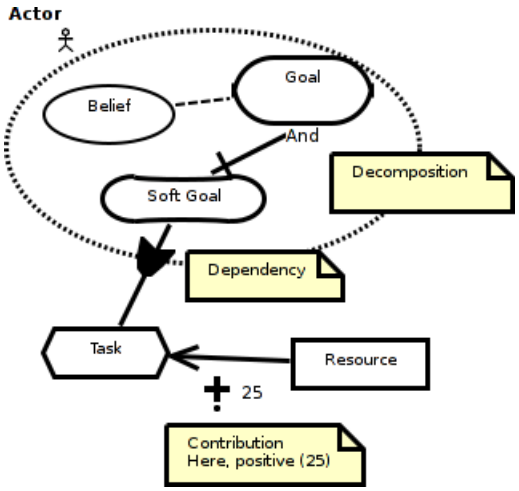

Figure 4.1: GRL symbols

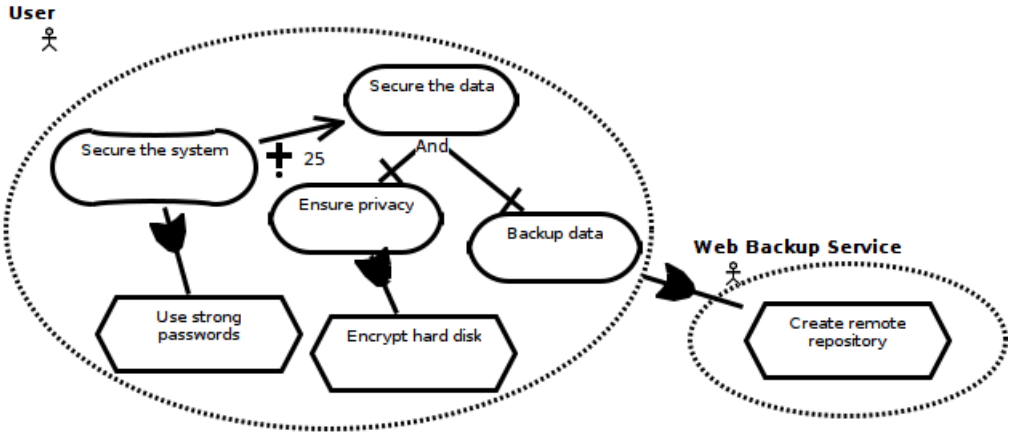

Figure 4.2: Simple GRL graph

algorithm is defined in the norm Z.151 norm to propagate the importance automatically from some goals to the linked ones.

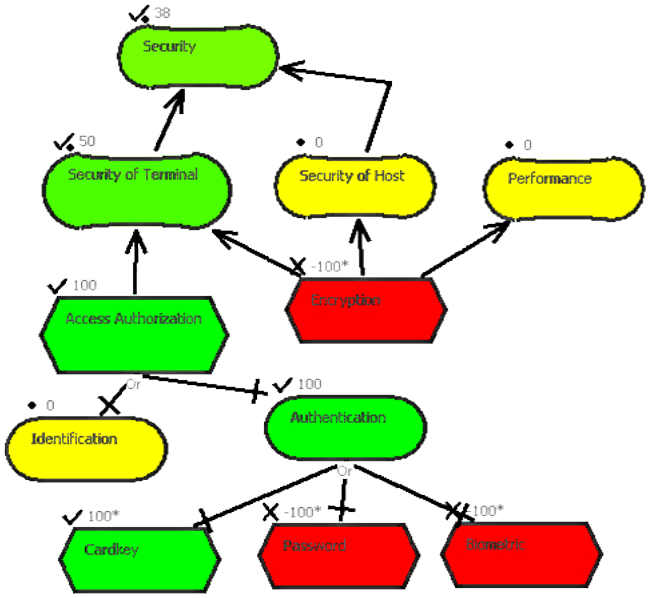

Figure 4.3: Evaluated GRL graph

4.2 UCM

UCM describes scenarios and possible execution paths. A map is composed of at least one entry point and one end point. Optionally, responsibilities reside inside components which can be actors, processes, etc. Choices and parallelism can also be represented using forks and joins. Composition of scenarios is provided through placeholders called stubs which refer to another use case map.

Commonly used symbols are shown in Figure 4.4 and are described in this paragraph. The scenario has one start point and three end points. The first component represents some process *component* by using a parallelogram, the scenario starts there and splits immediately into two parallel paths after the *and-fork*. The upper path leads to some *responsibility* represented by a cross and then enters the team *component*. There is a *waiting place* on the path followed by a choice (*or-fork*): the path ends on `cancelled` or continues to `work`. The path below leads to a *timer*; if a timeout occurs, we go to `timeout`, otherwise if the other execution path hits the timer, we will continue to the *stub*. The *stub* refers to another scenario with one start and one end point. After the *stub*, our scenario ends. A *stub* can be bound to several UCM's. For every binding it defines which *node connections* of the parent map are bound to which *start/end points*.

A more real-world scenario is presented in Figure 4.5 and describes a user buying some merchandises on the web. The user starts by filling an order. Then, an authentication scenario is executed by the shop. On failure, the shop adds an entry to the log and then the scenario ends with a failure; otherwise, the scenario continues normally. After receiving the payment and preparing the merchandise, it is sent to the user and the scenario ends with a success.

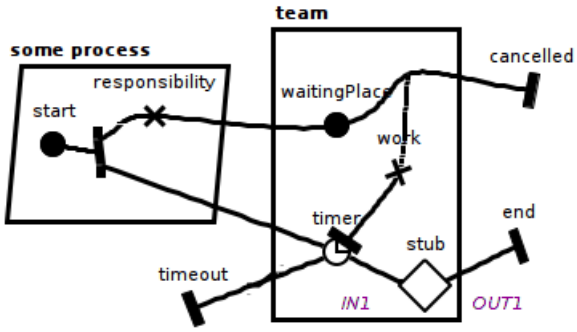

Figure 4.4: UCM symbols

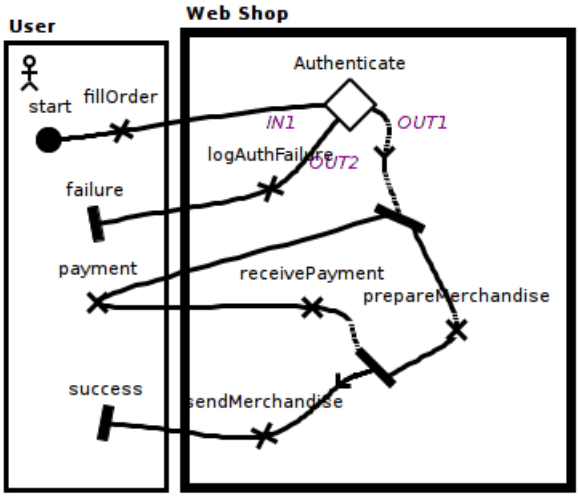

Figure 4.5: A simple scenario in UCM

UCM has multiple extensions that are not presented in this chapter. The first extension adds scenario definitions that specify variables linked to the path and allow to execute individual scenarios. The other extension adds performance annotations used for performance analysis.

4.3 Implementation

4.3.1 jUCMNav

jUCMNav is the current main tool for working with URN models [32]. It is based on Eclipse and offers a rich set of features such as analysis and transformation of existing models. An example of a UCM shown in the tool is presented in Figure 4.6. An execution path taken for some strategy is highlighted in red.

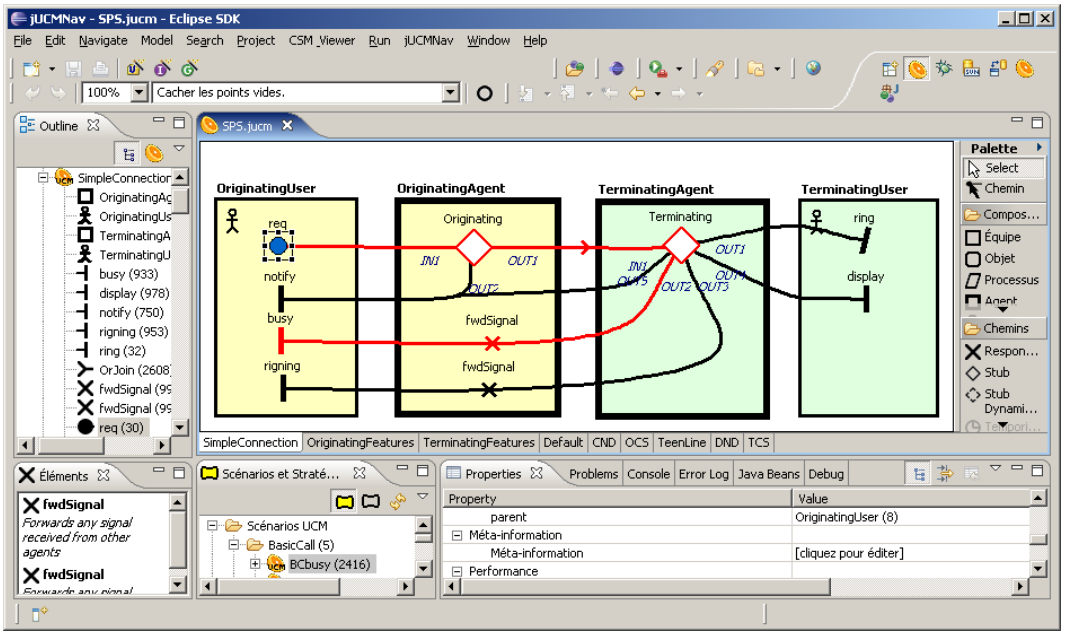

Figure 4.6: jUCMNav displaying a UCM

4.3.2 Representation challenges

GRL is simpler than UCM because it consists of fewer concepts and it is a graph-based notation. As long as the required elements can be represented, there is no special challenges in that notation. Nevertheless, it will be used to test whether our approach can deal with simple graphs. The editor must be able to draw nodes within other nodes and edges between them.

On the other hand, UCM's describe scenarios: the user will try to build a path going through some nodes. Labels can be attached to the parts of a path, for example, the condition for an or-fork can be given. It should also be possible to draw the direction arrow on the edge. Moreover, there are some position-constraints like a timer connected to an end point, meaning that they are represented near each other. Other features include, for example, labels that the user should be able to move around the node or and-forks that should rotate depending on the direction of entering and exiting parts of the path.

Although, representing multiple GRL and UCM models inside a single editor is not defined, it has been requested for the implementation of our tool. When two UCM models are represented, it should be possible to visualize the inclusion of one scenario into the other when the first has a bound stub to the second scenario.

Chapter 5

Proposal of a Grasyla extension

In this thesis, we are going to pursue two main objectives: extending the Grasyla language to achieve our problem statement and refactoring the implementation of the Grasyla interpreter in order to let it support the new features. The first objective consists in making Grasyla more generic and regular. This view will provide features that were lacking but necessary to define ad-hoc editors. That version is called Grasyla 2 in the rest of this document. The second objective consists actually in taking the opportunity, after several years of development, to clean the legacy implementation of Grasyla and to refactor it in order to prepare and facilitate the implementation of the new version.

In the beginning, we will detail the main aspects of our vision. Then, we will describe some of the elements defined as extensions of Grasyla and how the interpreter should behave when encountering them. A description of how the user interface should work and how it can be defined using Grasyla will be discussed later. Finally, an overview of a Grasyla generator for basic user-interfaces will be discussed.

5.1 Solution proposal

In this thesis, we are going to extend Grasyla, the language used in MetaDone (see 2.3.4 and 3.2). Instead of generating an editor for every model, our editor will be interpreted from a specification.

The main difference with the existing approaches is that we will use the defined language for rendering models from the repository as well as editing them. Also, there will be no continuous synchronization between the abstract syntax and the concrete syntax per say, the concrete syntax will not be present in the repository.

We may classify renderers into two categories:

- *Passive* renderers read the specification and the model to create a representation, once done it does not change. This is the case when exporting the model into a file: we just write it once and we are done.
- *Active* renderers listen for changes after the model has been represented, so elements of the representation can be updated. This allows us to define an editor for the language. The repository will send notifications to the rendering engines so they can update the created artifact with new current information.

There are two main approaches to updating the model. The first is keeping the concrete syntax in synchronization with the abstract syntax by defining a double-way mapping

(see 8.1.4 for more information). The second, used in our work, consists in handling events coming from the user interface which will trigger an execution of a user-defined script that performs a model transformation inside the repository. After committing the changes, the interpreter will update the screen; this is how different visualizations of the same model will be kept in synchronization. The overall process can be viewed in Figure 5.1. Moreover, the concrete syntax does not need to be stored as a model. Note that serializing models could be handled by replacing the screen with an output file and stopping the process after the model is rendered.

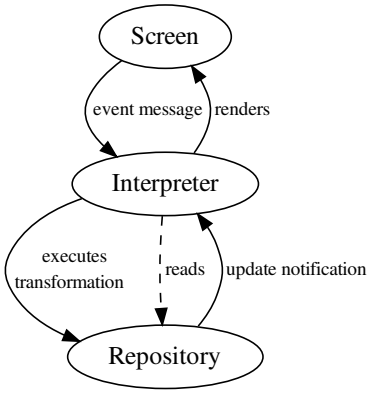

Figure 5.1: Solution overview

The default interface Developing an user interface from scratch is time consuming, even if we use a specific language for that purpose. The second goal of this thesis is to show how to produce a default visual specification from an existing meta-model. Such a specification will be a base that the users will be able to adapt to their needs.

User Requirement Notation The development will be guided to provide an editor for a chosen subset of the User Requirement Notation. The main goal is to handle the edition of both UCM and GRL in the same view.

5.2 Grasyla 2

The Grasyla language was described in section 2.3.4. The new Grasyla language will overcome some limitations of the previous language. Moreover, it will add some new features.

– The number of concepts in the language is reduced to a generic core. By doing this, the language is easier to understand, to extend and to evolve.
Previously, to extend Grasyla, the MetaDone program had to be modified: a meta-object that is a subtype of a Grasyla expression had to be declared, the parser has to be rewritten, and finally classes to render the new expression had to be added. By reducing the language to only the generic concepts, we will be able to extend the language just by writing a plug-in that will register itself as a renderer for some of the existing expressions depending on their property values. The first version of Grasyla had a fixed number of concepts, so adding a new construct was not possible.

- The options are no longer fixed values, they are now expressions. This makes the language more regular and more flexible.
  In the same vein, the values of attributes are now expressions. In the previous version, they were fixed values: this implies that the only parameter of the model that decides what attribute is used is the type of the current concrete object. However, attributes often depend on the properties of concrete objects. For example, a UCM stub has a dashed border if it is *dynamic*, and a solid border otherwise.
- One script may be used for more than one meta-model.
  By allowing to change the meta-model on-the-fly, we will allow the interpreter to render different models inside one view. For example, we could show a class diagram with a state chart in the same view and that would allow us to define links between them. This is required to represent UCM and GRL in the same view if they are both meta-models.
- An *import* statement has been added.
  As of now, Grasyla scripts are independent of other scripts. By adding an *import* statement to the language, equations can be reused. This will reduce the number of copied rules. Moreover, it will allow for example to import automatically generated scripts, for instance a script derived from some meta-model. Indeed, URN is composed of both UCM and GRL, building an integrated URN editor can be done by importing notation specifications for UCM and GRL into that of URN.
- The concept of variables has been introduced to allow the communication between rendered active expressions.
  Handling variables allows Grasyla scripts to store information in the repository about the visualization. An example of an application of variables are background colors that can be defined in a Grasyla script. For instance, to be able to show a collapsed or expanded GRL actor, saving the state in a boolean variable is necessary. Another example is the currently selected elements, they can be stored in a variable that will be updated when for example the user selects an object on the screen.

5.2.1 The meta-model of Grasyla

As explained in the previous sections, a new meta-model has been defined for Grasyla 2. That one has been defined in a monotonic way: all the features of the old version have been preserved in the new version. That meta-model is depicted in Figure 5.2 with the UML class notation. Although that notation is not expressive enough to express all the semantics details of the *Metal2* language, this language is now considered as sufficient to explain the main ideas of that work.

The meta-object `Script` refers to a Grasyla script and is also a meta-model. The objects `MetaObject`, `MetaModel`, and `ConcreteObject` are not part of the Grasyla script, they are elsewhere in the repository and denote respectively the meta-object, the meta-model and the concrete object.

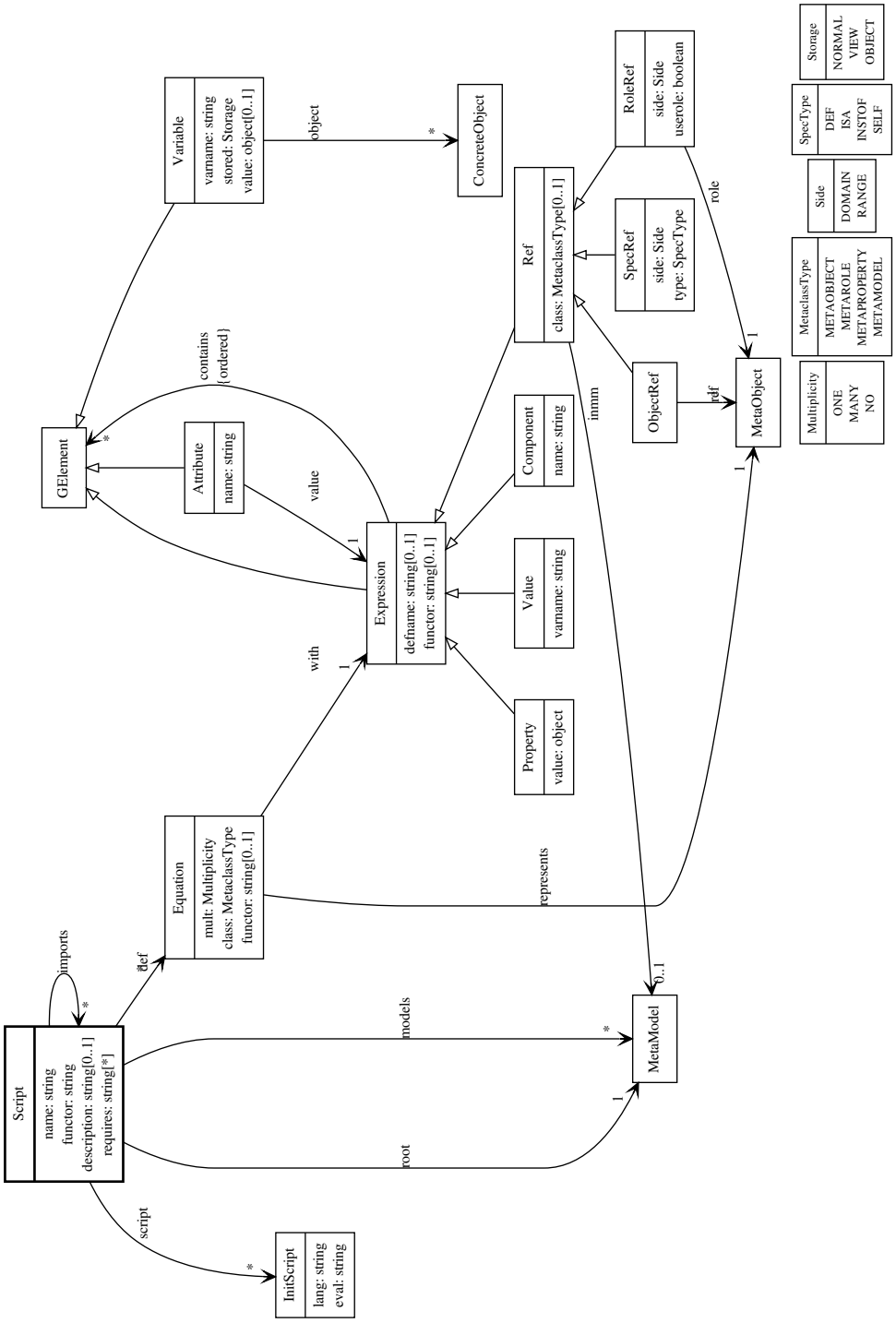

Figure 5.2: New meta-model of Grasyla

`Script.requires` should be a set. The classes with upper-case attributes are enumerations, also the type object is used for MetaDone primitive types.
Constraints for that meta-model include identifiers:

`Script.name`	This is a global identifier for all scripts in the repository.
`InitScript.lang`	Unique inside `script`
`Expression.defname`	Unique inside `Script`
`Variable.varname`	Unique inside `contains`

A Grasyla script is composed of equations and initialization scripts. An equation is defined for some multiplicity, meta-object with a given meta-class type and a functor that allows to choose the right equation depending on the context. The object will be represented with the pointed expression. An expression may contain elements such as attributes, other expressions or variable declarations. The expression may be a component, a property (constant value), a value of a variable or a reference to a linked object. In the last case, the linked object will be represented with another matching equation.

5.2.2 A concrete syntax

The concrete textual syntax for the Grasyla language will be described in this section. It is specified using the BNF notation explained in Appendix B.

The root element is *grasyla*. There can be any amount of white-space or Java-style comments between any two terminals. Most of the rules can be directly mapped to a meta-object in the previous section.

The *header* defines the script properties such as the supported models, the root model, the imported models, required features, etc. Then, it is followed by the definitions of often used expressions; these are expressions instantiated only once in the beginning of the script, this way they can be shared. The rest of the file contains the equations. To define links to meta-objects, their relative or absolute name can be given (*ref* and *aref* rules). Relative names are resolved from the `root` meta-model defined in the *header*.

· grasyla ◀ header (def)* body

· header ◀ **notation** lexID$_{\text{←--notation name}}$ **root** aref$_{\text{←--metamodel}}$ (
description lexSTRING
| **functor** lexID
| **import** **{** (lexID$_{\text{←--script name}}$)+ **}**
| **model** **{** (ref)+ **}**
| **requires** **{** (lexID)+ **}**
| **script** lexID$_{\text{←--lang}}$ lexSTRING$_{\text{←--script}}$
)*

· def ◀ **define** lexID$_{\text{←--name}}$ = expression (**;**)?

· body ◀ (equation (**;**)?)*

· equation ◀ **$** (lexID)?$_{\text{←--functor}}$ multiplicity xclass ref$_{\text{←--meta-object}}$ = expression

· multiplicity ◀ **one** | **many** | **no**

· xclass ◀ **metaobject** | **metaproperty** | **metarole** | **metamodel**

· ref ◀ (**root** | lexID$_{\text{←--name}}$ | aref) (**@** lexID$_{\text{←--relative name}}$)*

· aref ◀ (**@** lexID)+

· gelement ◀ expression | gattribute | gvariable
· expression ◀ expressioncont | lexID$_{\leftarrow\text{--functor}}$ **(** expressioncont **)**
· expressioncont ◀ expressionhead (**{** (gelement)* **}**)$^?$ | [(gelement)*]
· expressionhead ◀ component | property | gref | gvalue
· component ◀ lexIDonly$_{\leftarrow\text{--name}}$
· property ◀ value
· gref ◀ **$** (xclass)$^?$ (**in** ref$_{\leftarrow\text{--meta-model}}$)$^?$ (grefrole | grefobj | grefspec)
· roleside ◀ **domain** | **range**
· grefrole ◀ (**role**)$^?$ ref **.** roleside
· grefobj ◀ ref
· grefspec ◀ ***** lexIDonly$_{\leftarrow\text{--name}}$ (**.** roleside)$^?$
· gattribute ◀ lexID$_{\leftarrow\text{--name}}$ **:** expression
· gdefinition ◀ **&** lexID$_{\leftarrow\text{--definition name}}$
· gvalue ◀ **val** lexID$_{\leftarrow\text{--varname}}$
· gvariable ◀ **var** ([lexID$_{\leftarrow\text{--storage type}}$])$^?$ lexID$_{\leftarrow\text{--varname}}$ (= (
null
| value
| ref
))$^?$
· value ◀ lexBOOLEAN | lexINT | lexFLOAT | lexCHAR | lexSTRING
· lexBOOLEAN ◀ **true** | **false**
· lexINT ◀ `-?[0-9]+`
· lexFLOAT ◀ `-?([0-9]*.)?[0-9]+([eE][+-]?[0-9]+)?`
· lexCHAR ◀ `'([^']|\.)'`
· lexSTRING ◀ STRING | MULTSTRING
· lexIDonly ◀ ID | MARQID
· lexID ◀ lexIDonly | STRING
· STRING ◀ `"([^"]|\.)*"`
· MULTSTRING ◀ `s\{\{\{.*?\}\}\}`
· ID ◀ `[a-zA-Z_][a-zA-Z_0-9]*`
· MARQID ◀ `<.*?>`

5.2.3 Application to a simple Petri meta-model

This section, presents how to build a simple Petri editor to illustrate the concepts from the previous sections. The meta-model of the Petri net can be found in Figure 5.3.

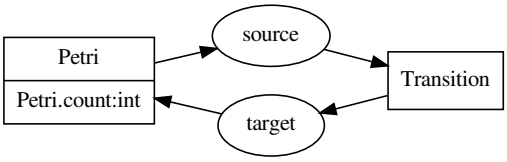

Figure 5.3: Petri meta-model

First, we define a simple representation of this model using Grasyla 2. The editor will be simple: places will become circle nodes with the count and transitions by rectangles. On selection of a place, an editor for the count and the background color will appear. An annotated screenshot of the rendered editor will be shown in Figure 5.5.

The presented script already imports another existing one which is available in section D.1. It defines equations to simplify building graph editors with a property panel, an example of the rendered editor is show in Figure 5.4. The type most commonly used type is `@CommonMetaObject` that is the super-type of all objects in the repository.

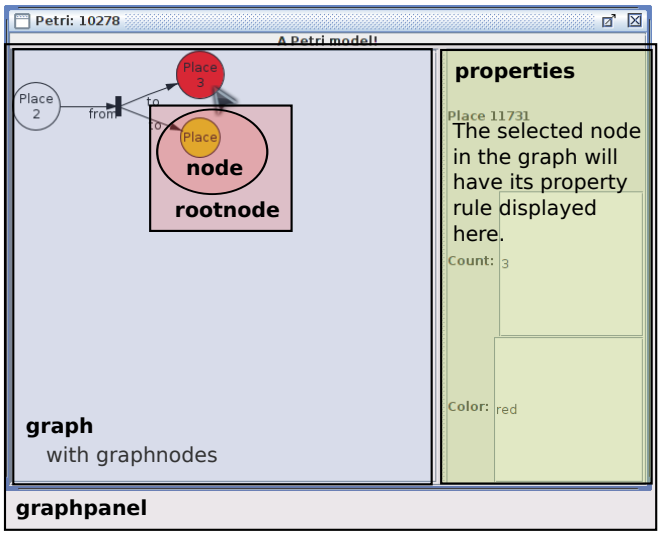

Figure 5.4: Skeleton of the graph editor

The first two equations define respectively the starting point and the contextual menu. As defined in the header, the first equation must have a `start` functor. The `menucreate` functor is used by the imported graph to define the contextual menu. Here, we define two items: creating a place and a transition. Each `menuitem` has a bound action handler which evaluates the contents when the event occurs, here a *click*. The last evaluated item is "consume" to stop propagating the action to the parent elements.

The next four equations define the elements of the graph. By default, all the meta-objects are not shown and we can override `node` or `rootnode` to show what we want. By convention, `node` is used to declare the graphical aspect of the object on the graph. `rootnode` delegates the rendering to `node`, it can be used to filter what objects are shown and to define edges of the graph. This mechanism is not specific to Grasyla, but is simply defined in the imported scripts (see graph rules in section D.1) and can be overridden.
`Transition` is an object that can be rotated, it contains a fixed black rectangle with a given size. The attribute `bitt` specifies the anchor points for the edges of the graph. Its type is "rotated" to indicate that the object should be rotated depending on the angles of entering and exiting edges. It is used for edges whose domain (source) is `Target` or the range (target) is `Source`. `Place` is represented by a circle. A variable `color` is defined for the object and used as the background color. The variable is linked only to the object to be able to modify it from the property panel described below. Then a box with a text "Place" and a centered value of the count is placed. To show the value, an equation with the functor `value` will be

found in the imported scripts to extract the value of the pointed meta-property. At last, a `bitt` attribute is defined as "shape" which will attach the edges at the border of the used shape. `Source` and `Target` are both represented by edges with a target shape that is a triangle. As these are used in the context of a meta-role in which case the used source is the domain and the target is the range of the concrete property. The labels are different, however they can be any graphical element and can be placed anywhere on the edge depending on the attributes they have.

Finally, the equations with `properties` functor define the contents of the property panel on the right of the graph. These are completely overridden here: nothing is shown by default, an editor for the count and the color is shown for `Place` and just a label for `Transition`. The edition elements look alike, they are in horizontal boxes that begin with a label and then the editor component, here a `textfield`. For the color, the same variable are used above is declared so we can use it in the current context. The first line is the value shown in the text field: the value of the count or the value of the variable. Finally, an action handler "validate" is added which is evaluated when the edition has to be validated (for example after the focus has been lost). An update is executed to put the new value in the edited object.

Simple Petri script

```
notation "Simple Petri"
root @Petri // the root meta-model
import {
    "Common Graph"
}
model { root } // the script is defined for the root model
functor "start" // the functor to use for the first equation
description s{{{
A Petri presentation.
}}}

$ start one metamodel root = boxV {
    "A Petri model!"
    graphpanel($*self)
}

$ menucreate one metamodel root = [
    menuitem {
        label: "New place"
        action: "click" {
            create{$Place}
            "consume"
        }
    }
    menuitem {
        label: "New transition"
        action: "click" {
            create{$Transition}
            "consume"
        }
    }
]

$ node one metaobject Transition = rotate {
    rectangle {
        background: "black"
```

```
        space { width: 6 height: 20 }
    }
    bitt: "rotated" {
        for: [ $Source.range $Target.domain ]
    }
}
$ node one metaobject Place = circle {
    var[object] color // declare the color
    background: val color // use the color value as the background
    boxV {
        "Place"
        boxH { spring value ( $"Place.count" ) spring }
    }
    bitt: "shape" {
        for: [ $Source.domain $Target.range ]
    }
}
$ rootnode one metarole Source = edge {
    targetShape: "triangle"
    "from"
}
$ rootnode one metarole Target = edge {
    targetShape: "triangle"
    "to"
}

$ properties one metaobject @CommonMetaObject = none // default
$ properties one metaobject Place = boxV {
    boxH { "Place " id spring } // shows the ID of the object
    // count editor
    boxH { "Count: " textfield {
        value($"Place.count")
        action: "validate" { update{$"Place.count"} }
    }}
    // color editor
    boxH { "Color: " textfield {
        var[object] color
        val color
        action: "validate" { update{val color}}
    }}
}
$ properties one metaobject Transition = boxH { "Transition " id spring }
```

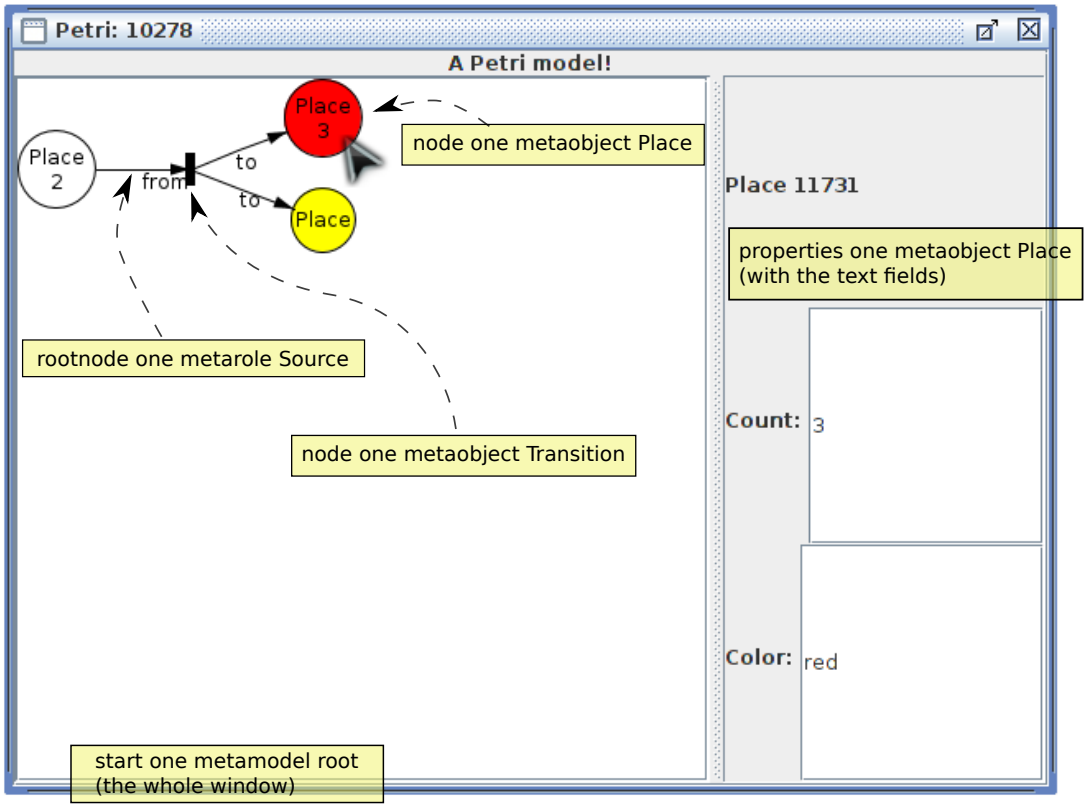

Figure 5.5: Simple Petri screenshot

5.2.4 Semantics

The new meta-model is generic enough to be extended in various ways. For example, connection-based, textual or even geometric-based representations can be handled by specifying the right component names. This section will describe the semantics of meta-objects defined in the Grasyla meta-model.

`Script` The concrete model representing a Grasyla script. It has a unique name and a description. The `root` points to the reference model of the script used to resolve relative references to other objects. Such a script can be used to represent models that are instances of the meta-model pointed by `models`. The `requires` field contains a set of strings indicating what features the interpreter of the script must have in order to be able to interpret this script. That feature allows a script to be interpreted by distinct kind of interpreters as long as they encounter the required features. If a `requires` value starts with `option:`, it is not used for filtering purposes but the interpreter may use it as an option. For example, `option: auto-edit` will add a contextual menu to show the legacy generated editor for Grasyla 1.

A script can import other scripts. The interpretation starts by executing the linked `scripts`. Then, the interpreter loads all the equations. The first equation is chosen using the default `functor`. Imported Grasyla scripts are loaded before the current script, this way rules from the imported scripts can be overridden by new rules.

`InitScript` The initialization script. This is evaluated once by the interpreter using the language `lang` and executing the contents of `eval`. `lang` must be a supported language by the interpreter. As this time, the supported scripting languages are: Groovy and JavaScript.

`Equation` The equations are rules used by the interpreter to `represent` objects of a meta-object type using the expression pointed by `with`. They are characterized by:
- a multiplicity `mult`: ONE, MANY, NO
- a meta-class type `class`: METAOBJECT, METAROLE, METAPROPERTY, METAMODEL
- a `functor`

`GElement` Common super-type for objects contained in expressions.

`Expression` An expression denotes an information that can be translated by the interpreter into some representation. It can have a `functor` which tells what functor should be used for selecting equations for the contained expressions. It is `composed` of ordered elements. The attribute `defname` is set when the expression can be shared (used by more than one other expression), it is optional. When an expression is used in multiple places, it needs a name to be able to easily identify it and being able to generate an unambiguous concrete representation of the Grasyla script.

`Attribute` An attribute is an expression with a `name`. The name can be significant for the interpreter for choosing what to do with the expression. For example, the attribute "width" can be used to indicate the width of a space in a GUI. In Grasyla, it looks like `width: someExpr`. Another example is "action" which indicates that the expression describes an action handler.

`Variable` A variable is a declaration which will attempt to set a variable in the current context to the given `value` or `object`. Every variable has a `name`. The supported types of variables are: strings, integers, real values, booleans and sets of concrete objects. The attribute `store` indicates the storage type of the variable:
- NORMAL means it is stored in memory
- OBJECT means it is stored in the repository for the object
- VIEW means it is stored in the repository for the expression and the object

`var``[VIEW] test` declares a variable `test`.

`Value` A usage of a `Variable`. When it is evaluated, it takes the value of the variable. **`val`** `test` reads the value of `test`.

`Property` A property is an expression with a fixed `value`. Supported types are the same as for the variables. `"hello"` is a simple example.

`Ref` A reference is an expression which may be used by the interpreter to select another equation to continue the rendering. We can indicate in what meta-model the type of the referenced concrete object will be resolved, that means if we want to change the currently used meta-model. Also, a meta-class type can be given to restrict the choice of the equations.

ObjectRef A reference to a meta-object. `$Test` desgins the type `Test`.

RoleRef A reference to a meta-role with a given side, it is used to select a linked meta-role or objects pointed by the role. The `side` indicates the side of the linked object; for instance, if we want to represent the object that is the target of some role, we would use RANGE. Moreover, there is a `userole` attribute which indicates whether the concrete roles or the other concrete objects should be used. In the example above, if `userole` is `true`, the interpreter will proceed with the concrete roles. `$ role SomeRole.range` designs the role of type `SomeRole` that is the domain of the current object.

SpecRef A reference to one of the objects on a side of one of the special types of roles. This exists because things like *instanceof*, *supertype* or *def* are not exposed as meta-roles by MetaDone. Moreover a *self* reference is added, it denotes the current object and can be used to reinterpret the current object with a different functor.

Component A component is a named expression. The interpreter may choose to interpret it differently basing its choice on the name.
A simple example is `boxV`, which evaluates the contained expressions and displays them vertically in a box. It can have attributes such as the background color, borders, action handlers, etc. Other examples of how a component can be handled will be provided later in section 5.2.4.

```
boxV {
    // the contents of the box
    background: "yellow" // background color attribute
    "Hello world!" // normal expression
}
```

Choosing an equation

The interpreter displays an object by selecting the best matching equation and producing a visualization accordingly. The selection process is detailed in algorithm 3. This algorithm consists of choosing all equations that can be used with the given parameters and then sorting them to choose the best matching one.

The different arguments used in the following algorithms are: $func$ for a functor string, $mult \in \{NO, ONE, MANY\}$ for the multiplicity, $mc \in \{MO, MM, MP, MR\}$ for the meta-class types and $mos \in 2^{MO}$ for the set of meta-objects. To choose an equation for a list $objs$ of concrete objects and a given functor, we need to initialize $mult$ from the size of $objs$, $mc =$ `null` or is given by the user and $mos = \{o.instanceOf | o \in objs\}$. If one wants to restrict the meta-objects to a meta-model, then mos would be filtered to leave only meta-objects defined in the given meta-model.

Note that we use the notation $a \subset b$ to indicate that a is a sub-type of b when these are meta-objects. The algorithm 1 finds all the equations matching the criteria described above. In algorithm 2 two equations are compared, the more specific one is higher. In the comparison, first the length of the functor is compared as equations with a functor are more specific than equations without a functor. Then, the types are compared, if one type is a sub-type of another, that equation is considered more specific. Finally, the meta-class type `MO` (METAOBJECT) is considered weaker than the other meta-class types.

Algorithm 1 Finding matching Grasyla equations

Require: *loaded* is the set of loaded equations

1: **function** FINDEQUATIONS($func$, $mult$, mc, mos)
2: $result \leftarrow \emptyset$
3: **if** $mc =$ `null` **then**
4: **for** $mc \in \{MO, MM, MP, MR\}$ **do**
5: **if** $mos = \emptyset \vee \exists mo \in mos : mo.isA(mc)$ **then**
6: $result \leftarrow result \cup$ FINDEQUATIONS($func$, $mult$, mc, mos)
7: **end if**
8: **end for**
9: **else**
10: **if** $func.length > 0$ **then**
11: $result \leftarrow result \cup$ FINDEQUATIONS("", $mult$, mc, mos)
12: **end if**
13: $result \leftarrow result \cup \{eq \in loaded | eq.functor = func \wedge eq.mult = mult \wedge eq.class = mc \wedge (mos = \emptyset \vee \exists mo \in mos : eq.represents \subset mo)\}$
14: **end if**
15: **return** $result$
16: **end function**

Algorithm 2 Comparing Grasyla equations

1: **function** COMPARATOREQUATION(e_1, e_2)
2: **if** $e_1.functor.length > e_2.functor.length$ **then** ▷ functor present, more specific
3: **return** GT
4: **else if** $e_1.functor.length < e_2.functor.length$ **then**
5: **return** LT
6: **end if**
7: $t_1 \leftarrow e_1.represents$
8: $t_2 \leftarrow e_2.represents$ ▷ selects the represented type
9: **if** $t_1 \neq t_2$ **then** ▷ compare the types
10: **if** $t_1 \subsetneq t_2$ **then**
11: **return** GT
12: **else if** $t_2 \subsetneq t_1$ **then**
13: **return** LT
14: **end if**
15: **end if**
16: **if** $e_1.class \neq MO \wedge e_2.class = MO$ **then** ▷ MO is considered to be a weaker constraint
17: **return** GT
18: **else if** $e_1.class = MO \wedge e_2.class \neq MO$ **then**
19: **return** LT
20: **end if**
21: **return** EQ
22: **end function**

Algorithm 3 Choosing a Grasyla equation

Require: $func \neq$ `null`, $mult \neq$ `null`, $mos = \emptyset \Rightarrow mult = NO$
Ensure: *eq* is the selected equation or `null`

```
1: equations ← findEquations(func, mult, mc, mos)
2: if equations = ∅ then
3:     eq ← null
4: else
5:     sort(equations, comparatorEquation)          ▷ sort using the comparator
6:     eq ← last(equations)                         ▷ get the most specific equation
7: end if
```

Interpreting Grasyla

To add the new functionalities, the functioning of the interpreter was altered. This will be described in detail in chapter 6. This section focuses on what kind of extensions can be created using the extended Grasyla language.

The interpreter can choose what type of output it will produce for a given component, for example, a `box` component can become a *Swing* panel, a widget or it could even become a string by wrapping the contents in a box drawn using ASCII art. The output is a component that is a list of objects of the given output type.

Expressions other than `Component`

`Property` A property is rendered as the value it holds. Examples include all the string values that are rendered as labels or the values for the width and the height of the transitions in the Petri example.

`Value` The value of the variable is represented. If it is undefined, an empty component is returned. Otherwise, if it is a concrete object, the object is rendered after choosing a matching equation. Finally, if it is a simple value, it is rendered the same way as a property.

`ObjectRef` A concrete property of the currently interpreted object should be rendered using a found equation. Such an object is usually an instance of the referenced meta-property.

`RoleRef` A concrete role is used to select the objects to interpret.

`SpecRef` One of the objects on the given side is rendered.

Some components

This paragraph describes briefly how some of the components are handled to give an overview of the language. Other components are possible to build, but it would be entering too much into the details. There are implemented components to build editors, tables, menus, selecting the range or the domain, getting the ID of an object from the repository, doing boolean operations, including images, etc. Moreover, new components can be added using plug-ins.

Generic components These can be used to produce any type of output. Such components are usually used to change the selection or the context of the interpretation.

Group A group will just create all of the contained components and return them as a list.

Head This will create the components only for the first concrete object of the current list of objects.

If Contains a list of `Guard` components, the result is the first child component that has the value `true` attached to it.

Guard Similar to a group, but *attaches* a boolean value evaluated for the `condition` attribute to the returned component.

List For each of the concrete objects in the current list, an element is created using an equation that matches a single element.

None A none component will always return a component formed of an empty list.

Tail Similar to the head, all but the first concrete object is rendered by reselecting a matching equation.

Graph-related components Used to build graphs.

Graph The rendered component is a graph. The contained expressions are rendered as nodes or edges.

Free A special node where the contained expressions are rendered as inside nodes, it makes possible to build hierarchic graphs. It should be possible for an edge to go from inside of one free component to another.

Edge An edge declaration is rendered as an edge of the graph if there exist anchors for it to be attached to. Otherwise, the edge is not shown.

Components for edition The *box* containing children components is the main example here. It may have an orientation, it is similar to the `Group` component, but a single object is returned. It is used to build simple layouts when combined with `spring` that acts as a filler. Text fields, buttons, check boxes and other components are also possible to build. The user can specify attributes such as colors, borders, etc.

Chapter 6

Architecture and implementation

Firstly, we will present some of the patters used in GUI development. Secondly, the architecture of the Grasyla interpreter and its implementation will be detailed. Thirdly, we will discuss the composition property of Grasyla scripts and how plug-ins can be developed in order to add new engines or event types. Finally, we will discuss how to handle events generated by the repository or by user interactions.

6.1 Patterns for GUI development

The goal of this section is to provide an overview of the most commonly used patterns for developing a GUI. The goal of the presented patterns is to separate the model (data) from its representation. It is also possible to find variations of the given patterns even if the main idea is still the same. The patterns below are explained in detail by Fowler [22, 23].

Model-View-Controller (MVC) MVC is a widely used design pattern used to develop different user interface elements. It separates the represented data, the controls and the presentation. The model consists of data representing the state of the program. The view is a representation of the state, it observes the model for changes. Finally, the controller manages user interactions. It translates inputs into operations on the model and may request the change of the view.

Model-View-Presenter (MVP) It is similar to the MVC pattern. However, the view handles the user events and the presenter acts as a mediator between the view and the model. In other words, the view translates low-level events into higher level calls on the mediator which in turn knows how the modify the model. The difference between the two main variants is that in one of them the view does not know anything about the model, it gets its data through the presenter, and in the other the view can read data directly from the model.

Presentation Model In this pattern, the view presents and modifies the data exposed by the presentation model. The presentation model is a partial copy of the real model and then it is synchronized with the real model automatically or upon user interaction (for instance after clicking on an *Apply* button). In other words, the presentation model is a facade to the model that is presented by the view.

NakedObjects NakedObjects is a pattern that derives automatically the user interface from the domain model. It allows to create object-oriented user interfaces where the interface is generated from the source code of the domain. This pattern is described in "Naked Objects" [46].

6.2 The Grasyla interpreter

This section will cover the implementation of a Grasyla interpreter created for MetaDone. Firstly, we will describe the different parts of the interpreter, their responsibilities and how they interact. Then, we are going to show some implementations of Grasyla engines used for rendering generic components.

6.2.1 Overview

Before going into details, let us start by defining the main parts of the implementation and their roles. The whole implementation is based on the MVP pattern, meaning that the interpreter is a mediator between the screen and the repository. Although, an implementation of an engine may also choose to copy internally some parts of the model, in which case, the followed pattern is the presentation model; for example, it can be used for pop-up windows.

Below, different parts of the interpreter are described, the related class diagram is represented in Figure 6.1. Basically, the interpreter is an engine initialized with some context. The interpretation starts when a build of a component of some type is requested through the method `buildComponent`.

Component A component is a container that stores a list of objects produced by an engine. The type of the produced object is determined by the engine. For some types, such as `String`, the component is able to reduce a list of objects into one object, otherwise it is the role of the engine to do so. In MVP, this is the *view*. If the component object can receive user events, the engine observes and translates them accordingly into events understood by other engines.

Context Contains the necessary information about what is rendered and has accessors to things like the current concrete object, meta-model, the script, the view, etc. In other words the context is a facade to the *model*.

Engine The *presenter* (or controller) that creates components and reacts to events. The different engines form a tree and each of them is linked to a Grasyla expression.

EngineFactory Chooses which engine to create for a given Grasyla expression and for a given output type. The factory is accessible for the engine from its context, it is used to create children engines.

EngineInterpreter The interpreter is a special engine that is the "root" for the interpretation of a Grasyla script. In other words, it may not have a parent engine.

6.2.2 The interpretation process

The implementation starts by choosing a meta-model, a concrete model and a view. This is done using a dialog window presented in Figure 6.2. The view is a model storing properties of the displayed objects and having a link to the Grasyla script to use during the interpretation.

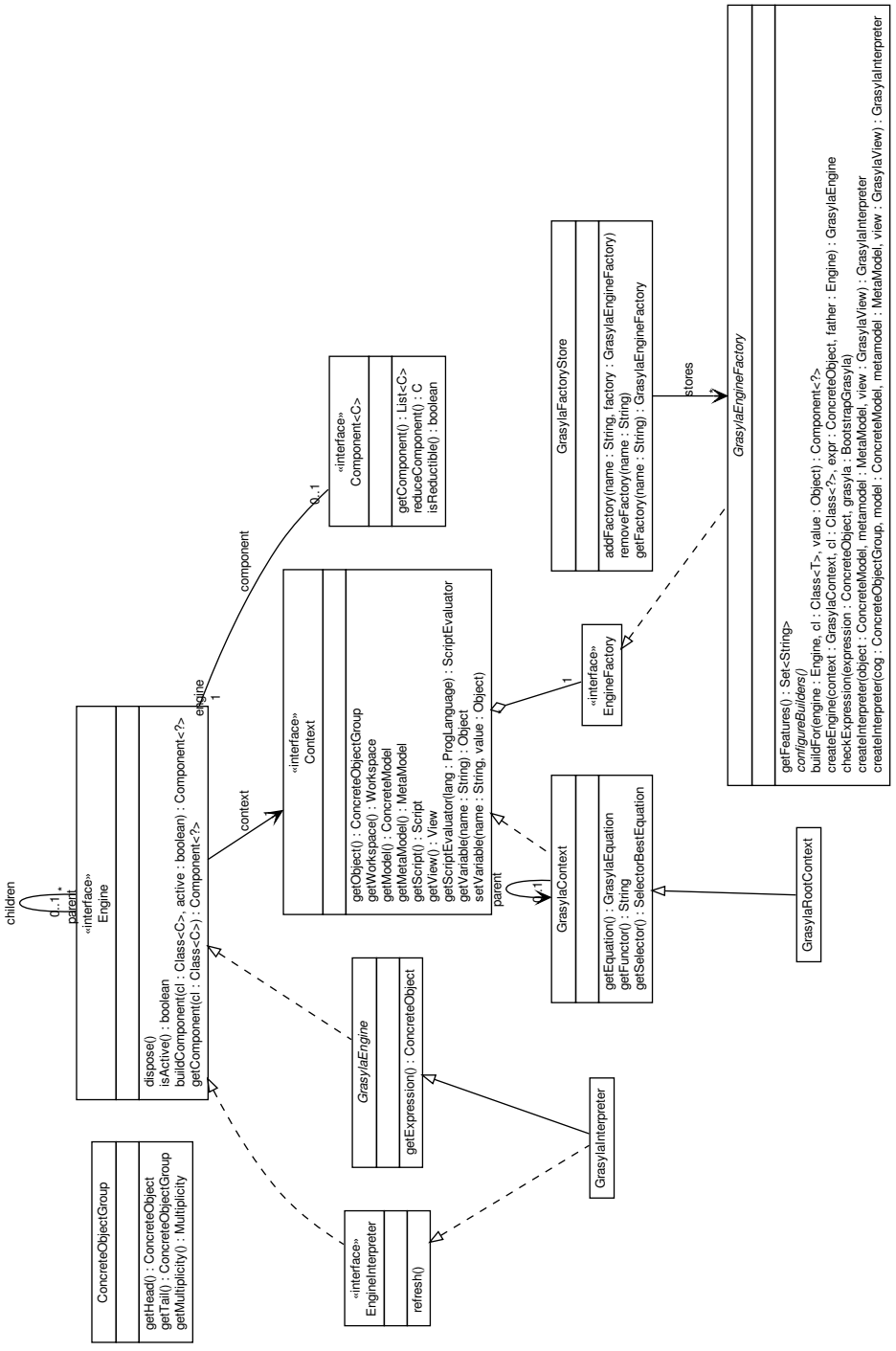

Figure 6.1: Interpreter classes

A `GrasylaEngine` is able to build a `Component` of a given type for some expression. It uses a `GrasylaContext` which has the references to the current equation, `GrasylaEngineFactory`.

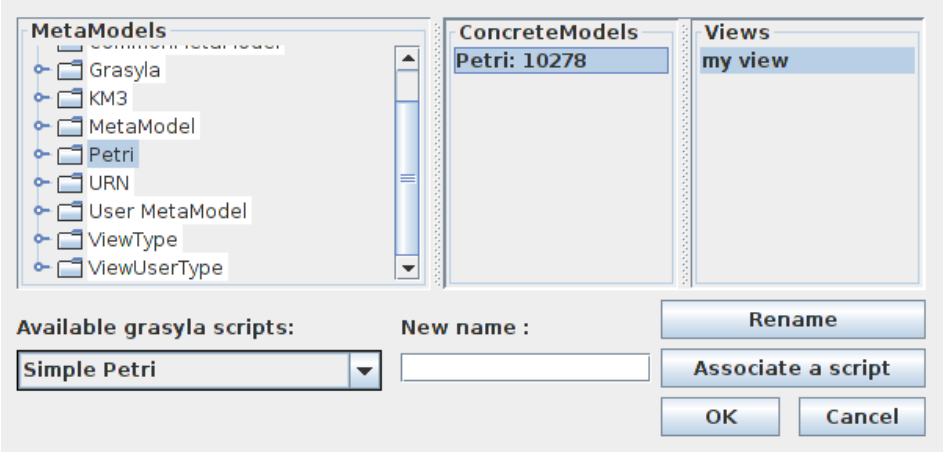

Figure 6.2: Selecting a view in MetaDone

Initiating the process First of all, we need to get an instance of `GrasylaEngineFactory` to be able to decide which engine implementations are used for which expressions. For this, we will use a globally available `GrasylaFactoryStore`. The store can choose the right implementation based on the requested features by the Grasyla script. By being globally available, any plug-in can add or modify the factories in the store that will be returned. For example, a plug-in can add a new engine type for some component.

Creating the interpreter The configuration starts by instantiating a `GrasylaRootContext`. Now a `GrasylaInterpreter` can be instantiated. The interpreter is then configured: attributes of the context are checked if they exist, initialization scripts are run and equations of the script are loaded (including these of the imported Grasyla scripts).

Requesting a component This last step is the same for every engine, it is described in the next section. In the case of a `GrasylaInterpreter`, it finds an equation matching the given object and creates a child engine that will build the component using the expression of that equation. Usually a `JComponent` object will be requested to build a *Swing* user interface, however that choice is made by the calling environment.

6.2.3 The engine classes

The responsibility of engines is to translate a Grasyla expression into an object of some requested type by a parent component. A simple example will describe the main idea: suppose we want to create a *Swing* component (`JComponent`) from a Grasyla expression. All the variables contained in the Grasyla expression must be instantiated in the view and handlers for events specific to the created `JComponent` must be registered. For example, a label component can be created (`JLabel`). To set the text of the label, a single `String` is needed. The contents are transformed by creating engines that will translate contained sub-expressions into `Strings` which are then concatenated. Moreover, attributes can be contained in the expression that can indicate what font or background color should be used.

The Figure 6.3 presents a part of the hierarchy of the engine classes. The light-blue classes are generic classes that are the base for developing all the other engines and are described in detail below. Other classes are just examples of existing engines.

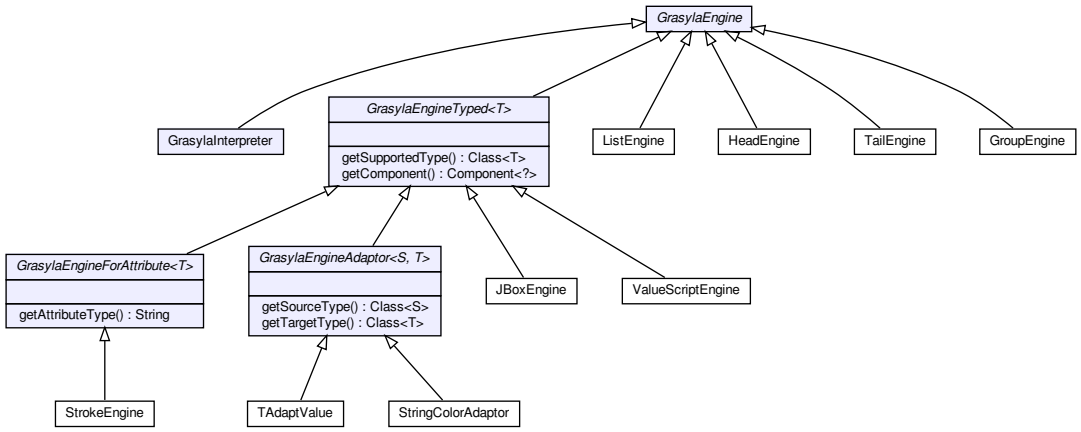

Figure 6.3: Engine classes

`GrasylaEngine` This is the base for all of the engines. It provides many methods to ease the development of sub-classes without worrying about all the details. For instance, the order in which things must be created or disposed is enforced by the super-class. `GrasylaEngine` has multiple responsibilities:

– Managing the state of the engine. The transitions are described in Figure 6.4.

DISPOSED No component is built and the engine does not have any children.

BUILDING In transition from DISPOSED to READY.

READY The component is built.

DISPOSING In transition from READY to DISPOSED.

ACTIVE The component is built and the engine is listening for events.

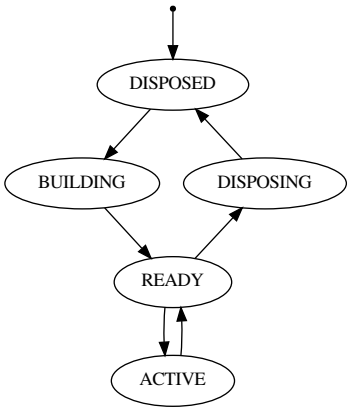

Figure 6.4: States of a `GrasylaEngine`

- Managing automatically its children. This includes responsibilities such as disposing them when the engine is being disposed, activation is also recursively triggered on the child engines, etc.
- Attribute handling. An abstract class `AttributeHandler` is defined and works for the most common cases of attribute handling. It means that attributes of the expression are interpreted and sent to the handlers for modifying the created component. This is done automatically for all the attributes that are not marked as *native* when a component has been created or is being deleted.
- Event handling. `GrasylaEngine` are notified of all the `EngineType` events. The default implementation dispatches the event to *action* attribute handlers and then passes the event to the parent engine or to the main event manager of MetaDone when the parent is not a `GrasylaEngine`. Another type of events that are handled separately are the notifications received from the children components when they change. These are used to update the component of the parent engine when their children change.
- Handling of the context. The context is initialized if necessary for every engine. This includes things such as declaring the used variables or setting the right functor in the context.
- Debugging code has been added to be able to draw the entire tree of the engines to a DOT file.

`GrasylaInterpreter` Already described in subsection 6.2.2. By being and instanceof a `GrasylaEngine`, it can be used to nest interpreters and let them handle independently some rendered part.

`GrasylaEngineTyped` An engine specialized for just creating one type of components. This allow to have engines specific to some type and avoids the need to check whether the requested type is supported.

In fact, most of the engines will handle only one type of components and factories will be able to choose which one to use in a given context. An example of such engine is `JBoxEngine` which creates only `JPanels` and populates them with the contained expressions. In contrast, `ListEngine` can create lists of any requested type.

`GrasylaEngineForAttribute` A specialization of a typed engine for returning an element from a set of possible values. When the result is a component containing immutable objects, this is the simplest class to use.

An example is a `StrokeEngine` that creates `Strokes` which are immutable objects in Java and are used to draw lines with different patterns (solid, dashed, dotted, etc.).

`GrasylaEngineAdaptor` A very useful abstraction that is specialized in converting a component of some type into a component of another type. Often engines will be able to create a component of a single type and the developer would have to write an implementation for each possible type. However, we could write an engine that would request a component of some generic type and then convert it into a more specific type.

Composed with the abilities of `GrasylaEngineFactory`, it is possible to build a chain of transformations instead of having to handle multiple types by each implementation. For example, suppose that we have a Grasyla expression which is a script coded in some scripting language. Now, we want the result as a Java `Color` object, however we only have an engine that is able to build a `Value` (abstraction for numbers, strings and booleans) by executing the script. A value can be easily cast into a `String`. Then the adaptor `StringColorAdaptor` can translate a `String` into a `Color`. This allows us to create an object representing a specific color in Java by converting the result of the script into that color instead of rewriting a script evaluator that returns directly colors. Also, any other producing a `String`, or a `Value`, can now be used to create Java colors without any additional code.

`GrasylaEngineFactory`

In short, a factory must be able to instantiate a `GrayslaEngine` from a Grasyla expression and the expected output component type.

The implementation provides methods to initialize the bindings between types and engine classes in a declarative way. Below, you will find an example of a `configureBuilders` method. In this example, we can already build groups of objects or strings, convert all kind of values into strings and show the internal identifier of a concrete object.

```
@Override
protected void configureBuilders(BootstrapGrasyla grasyla) {
    // newBuilder(Class<?>, Class<? extends GrasylaEngine>)
    // creates a builder for the given output type and engine type

    // bind ValueEngine for a Grasyla value and the type MutableValue
    addBuilder(grasyla.getBootsMO_Value(),
        newBuilder(MutableValue.class, ValueEngine.class));
    // component "id" and type Value is provided by SelfIDEngine
    addBuilder("id", newBuilder(Value.class, SelfIDEngine.class));
    // component "" and any type is provided by GroupEngine
    addBuilder("", newBuilder(Object.class, GroupEngine.class));
    // component "" or "indent" and String type is provided by TGroupEngine
    BuilderOfEngine<TGroupEngine> text = newBuilder(String.class,
        TGroupEngine.class);
    addBuilder("", text);
    addBuilder("indent", text);

    // adaptor: MutableValue -> Value
    addBuilderAdapter(MutableValue.class,
        newBuilder(Value.class, MutableValueAdaptor.class));
    // adaptor: Value -> String
    addBuilderAdapter(Value.class,
        newBuilder(String.class, TAdaptValue.class));
```

```
}
```

The responsibilities of this class are the following:

1. Choose the most specific engine type for a given Grasyla expression.
2. If it is impossible to meet that expression to some engine, find the shortest path using the adaptors to produce the wanted type.
3. Produce error messages when no engine matches.

Implemented mappings

This section will describe which mappings have been implemented so far. The Figure 6.5 shows all of them.

The ellipses represent the possible output types. The gray ellipsis is `Object` which denotes any type. Rose ellipses are defined in `BaseGrasylaFactory` that is a sub-type of `GrasylaEngineFactory`. And green ellipses are defined in `DefaultGrasylaFactory` that adds support for graphs and *Swing* components to the base factory. It is possible to bind an engine for a Grasyla component or another Grasyla expression type, these are represented respectively by plain text representing the component name and rectangles representing the Grasyla type. Some of the names are between parentheses to denote a set of names handled by the same engine.

- (group): *empty*, group
- (script): grv, js, etc.
- (shapes): circle, ellipse, rectangle, etc.

Each edge represent an engine. The names of the engines are not shown to make the image readable. Every dotted edge is bound by an `addBuilder` call and every solid directed edge is bound by an `addBuilderAdapter`.

6.2.4 A complete example

Let us go through the example of the creation of a *Swing* component for a single concrete object of type `Test` requested by a parent component. The equations are defined in Figure 6.6.

The expected result is a `JLabel` containing the value of the concrete property *content* with a monospace bold font. The interpretation process is described in Figure 6.7.

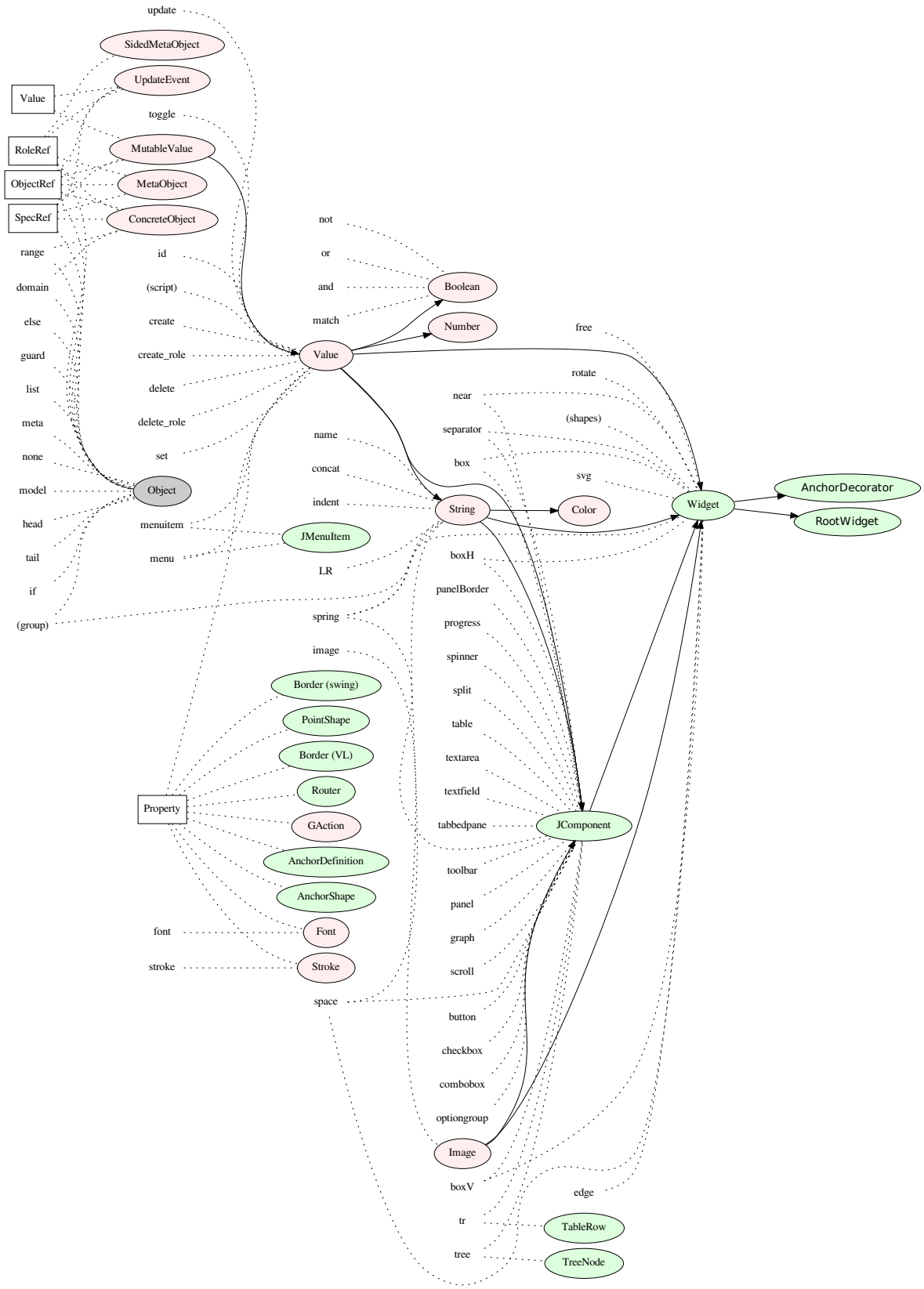

Figure 6.5: All implemented outputs for Grasyla

Concrete syntax

```
$ one metaobject Test = concat {
    font: "monospace" {
        bold: true
    }
    value($content)
}
$ value one metaproperty @CommonMetaObject = $*self
```

Abstract syntax

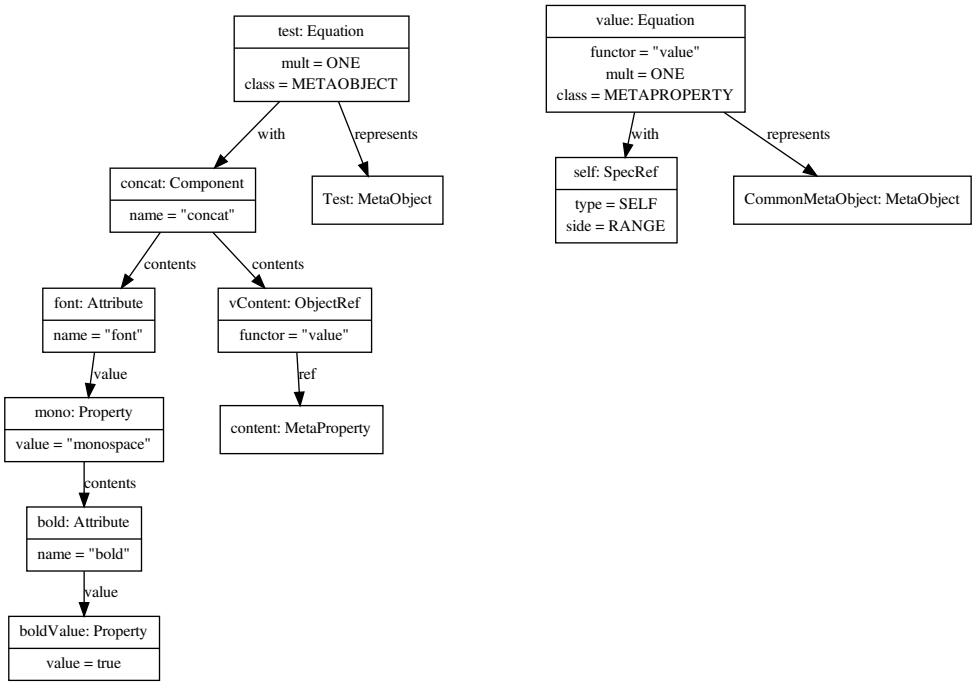

Figure 6.6: Used equations for the example

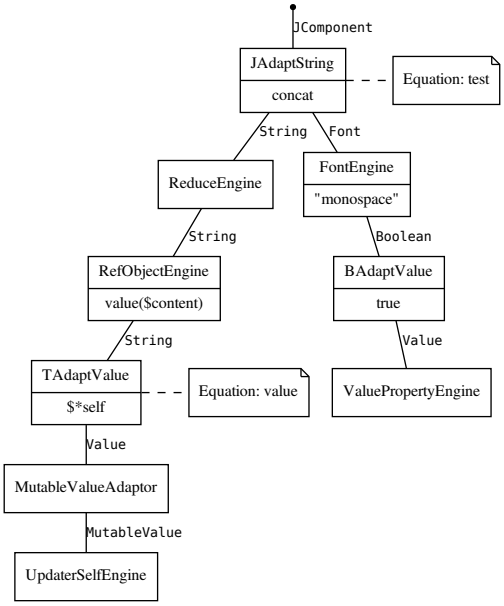

Figure 6.7: Instantiated engines for the example

Nodes represent the type of the instantiated engine and the Grasyla expression being interpreted. If no expression is present, the parent engine is an adaptor. The label on the edges indicate the requested type from the parent.

In the example, the first matched equation is the one containing *concat*. However, an engine converting a *concat* into a `JComponent` does not exist, but it can be built using `JAdaptString` that transforms a `String` into a `JComponent`. Then, *concat* can be transformed using `ReduceEngine` which evaluates the contained expressions and reduces them (concatenation for a string).

The only expression present is the value of the content. The engine handing that expression is `RefObjectEngine` that will find an equation matching the `content` property of the current `Test` object with a functor *value*; the other equation matches and is used. The *self* expression can be evaluated into a `MutableValue` which is an object delegating reads and writes to the repository. The `Value` can be built by simply reading the value and then it can be transformed into a `String`.

The attribute *font* is handled at the level of `JAdaptString`: an object of type `Font` is requested for the property "monospace". `FontEngine` will instantiate a monospaced font and will build a child engine that will return a `Boolean` for the attribute *bold*. The value of that attribute is a property that has a `Value` that can be converted into a `Boolean`.

6.2.5 Allowing plug-ins to extend the behavior

MetaDone is based on the OSGi architecture [45] which allows to load plug-ins (bundles) at runtime. All that must be implemented is an activator that registers an instance of the `MetadonePlugin` interface. The plug-in may register event handlers for the application or modify exposed objects. The first way to extend the existing behavior is managing the events and the second is to create and register a new Grasyla factory.

Managing events Every event that is not consumed by the interpreter will end up dispatched to the main MetaDone event manager and any plug-in can listen for these events. This includes clicking on an object, contextual menus, etc. In response to such events, the plug-in can do anything it wants: change some objects in the repository, add menus to a contextual menu event, etc. With that mechanism, it is now easy to extend the behavior. Moreover, plug-ins can also send their own events through that interface.

Creating factories New factories can be created and added to `GrasylaFactoryStore`. This feature allows to extend the Grasyla language with new components impemented as plugins. For instance, it would be possible to extend it with a plugins to draw charts.

6.3 Composition of visualization scripts

This section will cover the defined common rules and the generator for default user interfaces. The Grasyla scripts for this section are available in Appendix D.

Importing Grasyla scripts and the common definitions

When building an editor, a set of common equations is often used. Therefore, we have defined them once in a common library: see section D.1. These rules were built by factoring all common rules to the URN and the Petri editor as well as defining other rules that seemed useful.

Generator of default interfaces

When building an editor, the components used for editing properties are often the same and depend only on the type of the meta-property. The process of creating these rules has been automatised by providing a generator of a Grasyla script from a meta-model.

The generator of rules for graphical components has been added as a MetaDone plug-in. It adds a menu which prompts the user to choose a meta-model from the repository and then generates a new Grasyla script in the repository that contains the equations for editing objects of the given model. The generated script cannot be used on its own, but contains rules that can be used or redefined by importing scripts.

The generated equations are described in section D.2.

Advantages of importing the scripts

Common rules can be factored into many files and they can be separated from the definitions for a specific language. As shown, generators can be built to create new scripts that

can also be imported; these are necessary as the language is not generic enough to define sets of similar rules with variation points.

Importing scripts is not restricted to predefined common rules. A user can create his own set of rules and reuse them is his scripts. Still, it is possible to override imported definitions, so importing a script does not constrain the user.

6.4 Event handling

MetaDone did not support any kind of event handling. In this section, we are going to describe two types of events that will be added in the implementation that are necessary for the editor. These are the data change notifications produced by the repository and the user events produced by some view. Moreover, this will allow MetaDone to redraw just the parts that have changed instead of redrawing the whole screen after each modification.

6.4.1 Generic event framework

Events are implemented using the observer pattern, here the observable object is the repository. An observer will send a subscription request to an observable object to be notified of changes.

The interface used in MetaDone is described in Figure 6.8, the `Subscriber` is the equivalent of the observer and the `EventTower` is the equivalent of the observable. In the observable class, the method `trigger` dispatches the event to all the subscribers. Note that the wildcards for generic types in `subscribe` are in fact super-types of T which is a sub-type of M.

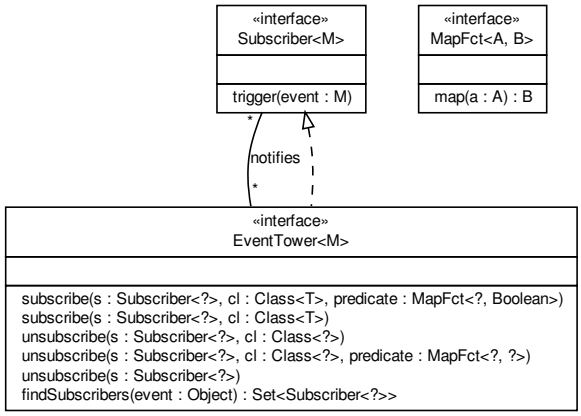

Figure 6.8: Interface for events

Due to the large amount of possibly subscribed objects and the large number of events produced by the repository, we will provide an interface that will be able to subscribe to events of a certain type that will verify certain predicates. The type will be simply represented by

a Java class object and the predicate is a function mapping an input to a boolean value. In fact, most of the subscribers will be interested by events that are one of the following types:

- Being notified every time a change is made may be implemented by listening to all events of the type `Object` and a predicate that always matches.
- An object may want to listen for all changes only of a given type. This is the case when the observed object does not produce too many events, but we want to be notified only for events of a given type.
- The most common case is when we want all the event of a given type with one of its properties equal to a given value. This is for example the case when we want to be notified for all events concerning a specific object in the repository.

6.4.2 Notifications from the repository

With the framework defined above, the implementation will provide events of types defined in Figure 6.9.

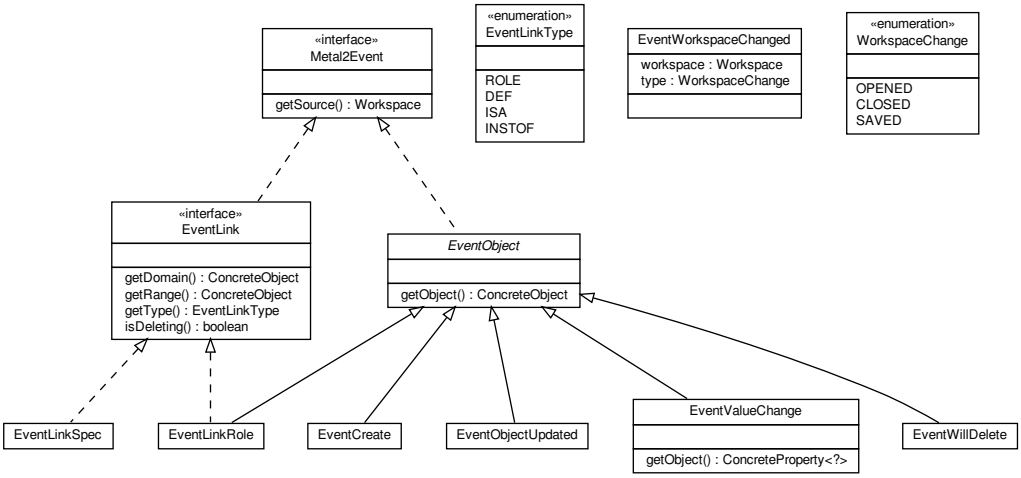

Figure 6.9: Notification events for the repository

- `EventObject`: super-type of all events related to an object change; this contains a reference to the object that has been modified
- `EventCreate`: fired when a new object has been created
- `EventValueChange`: fired when the value of the concrete property changed
- `EventWillDelete`: fired when an object will be deleted
- `EventLink`: fired when a link between two objects has changed (it was created or deleted)
- `EventLinkRole`: `EventLink` for links that are represented by concrete roles
- `EventLinkSpec`: a sub-type of `EventLink` for links that are are not ROLE links; these are properties from *Metal1* language
- `EventWorkspaceChanged`: fired when the state of the repository has changed

6.4.3 User events

The interpreters will be also able to use the event framework to communicate between them or to allow plug-ins to handle some events too. One of these event types is `UserEvent` that represent all the events initiated by the user. These can be as simple as clicking on an object or be more complex like connecting two objects or showing a contextual menu.

Plug-ins are able to observe these events so they can react. For example, a plug-in may add a menu item in a contextual menu when a certain type of object was at the source of the click.

6.4.4 Implementing efficiently the later case

A common case is to use predicates to be notified only for relevant events. However, with a large number of predicates to check, a simple algorithm is linear in the number of predicates. When an event is triggered, after filtering by the type of the event, we must apply every predicate to that event and notify only observers for which the predicate was `true`.

We may note from the previous examples, that the most common case verifies the equality of some property of the event. Thus, most of the predicates can be written in the form: extract a property and then verify the equality to some value. Extracting a property is a function (can be implemented by a sub-type of `MapFct`) that can be reused for every predicate that checks that property in the comparison.

Finally, building a set S of pairs (f, m) where f is a function described above and m is a map from object to a set of observers, solves the problem efficiently.

$$observers = \{o | \exists (f, m) \in S \wedge o \in m(f(\texttt{event}))\}$$

In other words, to find the observers, we must iterate over $setfm$, the observers to add to the accumulator are in the set mapped for the key obtained by applying f to the given event.

The operation of the map m can implemented with a constant time complexity instead of a default linear time which is comparing the result with all the possible keys of that map. By defining the most common used functions f, we will greatly reduce the time necessary to find the relevant observers because the complexity is now linear to the number of such used functions.

Chapter 7

Validation

This chapter explains how our solution solves the problem described in chapter 3. At the end, we will criticize the solution by giving the pros and cons.

Let us briefly remind what were the goals and how they are tested. To check whether it is easy to integrate with other languages, we check whether the scripts can be composed and whether it is easy to build a single editor integrating two languages. The effort is measured by how difficult it was to specify the editor and how many lines of code were required. Also, the graphical completeness is discussed for URN. Finally, comments on the editor usability will be given by reviewing the generated editor.

7.1 Editor for URN

Building the editor for URN in MetaDone consists in defining the meta-model for URN. Then, defining the Grasyla scripts and developing a plug-in to define the meta-model in MetaDone and to better integrate the language in the tool.

The goal is to build a UCM editor similar to the one in jUCMnav, but with some additional features like showing both UCM and GRL in the same view, collapsing and expanding an actor, etc.

7.1.1 Defining the meta-model

The meta-model of URN is defined in *User Requirements Notation (URN) - Language Definition* [28], it specifies the meta-model of GRL, UCM and its extensions. With Prof. Amyot, we agreed not to implement extensions for UCM and also to skip some attributes that are not centric. Indeed, these features can be added later and do not bring anything interesting with respect to our goals. The specification contains also meta-model extensions for the concrete syntax, these were totally skipped, as MetaDone stores the concrete syntax in the view. This allows the meta-model definition to stay more conceptual. The relevant parts of the meta-model were presented in chapter 4.

URN is specified in UML which differs from the language used in MetaDone. The meta-model was presented in chapter 4. Some meta-objects could be represented as meta-roles in MetaDone. A simplification has also been made for `ElementLink` in GRL: an enumerated attribute was added instead of defining sub-types for `Contribution`, `Dependency` and `Decomposition`.

7.1.2 Defining the Grasyla script and the plug-in

The entire script is available in Appendix E. This section will just present an overview of its contents. In fact, there are four scripts, we a going to review them one by one and list the URN characteristics that are addressed by each script.

URN Common Contains definitions used in the editors.

GRL Editor Specifies an editor for `GRLgraphs`. This editor is quite simple, it contains just actors and intentional elements, this is a simple graph.

- `IntentionalElements` can be moved inside an `Actor`. An element is added into an actor by drawing a link between the actor and the element. The size of the actor is adapted to enclose its elements.
- When the user double-clicks on an `Actor`, its state is changed between collapsed and expanded. An expanded actor contains its intentional elements, whereas a collapsed actor is represented as a circle. Edges that linked its intentional elements are still represented, but now they are attached to the actor. This is done by using a boolean variable and rendering the contents only when its value is true.

UCM Editor Specifies an editor for `UCMmaps`.

- Connected components must be represented near each other. This is done by using a `near` Grasyla component which tells the interpreter that two objects in a graph must always be represented together. When one of these is dragged, the other follows.
- The parts of the path are bent depending on their direction and objects on the path are rotated depending on the side from which connections come in and out.
- Images for the actor and the timer are defined and will be loaded by the plug-in.

URN Editor Specifies an integrated editor for GRL and URN.

- Integrates GRL and URN into a single window.
- Links can be drawn from a `Stub` to a `UCMmap` to represent graphically the bindings. This feature is not present in the URN specification, however it has been requested.
- This editor is built by reusing the existing editors and integrating them.

The plug-in

A plug-in has been built for integrating better the editor into MetaDone.

- When the plug-in is loaded, it defines the URN meta-model in the repository.
- When a `URNmodelElement` is created, it is initialized with a random identifier. Other attributes are also initialized to their default values depending on the type.
- Loading images for the `Actor` and the `Timer` from the class-path.

No generated content

The generator (see 6.3) was developed after developing this editor, this is why the scripts do not depend on generated content. This choice has been made because it allowed to see what patterns often occur and to have a better idea of how the generator should work. By developing a first editor manually, we could see patterns that were repeated or that would be repeated if more elements of the same type were added.

7.1.3 The result

To open the editor, the user chooses a model and a script to use from the menu. The user can create new elements by right-clicking on the scene and choosing the wanted contextual menu. Links are drawn by holding the `CTRL` key and joining two elements on the scene. Also, an automatic layout is provided for all graphs or nodes in the graphs that contain other graphs.

Screenshots of the resulting editor are shown in figures 7.1, 7.2, 7.3 and 7.4. Figure 7.1 presents the same GRL graph in two separate windows , once with a collapsed *ISP actor* and once with the actor expanded; on the right side the properties panel for *Sharing information* belief is shown. Figure 7.2 and Figure 7.3 show two connected UCM maps. In the first, the link is selected and appears in red, in the second, the map has been modified. Figure 7.4 represents previously viewed models in different windows, a list of all objects of the selected UCM is shown (a lot of objects can be seen, some of them are internal to MetaDone). *All* the contents of the windows are managed using Grasyla: this includes the graph, the property panel, the tree view for URN and the events. For example, the selection of an element on the graph triggers an update of the property panel for the newly selected object.

Then, Figure 7.5 compares outputs of MetaDone with these of jUCMnav. You can note minor differences in the notations, this is due to the difference between used libraries and to the differences between implemented meta-models. Also, the representation implemented in MetaDone could still be improved, like being able to mask GRL importance of intentional elements. UCM maps are rendered very well, however the approach to build them in MetaDone is completely different from jUCMnav. In fact, MetaDone draws graphs and their semantics are not verified, so it is easy to build an invalid UCM, by for example adding an unlinked responsibility. jUCMnav has a transformational approach: the existing path is transformed into another valid one, this means that the map is always valid.

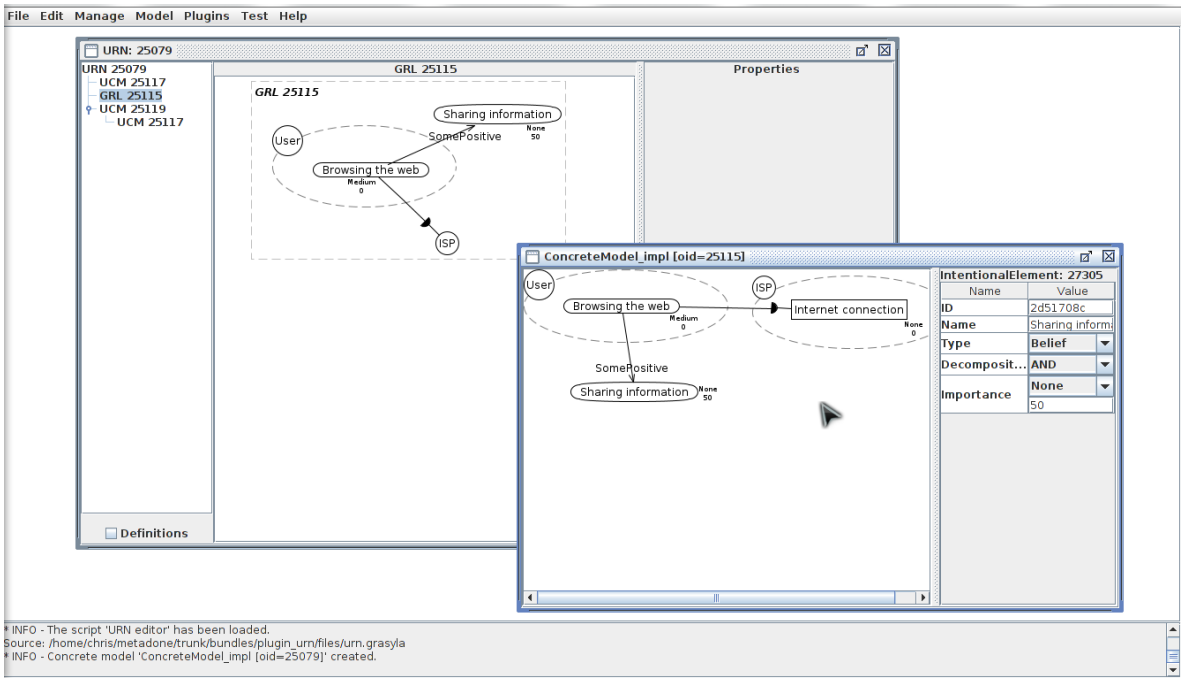

Figure 7.1: MetaDone - GRL graph

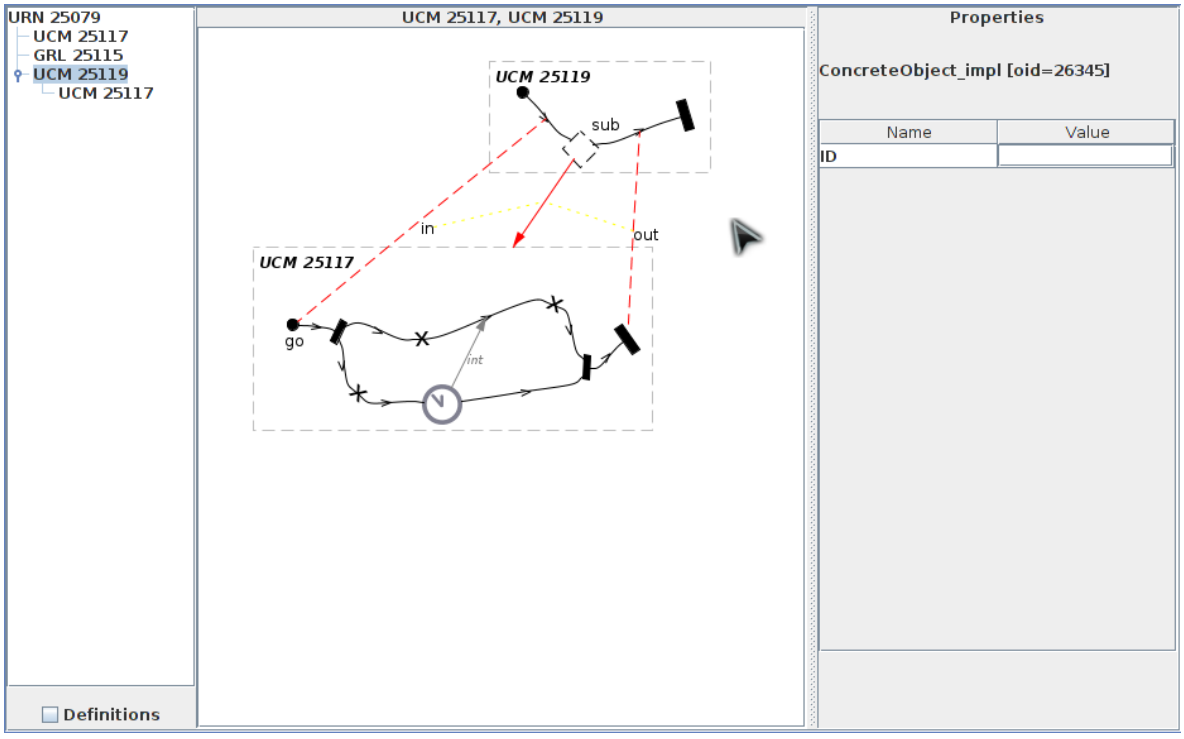

Figure 7.2: Two connected UCM

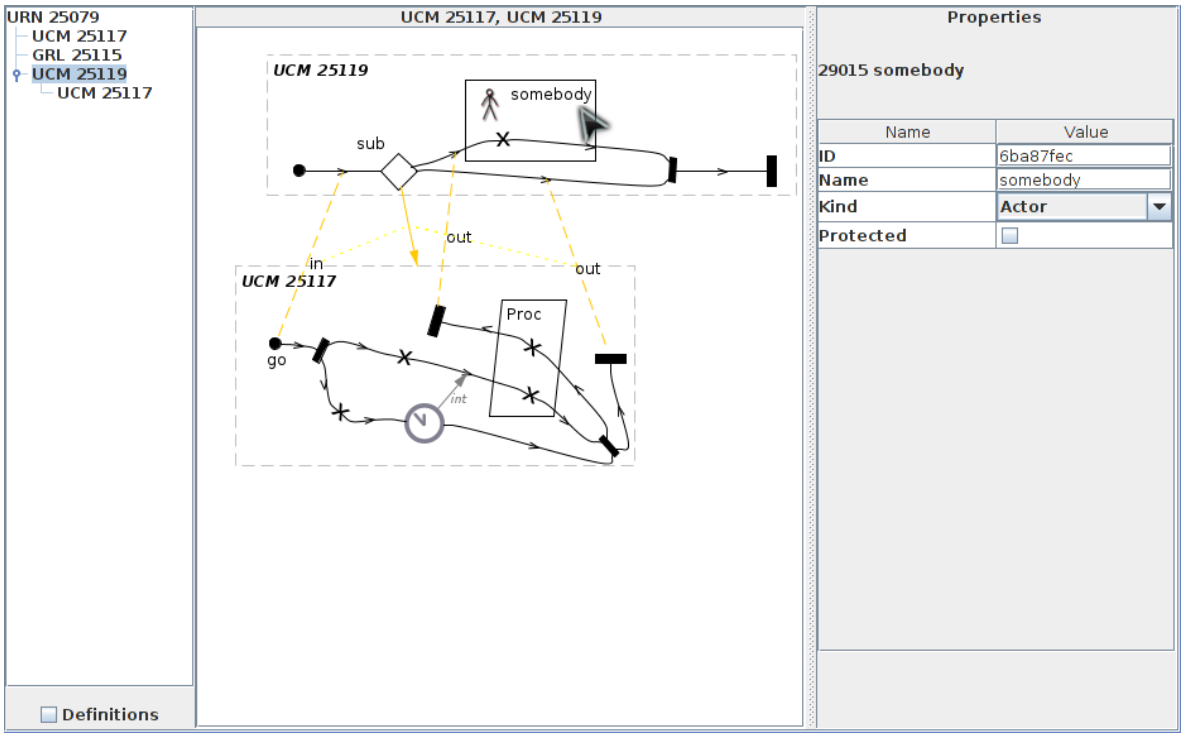

Figure 7.3: Two connected UCM after some transformations

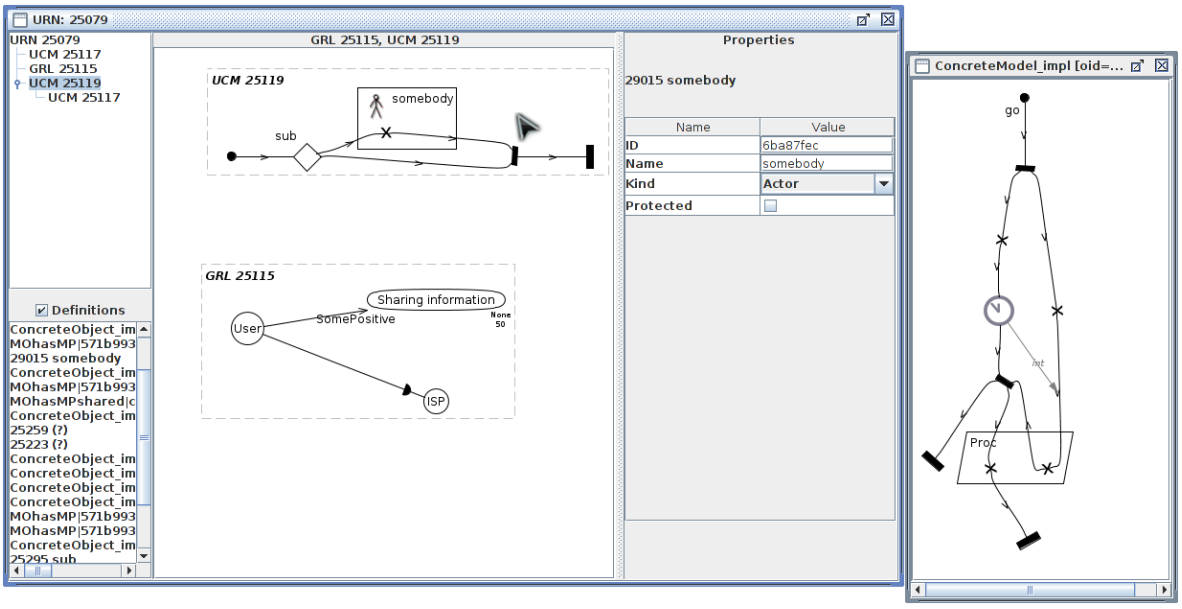

Figure 7.4: MetaDone - edited models

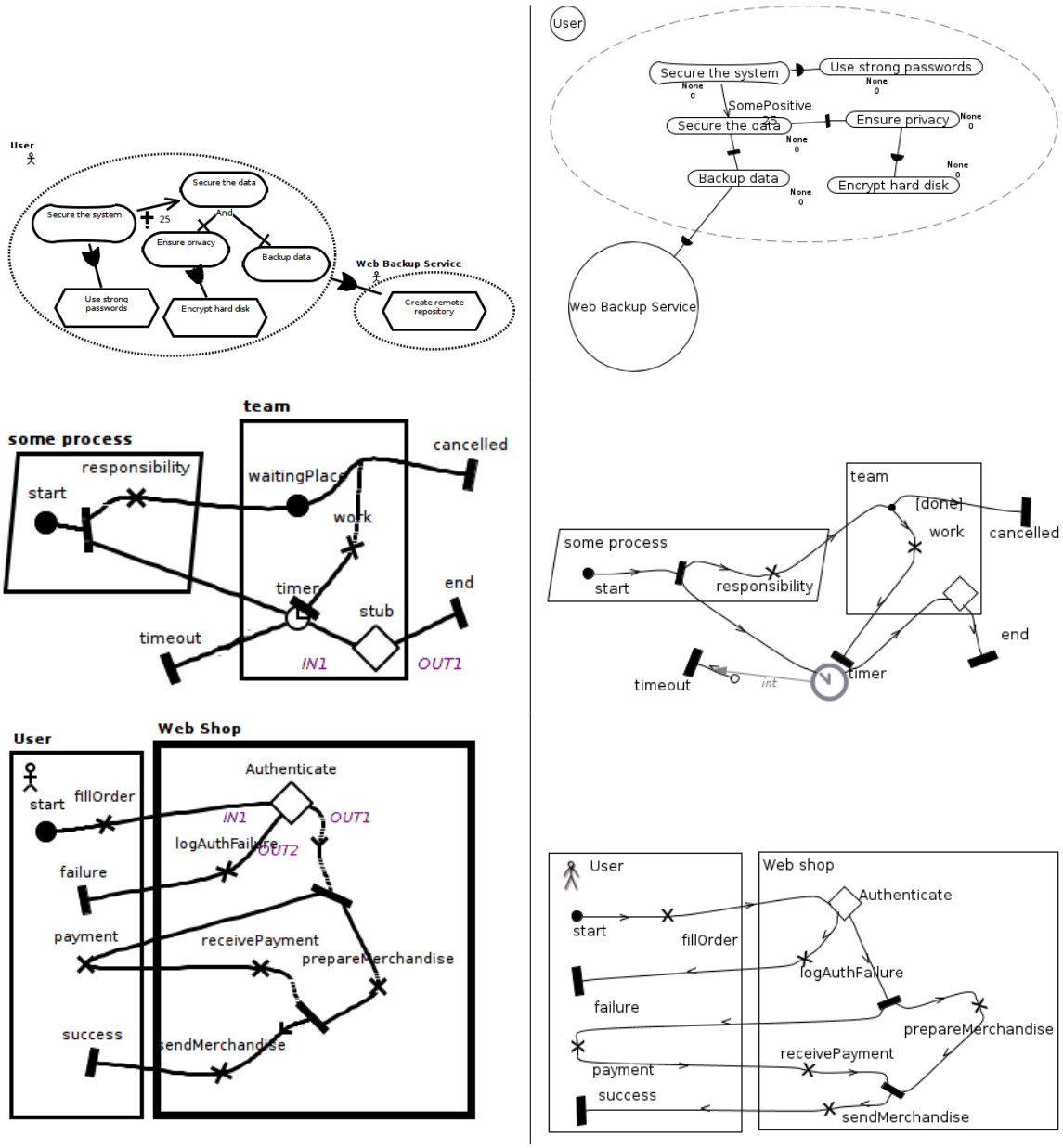

Figure 7.5: Comparing with jUCMnav
On the left side, we have the models exported from *jUCMnav*. On the right side, these from *MetaDone*. The diagrams were copied from the presentation of URN (see 4).

7.2 Critique

Integration with other languages The scripts can be easily composed through the usage of the *import* statement. Building an integrated editor for UCM and GRL was done. The editor works and the script integrating both editors is not very complicated, however we need to set the right model depending on the selected object.

Effort The editors are described with about 1000 lines of Grasyla. This is very little compared to jUCMnav which has about 12000 lines for the editor part (the whole project is more than 200000 lines of code). Moreover, generators could save another part of the work. For a simple meta-model, such as the Petri meta-model, an editor can be created in few minutes. The difficult part is learning how Grasyla works and being able to debug the interface. Error messages were made to be as meaningful as possible when the script fails to be loaded. Moreover, if the rendered result is not exactly what the user wants, they can dump the graph of the engines, with their states and other information, to a DOT file and have more information on what is exactly used.

Graphical completeness The default engines of the Grasyla interpreter allow to render easily graph-based languages. Even complex features like edges connected to edges, components placed on an edge or hierarchical graphs are supported. Text can also be generated and even some geometric-based features like the `near` Grasyla component exist. Moreover, plug-ins can be added to support any kind of representation.

All of the syntax required in URN was implemented with the exception of the timeout edge. A lot of different default shapes and border types are present. Features such as colors, fonts or the way edges are routed from node to node are integrated. It is also quite easy to load images. Nevertheless, the timeout edge was represented with a label on the edge in a different color, drawing a broken edge was not possible and is not also done in jUCMnav; the used library does not offer this feature.

Editor usability A lot of default edition components were provided such as text fields, check boxes, tables, etc. In fact, for most of the components from *Swing* were implemented. Users can specify the editor they exactly want and a lot of variations are possible.

Still, usability is a weak point. The following features are still missing:

- Specifying keyboard shortcuts.
- Dragging an element to put it inside another element. Right now, to add an object inside another, it must first be created and then a link must be created between the container and that object. It would be more easy when objects could just be created directly inside some container, still this can be done using user-defined scripts. Another feature would be to trigger link creation when an object is dragged on top of another one.
- Detecting unrepresented model elements. When an object is not represented directly on the screen, it is difficult to be able to modify it. For example, to remove an object from its container, buttons were used that are shown only if an object is contained in some container and that delete that link on click. However, this is inherent to all graphical editors.
- Fully functional undo/redo. A basic undo/redo functionality was implemented, however it works only in few cases and will have to be improved.

– Constraint checking. The user can draw any model as long as it respects the abstract syntax. However, constraints are not checked.

In other words, the editors looks exactly as we want it to, but features to modify models quickly are still lacking.

Chapter 8

Related Work

This chapter will discuss other existing approaches and frameworks to build graphical user interfaces from high-level specifications.

8.1 Existing approaches for visual language specification

The information contained in models is defined by their meta-models, these are also called their *abstract syntax*. The *concrete syntax* is defined by a visual language specification, which basically is a mapping between meta-objects and their representation. An overview of the existing types of approaches is described in *A Survey of Visual Language Specification and Recognition* [40].

For each of the main types of representations, we are going to describe the related work.

8.1.1 About text-based editors - textual languages

This kind of approach is used for *textual languages* where the grammar of the language can be used to build a text parser for the language, those grammars are also called *string grammars*. The model is serialized in a text file which is parsed to extract the abstract syntax.

Textual representations of models can be in some cases more efficient to implement and to use than graphical ones. A domain-specific language can be understood sometimes more easily as text, because the keywords are more expressive than shapes of which the user have to know the meaning in a particular context. It has been demonstrated [47] that graphical languages are not always more powerful than textual languages. For instance, graphics are often ambiguous and can be interpreted differently by the viewers.

The information shown on the screen by using a graphical representation is more widespread and the user has to zoom or scroll [26]. Generally, the graphical representation is better for an overview of the system and the textual representation is more handy for viewing details. It is also believed that writing code is faster than drawing diagrams because engineers have to constantly switch between the mouse and the keyboard for drawing a diagram and nowadays templates and auto-completion exists for text. Some parts of the model are still textual, such as conditions or notes, however they are often badly integrated into the graphical editor. Moreover, text is platform and tool independent [26]. Text can also be version controlled easily using revision control tools such as SVN. Experiences show that versioning models using tools designed for text can lead to many problems [26]. Changes of the model often lead to large

changed in the serialized version of that model. Moreover, conflict resolving is hard because the serialized text is not always readable.

Xtext for Eclipse [54] is a framework for developing external textual domain-specific languages. That framework brings the possibility to create parsers and editors for DSL's in Eclipse.

Another existing tool is Rascal [36] which is a meta-programming language. It can be integrated into Eclipse and allows to create editors.

Even if MPS (see 2.3.6) is another tool which presents the data in a textual form, it is not a text editor. It is a projectional editor and its models are serialized in an XML format.

8.1.2 Predicate-based approaches

This approach consists in defining rules which map concepts to an element of the syntax. These elements may be composed of other elements or references to other objects in the model which will be represented in their turn by using another rule. The advantage of this method is its simplicity and the fact that the concrete syntax does not need to be stored as a model.

An example implementation is the Moses tool [30, 31]. Their meta-model is an attributed graph. The DSL has mappings for *graph types*, *vertex types*, *edge types* and it also supports *predicates* which check the validity of the model. Their meta-model is hierarchical: a vertices can contain graphs. The tool is simple and can only represent connection-based models.

Another work presents a declarative specification for the ViatraDSM framework [48] which is integrated into Eclipse. The implementation consists in having a diagram model (concrete syntax) and a logical model (abstract syntax) with a mapping between them that keeps them synchronized in both directions; the EMF-based mapping is used. Then, the diagram on the screen is the direct representation of the diagram model.

Tools such as GME, GMF (Eclipse), MetaDone or MetaEdit also use this approach.

8.1.3 Constraint solving

Constraint solving can be useful to synchronize models. One may define that a value in the repository must be equal to the value of a label in a displayed model. This way, the constraint solver of the application will automatically synchronize both models.

In "A Graph Based Framework for the Implementation of Visual Environments" [49], an architecture for storing visual expressions is defined. There are three levels of representation:

- *Abstract Syntax Graph*: Structure of the diagram according to the visual language. Used for the interpretation and for syntax directed editing.
- *Spatial Relations Graph*: Structure of the diagram seen as a picture. Used for layout editing.
- *Physical Layout*: The graphics, defines what the user sees. Used for free editing.

To maintain the correspondences between different layers a constraint solver is used. It can generate and check the physical layout.

8.1.4 Graph rewriting

Graph rewriting approaches are model transformation techniques for transforming a given graph into another graph. These transformations are described using grammars; which are composed of rules that match elements in the input graph to elements in the output graph.

Graph grammars

Graph grammars and triple graph grammars are both described in *Triple Graph Grammars: Concepts, Extensions, Implementations and Application Scenarios* [35] as well as in "Relating SPO and DPO graph rewriting with Petri nets having read, inhibitor and reset arcs." [7]. This section presents a brief summary of what are graph grammars.

Graph grammars are used for graphs whereas BNF is used for text. They were created to generalize Chomsky's string grammars [8]: the Abstract Syntax Tree (AST) can be extracted while parsing text, this approach is generalizable to graphs if we consider that the words are considered as the vertices of the graph.

A graph grammar is a pair (G_0, P) where G_0 is the initial graph and P is the set of production rules. By applying derivations, the transformed graph will be obtained from the initial graph.

Graph grammars are theoretically described using the category theory, a brief description of the implementation is in Appendix C. Single pushout and double pushout for triple graph grammars are described there.

An implementation An implementation has been done in AToM^3 [27]. User actions trigger rules of the grammar that will synchronize the model using the changes in the representation. The rules of the grammar are triggered by user actions, so both the model and its representation are synchronized. This way, the user uses the tool to modify the representation of the model and then, using derivations, the model is kept in synchrony.

Similar approach and its implementation

Another approach which is similar to triple graph grammars is presented in "Making Metamodels Aware of Concrete Syntax" [21] and in *Correctly Defined Concrete Syntax for Visual Modeling Languages* [6]. Meta-models are used to describe the abstract syntaxes and concrete syntaxes are described using models. An intermediary layer between both syntaxes contains *display manager classes* which play a similar role to the interface in triple graph grammars: it is a mapping between meta-objects and their representation (display object).

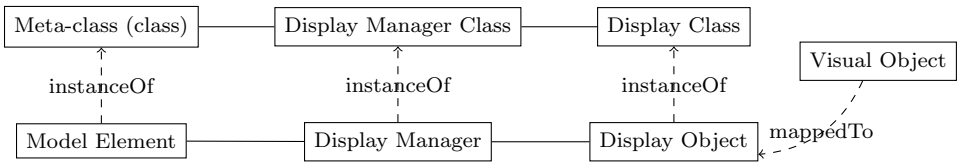

Figure 8.1: Architecture with *display manager classes*

The values in the representation are set using a constraint solver for OCL rules placed in the context of the display manager classes. The constraint solver also works for the concrete syntax by checking rules such as *contained*, *nearby*, *overlaps*, etc.

An example of a graphical rendering using SVG is described in "Graphical Concrete Syntax Rendering with SVG" [20]. The visual syntax is specified using SVG templates. Some constraints and properties are added by extending the SVG format to handle some more node and attribute types.

In his thesis [19], Fondement summarizes this research and defines a concrete syntax for models.

8.2 Frameworks

The development of user interfaces is time-consuming as well as repetitive, parts of the development can be automatized. Some of these frameworks are compared by Kennard and Leaney [34].

Apache Isis

Apache Isis is a Java framework that is the main implementation of the NakedObjects pattern [4]. The entire user interface is generated from the domain objects.

Metawidget

MetaWidget [43] is a smart Java widget which populates its contents by analyzing some domain objects. Basically, it is a Object Interface Mapping (OIM) technology for Java, it can inspect objects and create the user interface from found informations. The full explanation can be found on their website.

UsiXML

UsiXML [51] is a User Interface Description Language (UIDL) which supports multiple levels of independence such as different platforms, context, user languages, etc.

XUL

XUL (XML User interface Language) [55] is a UIDL developed by Mozilla. XUL documents are composed of widget definitions for elements that are presented on the screen. JavaScript is used to handle events and communicate data with a server, this is similar to what a dynamic web page does. XUL only defines the widgets but does not link them to a model.

XForms

XForms [53] is an UIDL designed to be embedded in other markup languages such as HTML, SVG, etc. The standard defines models, controllers and views which reduces the need for scripting.

Final remarks

The existing frameworks are not extensible enough for meta-modeling. For instance, UsiXML and XUL are notations with a fixed set of components, they do not include graphs which are used for all connection-based languages. Apache Isis and Metawidget rely on a definition of the meta-model as Java classes which cannot be done for MetaDone.

Chapter 9

Summary and future work

This last chapter concludes this thesis. We are going to begin by describing the advantages of Grasyla 2 and give some metrics. Secondly, we are going to present the limitations of Grasyla. Thirdly, we are going do give ideas for what can be done to improve further Grasyla or MetaDone. Finally, we are going to summarize this work.

9.1 Implementation

The advantages of Grasyla 2 over Grasyla 1 were already presented in section 5.2. The language is now regular and extensible. It handles multiple models, has an *import* statement and can specify editors. Comparing to other approaches Grasyla has all the features to build simple editors in a few minutes: using common scripts and the editor generator most of the hard work is already done.

Some metrics As of now, MetaDone Java source code is 3 Mo and contains more than 87 kloc. All code related to Grasyla is 33.6 kloc. The table below contains the numbers for some components. Components detailled inside Grasyla 2 are not mutually exclusive and do not include all the classes.

Component	*kloc*	*# classes*	*Notes*
Meta-model	1.6	9	The definition of the meta-model and other classes.
Parser	1.9	19	ANTLR parser for the concrete syntax.
Grasyla 2	30.1	208	All Grasyla interpreter and engine classes.
- Engines	14.1	97	The 5 abstract engines and 92 implementations.
- Swing	4.3	39	Swing engines and helper classes.
- VL	6.6	34	Visual Library engines and helper classes. VL is the library used to create graphs.
- VL-ext	5.6	44	Visual Library extensions to implement new features.

9.2 Limitations of Grasyla

There are a few limitations of what can be done using Grasyla. Even if the language is very generic, it has some constraints and uses a predicate-based approach.

Editing more than one object at the same time Some of the editors, including Eclipse-based editors, have the ability to filter common properties of the selected objects and being able to set them at once. For example, a user can select some shapes and they would be able to set common properties to these objects such as the background color at once. In Grasyla, this is not impossible to build, however it requires equations to be defined in the script for every selectable type and these would have to use scripting to set a property on many objects at once.

Aggregating objects It is not possible to define equations working on many types. For example, a user might want to define a single expression for two linked objects. The workaround is to define an equation for the role between these object and represent it.

No static typing Grasyla is not a statically typed language. This means that type error are found only at runtime. The fact that the meta-model of Grasyla does not define explicitly what kind of arguments are expected for the contained elements makes the language more error-prone.

9.3 Future work

This section will be divided in two parts: the first will discuss what can be improved in Grasyla and the second will discuss MetaDone.

Grasyla

Text parsing Generating a text parser from a Grasyla specification could be written to import models into Grasyla. Some work was already done for Grasyla 1 [39], but changes must be done to see to what extend this is possible to do for Grasyla 2, also no implementation exist yet.

XML serialization Right now, there is no serialization mechanism. Engines that support XML serialization could be added easily to Grasyla. Then, similarly to a GUI generator, a generator for XML could be built. That would allow to store models in Grasyla in XML files. A parser is still needed and can be build based on equations of a Grasyla script, but this is outside of the scope of this thesis.

A specification for a Petri model could look something like:

```
notation "Petri XML"
root Petri
model { root }
$ one metamodel root = xml {
    name: "petri"
    $*def
}
$ one metaobject Place = xml {
    name: "place"
    attributes: xmlattr {
        id: id
        count: value($"Place.count")
    }
}
```

Improving the GUI generator The currently implemented GUI generator provides useful equations when building an editor. Even if the generator works, it can still be improved for generating equations that use other types of widgets. For instance, a default mapping for graph nodes and equations that generate labels that are replaced by a text field when double clicked could be specified to build editors inside the graph.

A generator for a model debugger could also be generated. Such a debugger would be able to access and represent all the elements of a model without necessarily being able to edit everything.

Partial view support Grasyla renders visualizations of a model, rendering a view (representing only a subset of objects of a model) is more difficult to achieve. However, this is possible to do by using a boolean variable for each root element which would tell whether the object is represented or by defining a variable which would be the set of objects to show. Still, the process is not automated.

Components for geometric-based languages The number of engines for geometric-based representation is very limited: a few more could be developed. Events should also be triggered when an object is moved on top of or dragged outside of another object on the screen. For example, this would allow a user to move a class on top of a package and a script could be triggered to add that class to the package.

MetaDone

MetaDone is still a work in progress. This work has added support for specifying easily concrete syntaxes to manipulate and edit existing models.

Constraint checking Being able to check constraints of a model is very important. Most models have constraints that are not restrained by their abstract syntax. A very good example are UCM maps, where most of the elements can have a single ingoing or outgoing path. Specifying minimal cardinalities is not supported in Metal2, however often model objects have required attributes.

Even Grasyla could benefit from a constraint checker. Constraints could be written to ensure that components have the required attributes.

Clean separation between mutators and readers For testing purposes, the code that changes some data is written in Grasyla. However, the role of Grasyla is to specify representations. Ideally, Metal2 should define executable objects that trigger parametrized model transformations inside some transaction. The user or a Grasyla script should be able to alter the model only through these mutators.

Undo/redo Undo/redo is very important for any editor as it is often the case that a user deletes some objects by mistake. An implementation could use the *command pattern* for modifications of the repository, logging these command and replaying a command with an opposite effect would solve the problem.

Key shortcuts Shortcuts are important as they allow to speed up the development process. For a better usability these should be implemented.

9.4 Summary

In this thesis, we have presented an approach that was taken to extend a language for specifying the graphical representations for meta-models. Indeed, Grasyla has been modified to become a very generic language for GUI specification. We have shown how it is possible to define edition components in such a language and how they can be implemented. Furthermore, note that a language like Grasyla can be built for any typed graph. In fact, XSL is similar to Grasyla in many ways.

By implementing this approach, we have shown how a model such as URN can be integrated into MetaDone. That language have a lot of features that may be difficult to implement in some tools, thus it makes us confident that Grasyla can be used to describe more easily model that could not have been specified in other tools.

Bibliography

[1] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. *A Survey on Model Versioning Approaches*. Tech. rep. Johannes Kepler University Linz, 2009. URL: http://smover.tk.uni-linz.ac.at/docs/IJWIS09_paper_Altmanninger.pdf.

[2] Daniel Amyot, Hanna Farah, and Jean-François Roy. "Evaluation of Development Tools for Domain-Specific Modeling Languages". In: *System Analysis and Modeling: Language Profiles*. Ed. by Reinhard Gotzhein and Rick Reed. Vol. 4320. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 183–197. ISBN: 978-3-540-68371-1. DOI: 10.1007/11951148_12.

[3] Daniel Amyot and Gunter Mussbacher. "User Requirements Notation: The First Ten Years, The Next Ten Years (Invited Paper)". In: *Journal of Software* 6.5 (2011). URL: http://ojs.academypublisher.com/index.php/jsw/article/view/0605747768.

[4] *Apache Isis*. URL: http://incubator.apache.org/isis/.

[5] *AToM3 A Tool for Multi-formalism Meta-Modelling*. URL: http://atom3.cs.mcgill.ca/.

[6] Thomas Baar. *Correctly Defined Concrete Syntax for Visual Modeling Languages*. Ed. by Oscar Nierstrasz et al. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 111–125. DOI: 10.1007/11880240_9.

[7] Paolo Baldan, Andrea Corradini, and Ugo Montanari. "Relating SPO and DPO graph rewriting with Petri nets having read, inhibitor and reset arcs." In: *Electr. Notes Theor. Comput. Sci."* (2005), pp. 5–28.

[8] Noam Chomsky. "On certain formal properties of grammars". In: *Information and Control* 2 (1959), pp. 137–167. DOI: 10.1016/S0019-9958(59)90362-6. URL: http://www.diku.dk/hjemmesider/ansatte/henglein/papers/chomsky1959.pdf.

[9] Noam Chomsky. "Three models for the description of language". In: *IRE Transactions on Information Theory* (1956), pp. 113–124. URL: http://www.chomsky.info/articles/195609--.pdf.

[10] Sergey Dmitriev. "Language Oriented Programming: The Next Programming Paradigm". In: *onBoard* (2004).

[11] *Eclipse - The Eclipse Foundation open source community website*. The Eclipse Foundation. URL: http://eclipse.org/.

[12] Vincent Englebert. “A Smart Meta-CASE: Towards an Integrated Solution”. PhD thesis. Computer Science Dept. Rue grandgagnage 21. 5000 Namur. Belgique: University of Namur, 2000.

[13] Vincent Englebert. *Metal1: a formal specification*. Tech. rep. University of Namur, 2006.

[14] Vincent Englebert. *Metal2: a formal specification*. Tech. rep. University of Namur, 2007.

[15] Vincent Englebert. *The MetaDone Architecture: an Overview*. Tech. rep. University of Namur, 2009.

[16] Vincent Englebert and Jean-Luc Hainaut. *Grasyla: Modelling Case Tool GUIs In Meta-CASEs*. 1999.

[17] Vincent Englebert and Patrick Heymans. “Towards More Extensible MetaCASE Tools”. In: *International Conference on Advanced Information Systems Engineering (CAiSE'07)*. Ed. by A.L. Opdhal J. Krogstie and G. Sindre. LNCS 4495. 2007, pp. 454–468.

[18] Vincent Englebert and Krzysztof Magusiak. *The Grasyla language*. Tech. rep. University of Namur, 2011.

[19] Frédéric Fondement. “Concrete Syntax Definition for Modeling Languages”. PhD thesis. École polytechnique fédérale de Lausanne, 2007.

[20] Frédéric Fondement. “Graphical Concrete Syntax Rendering with SVG”. In: *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*. ECMDA-FA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 200–214. ISBN: 978-3-540-69095-5. DOI: 10.1007/978-3-540-69100-6_14.

[21] Frédéric Fondement and Thomas Baar. “Making Metamodels Aware of Concrete Syntax”. In: *In ecmda-fa '05, volume 3748 of LNCS*. Springer, 2005, pp. 190–204.

[22] Martin Fowler. *GUI Architectures*. 2006. URL: http://www.martinfowler.com/eaaDev/uiArchs.html.

[23] Martin Fowler. *Presentation Model*. 2004. URL: http://martinfowler.com/eaaDev/PresentationModel.html.

[24] *GME: Generic Modeling Environment — Institute for Software Integrated Systems*. URL: http://www.isis.vanderbilt.edu/Projects/gme.

[25] *GOPRR*. MetaPHOR group, 1999. URL: http://metaphor.it.jyu.fi/a1goprr.html.

[26] Hans Grönniger et al. *Text-based Modeling*. Braunschweig, Germany: Technische Universitat Braunschweig, 2008.

[27] Esther Guerra and Juan de Lara. “Event-driven grammars: relating abstract and concrete levels of visual languages”. In: *Software and Systems Modeling* 6.3 (2007), pp. 317–347. ISSN: 1619-1366. DOI: 10.1007/s10270-007-0051-2.

[28] ITU-T. *User Requirements Notation (URN) - Language Definition*. Tech. rep. ITU-T Z.151. International Telecommunication Union, 2008. URL: http://jucmnav.softwareengineering.ca/ucm/bin/view/UCM/WebHome.

[29] ITU-T. *User Requirements Notation (URN) - Language requirements and framework*. Tech. rep. ITU-T Z.150. International Telecommunication Union, 2003.

[30] Jörn Janneck and Robert Esser. "A Framework for Defining Domain-Specific Visual Languages". In: *Workshop on Domain-Specific Visual Languages*. 2001.

[31] Jörn Janneck and Robert Esser. "A predicate-based approach to defining visual language syntax". In: *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments*. HCC'01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 40–47. ISBN: 0-7695-0474-4.

[32] *jUCMNav: Juice up your modelling!* URL: http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome.

[33] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008, pp. 160–190. ISBN: 978-0-470-03666-2.

[34] Richard Kennard and John Leaney. "Is there convergence in the field of UI generation?" In: *Journal of Systems and Software* (2011). ISSN: 0164-1212. DOI: 10.1016/j.jss.2011.05.034.

[35] Ekkart Kindler and Robert Wagner. *Triple Graph Grammars: Concepts, Extensions, Implementations and Application Scenarios*. 2007.

[36] Paul Klint, Jurgen Vinju, and Tijs van der Storm. *Rascal - Web Home*. URL: http://www.rascal-mpl.org/.

[37] Benoît Langlois, Consuela-Elena Jitia, and Eric Jouenne. "DSL Classification". In: *Transformation* (2007).

[38] Juan de Lara and Hans Vangheluwe. "Using AToM as a Meta-Case Tool". In: *ICEIS'02*. 2002, pp. 642–649.

[39] Krzysztof Magusiak and Vincent Englebert. *Extending MetaDone Grasyla with text generation and parsing facilities*. Tech. rep. University of Namur, 2011.

[40] Kim Marriott, Bernd Meyer, and Kent B. Wittenburg. *A Survey of Visual Language Specification and Recognition*. New York, NY, USA: Springer-Verlag New York, Inc., 1998. ISBN: 0-387-98367-8.

[41] *Meta Programming System*. JetBrains. URL: http://www.jetbrains.com/mps/.

[42] *MetaCase - MetaEdit+ Modeler DSM Tool*. MetaCase. URL: http://www.metacase.com/mep/.

[43] *Metawidget*. URL: http://metawidget.org/.

[44] OMG. *Meta Object Facility (MOF) Core Specification Version 2.0*. OMG Available Specification. Object Management Group, 2006. URL: http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[45] *OSGi Alliance — Specifications*. OSGi Alliance, 2012. URL: http://www.osgi.org/Specifications/HomePage.

[46] Richard Pawson. "Naked Objects". PhD thesis. University of Dublin, Trinity College, 2004.

[47] Marian Petre. "Why looking isn't always seeing: readership skills and graphical programming". In: *Commun. ACM* 38.6 (1995), pp. 33–44. ISSN: 0001-0782. DOI: 10.1145/203241.203251.

[48] István Ráth. “Declarative Specification of Domain Specific Visual Languages”. MA thesis. Budapest University of Technology and Economics, 2006, pp. 91–108.

[49] J. Rekers and A. Schürr. “A Graph Based Framework for the Implementation of Visual Environments”. In: *IEEE Symp. on Visual Languages*. IEEE Computer Society Press, 1996, pp. 148–155.

[50] *The Protégé Ontology Editor and Knowledge Acquisition System*. URL: http://protege.stanford.edu/.

[51] J. Vanderdonckt et al. “UsiXML: a User Interface Description Language for Specifying Multimodal User Interfaces”. In: *Proc. of W3C Workshop on Multimodal Interaction WMI'2004*. 2004.

[52] Markus Voelter and Konstantin Solomatov. “Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS”. In: *Software Language Engineering, Third International Conference, SLE 2010*. Ed. by Mark van den Brand, Brian Malloy, and Steffen Staab. Lecture Notes in Computer Science. Springer, 2010.

[53] *XForms 1.1*. W3C, 2009. URL: http://www.w3.org/TR/xforms/.

[54] *Xtext*. Eclipse. URL: http://www.eclipse.org/Xtext/.

[55] *XUL - MDN*. Mozilla. URL: https://developer.mozilla.org/En/XUL.

Appendix A

Glossary

Abstract syntax Structure defining the concepts of a language. Besides, the semantics of the language will be defined using the abstract syntax. Meta-models are the abstract syntax of languages.

Concrete syntax A particular encoding of an abstract syntax. In other words, it is a set of rules that define the notation of a language.

Embedded language A language that can be used *as is* inside a host language. Meaning that some constructs of that language will be valid elements in the host language.

Graph grammar A graph grammar is a set of rules used to transform a source graph into a target graph. More information is given in section 8.1.4.

Instance of An object is an *instance of* a type if it contains the attributes defined by that type. In the set theory, o is an *instance of* T if and only if $o \in T$.

Meta-model A model that describes a model.

Meta-object An object present in a meta-model.

Model A simplified representation of a system under study.

Projectional editor (structure editor) A document editor that is aware of the data structure it edits. Instead of letting the user change some concrete syntax and parse it, a projectional editor works directly on the abstract syntax.
Any What You See Is What You Get (WYSIWYG) editor is also a structure editor: it changes some known data structure while presenting it in some user-friendly form.

Semantics The meaning associated to the concepts of a language. See also the definition of the abstract syntax.

Strict meta-modeling Every object must be an instance of exactly one object in the immediate higher meta-level.

Sub-type A subset of a population of some type.

View A view is a graphical representation built from some of the objects of a given model.

Visualization A visualization is a *view* built from all the objects of a model.

Appendix B

Backus-Naur Form used in this document

The Backus-Naur Form (BNF) used in this document borrows notations from regular expressions. The alternative operator | is read as a choice between all the words on the left side until the next alternative operator and those on the right side. Terminals can be expressed in two ways, using a keyword or a PCRE (Perl Compatible Regular Expression).

The root rule or how the spacing or comments are handled will be expressed in a paragraph before the definition of a language.

· rule1 ◀ rule2 (r0)$^{?}$ (r1)* (r2)$^{+}$ (r3)$^{1\text{-}5}$

· rule2 ◀ **abc** (**def** rule3 | **other**)$_{\leftarrow\text{--this is a group}}$

· rule3 ◀ `[1-9][0-9]*`

In the example language above:

- *rule1* is composed of *rule2* followed by four other rules
  - *r0* is optional: 0 or 1 occurrences
  - *r1* can be repeated: 0 to ∞ occurrences
  - *r2* has multiple occurrences: 1 to ∞
  - *r3* has a specified number of occurrences
- *rule2* is a keyword "abc" followed by a named group which is an alternative between "def" followed by *rule3* and "other"; the "this is a group" is just a comment related to the pointed group
- *rule3* is composed of a terminal expressed as a regular expression

Appendix C

Graph rewriting

This chapter gives an overview of how graph grammars and triple graph grammars are implemented by giving the theoretical grounds.

Used category theory notions

Category theory allows to understand studied objects by describing the relations with others objects. A category C is given by a collection of *objects* and *morphisms* (also called arrows). Each morphism is a function $f : X \to Y$ where X is the domain and Y is the codomain; they can be composed, the composition is associative. For each object there exists an identity morphism (id_X). An example of a category is the partial order $(P, \leq)$ where objects are the elements of the partial order, morphisms represent the relation ($\leq$).

Functors If we create a category where objects are categories and the morphisms are mappings between categories, these homomorphisms are called *functors*. Formally, if F is a functor, we have $F(id_X) = id_{FX}$ and $F(g \circ f) = F(g) \circ F(f)$; functors preserve all commuting diagrams.

Universal constructs

The following constructs describe general properties that apply to objects of a category. Each of these constructs have a *dual* that can be built by reversing all the morphisms.

Pushout and pullback A *pushout* for two morphisms $f : A \to B$ and $g : A \to C$ is an object D and the morphisms h and i such that the square commutes:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle h} \\ C & \xrightarrow{i} & D \end{array}$$

Single pushout (SPO)

A derivation consist of matching the left-hand side L of a rule with a part of a graph G and replace that match by the right-hand side R. The application of derivations defines a dynamic evolution of a single graph.

The initial graph is transformed using derivations which transform the abstract syntax nodes into concrete syntax nodes using the production rules.

More formally, given a graph G, a production $q : L \xrightarrow{r} R$ and a *match* $g : L \to G$, a direct derivation is $\delta : G \Rightarrow_q H$ if for morphisms d and h the following diagram is a pushout square.

$$\begin{array}{ccc} L & \xrightarrow{r_q} & R \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle h} \\ G & \xrightarrow{d} & H \end{array}$$

Double pushout (DPO) - triple graph grammars

Triple graph grammars define a relation between two models instead of an evolution of a single model. They consist of three parts: the source model, the target model and the correspondence graph (also called the interface). Such a grammar can be used to derive one of the given models from the other or to synchronize already existing models.

For K, the correspondence graph, and two parts of a model L and R, a production rule $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ is composed of a name p and a pair of morphisms $l : K \to L$ and $r : K \to R$.

Given a graph G, a production p and a *match* $g : L \to G$, a direct derivation is $\delta : G \Rightarrow_p H$ if the following diagram can be constructed where both squares can be pushouts.

$$\begin{array}{ccccc} L & \xleftarrow{l_p} & K & \xrightarrow{r_p} & R \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle h} & & \downarrow{\scriptstyle k} \\ G & \xleftarrow{b} & D & \xrightarrow{d} & H \end{array}$$

Note that both l and r are morphisms and the diagram is symmetrical, indeed the notion of source/target model are interchangeable. So, it is possible to build the source model from the target model too by finding a match $k : R \to H$.

Appendix D

Available Grasyla equations

D.1 Default equations

Some scripts have been defined to ease the development of new scripts. They contain generic rules that can be used by any model.

Common rules

```
notation "Common"
root @CommonMetaModel

description s{{{
A set of rules that are often handy to build other scripts.
}}}

// default representations
$ no metaobject @CommonMetaObject = none
$ many metaobject @CommonMetaObject = list

// counts
$ empty no metaobject @CommonMetaObject = true
$ empty one metaobject @CommonMetaObject = false
$ empty many metaobject @CommonMetaObject = false
$ exists no metaobject @CommonMetaObject = false
$ exists one metaobject @CommonMetaObject = true
$ exists many metaobject @CommonMetaObject = true
$ count no metaobject @CommonMetaObject = 0
$ count one metaobject @CommonMetaObject = 1
$ count many metaobject @CommonMetaObject = grv{s{{{
    context.getObject().size()
}}}}

// value
$ value one metaproperty @CommonMetaObject = $*self
$ value one metaobject @MetaObject = $@<MetaProperty.name>
$ value_force one metaobject @CommonMetaObject = value($*self)
$ value_force no metaobject @CommonMetaObject = "(?)" {
    color: "gray"
    font: font {
        italic: true
    }
}
$ bool no metaobject @CommonMetaObject = false
$ bool one metaobject @CommonMetaObject = true
$ bool one metaproperty @CommonMetaObject = if {
    guard {condition: value($*self) true}
    guard {false}
```

```
}

// names
$ id one metaobject @CommonMetaObject = id
$ name no metaobject @CommonMetaObject = ""
$ name one metaobject @CommonMetaObject = name
$ name many metaobject @CommonMetaObject = list {
    separator: ", "
}

// enum
$ enumRestricted one metaobject @MetaProperty = bool($@<Enum Restricted>)
$ enumString one metaobject @MetaProperty = value($@<Enum String>)
$ enumInteger one metaobject @MetaProperty = value($@<Enum Integer>)
$ enumFloat one metaproperty @MetaProperty = value($@<Enum Float>)
$ enumChar one metaproperty @MetaProperty = value($@<Enum Char>)

// menu
$ menucreate one metaobject @CommonMetaObject = none
$ menudelete no metaobject @CommonMetaObject = none
```

Property rules

```
notation "Common Properties"
root @CommonMetaModel
import {
    "Common"
}

description s{{{
Rules to build effectively:
- property tables
- definition trees
}}}

// PROPERTY TABLES

$ properties one metaobject @CommonMetaObject = boxV {
    boxH {
        name($*self)
        spring
    }
    property_table($*self)
    spring
}
$ properties one metaproperty @CommonMetaObject = none
$ properties many metaobject @CommonMetaObject = scroll {
    <view>: boxV{ list }
}

$ property_table no metaobject @CommonMetaObject = "No properties" {
    font: font {
        italic: true
    }
}
$ property_table one metaobject @CommonMetaObject = table {
    columns: [ "Name" "Value" ]
    property_tr($*self)
}

$ property_tr one metaobject @CommonMetaObject = none

// DEFINITION TREES

$ buildtree_def one metaobject @CommonMetaObject = none
$ buildtree_def one metamodel @CommonMetaObject = tree_def($*self)
```

```
$ tree_def one metaproperty @CommonMetaObject = none
$ tree_def one metaobject @CommonMetaObject = tree {
    label: tree_def_label($*self)
}
$ tree_def one metamodel @CommonMetaObject = tree {
    showRoot: false
    label: tree_def_label($*self)
    $*def
}
$ tree_def_label one metaobject @CommonMetaObject = name
```

Graph rules

```
notation "Common Graph"
root @CommonMetaModel
import {
    "Common Properties"
}

description s{{{
Rules for building graphs.
}}}

$ graphpanel one metamodel @CommonMetaObject = panelBorder {
    var selected
    center: split {
        first: graph($*self)
        second: boxV{ properties(val selected) }
        position: 0.7
    }
}

$ graph one metamodel @CommonMetaObject = graph {
    animate: true
    align: val graph_align
    graphnodes($*self)
    action: "select" {grv{s{{{
        def sel = event.getSelection().collect{
            it.getContext().getObject().iterator().toList()
        }.flatten() as Set
        context.setVariable("selected", sel)
        "consume"
    }}}}}
    action: "contextualMenu" {
        menucreate($*self)
    }
}
$ graphnodes one metamodel @CommonMetaObject = rootnode($*def)

$ rootnode one metaobject @CommonMetaObject = node($*self)
$ node one metaobject @CommonMetaObject = none
```

D.2 Generated equations

This section describes what is generated for user interfaces. For a chosen meta-model, the generated script will be called *model-name* (properties). We are going to use the notation <var> for variables.

To summarize, the choice of edition components is based on the cardinality of the property and the type of the object. For example, a rule generating a check box will be produced for a boolean property.

Meta-role editors For every meta-role an equation is generated to edit it. The expression is a table containing existing linked objects and buttons to add and remove roles.

```
// for every meta-role (<mr>)
$ edit_<domain>_<mr> one metarole <domain> = ...
```

Property editors For every meta-property contained in a meta-object edition equations of the given form are generated.

```
// for every meta-object (<mo>) and meta-property (<prop>)
$ edit_<mo>_<prop> one metaobject <mo> = ...
```

- By default a *text field* is generated.
- *Check boxes* are built for boolean types.
- *Spinners* are used for numbers.
- *Combo boxes* are used for values that are enumerated.
- For attributes with a cardinality different from 1, a list is built.

Table rows Table rows are generated for property tables. `<super>` represents the direct super-types of the current meta-object.

```
// for every <mo>, these equations are generated:

$ property_tr one metaobject <mo> = property_tr_<mo>($*self)

// equation for each meta-property (<prop>)
$ property_tr_<mo>_<prop> one metaobject <mo> = tr {
    "<prop>"
    edit_<mo>_<prop>($*self)
}
// equation for each meta-role (<mr>)
$ property_tr_<mr> one metaobject <mo> = tr {
    "<mr>"
    edit_<mo>_<mr>($*self)
}
$ property_tr_<mo> one metaobject <mo> = [
    // for each <super> {
        property_tr_<super>($*self)
    // }
    // for each <prop> {
        property_tr_<mo>_<prop>($*self)
    // }
    // for each <mr> {
        property_tr_<mr>($*self)
    // }
]
```

Create menus A default contextual menu is defined for the model to create object instances.

```
$ menucreate one metamodel <model> = ...
```

Appendix E

URN: Grasyla scripts

The following Grasyla scripts were defined to build the editor presented in section 7.1. The script is separated in four parts: a small set of common definitions, GRL Editor, UCM Editor and URN Editor (built by importing GRL and UCM editors and adding rules).

File	Lines
URN Common	25
GRL	273
UCM	449
URN	289
total	1036

URN Common Definitions

```
notation "Common URN"
root @URN
import {
    "Common Graph"
}
description s{{{
Common definitions for URN.
}}}

$ rootnode one metaobject @CommonMetaObject = none
$ rootnode one metarole @CommonMetaObject = node($*self)


// PROPERTIES

$ property_tr_URNElement one metaobject URNmodelElement = [
    tr { "ID" textfield{
        value($id)
        action: "validate" {update{$id}}
    }}
    tr { "Name" textfield{
        value($name)
        action: "validate" {update{$name}}
    }}
]
```

GRL Editor

```
notation "GRL graph"
root @URN@GRL
import {
    "Common URN"
}
model { GRLgraph }
functor "graphpanel"
requires {
    "option: auto-edit"
}

description s{{{
Represents a single GRL graph.
}}}

define noBorder = "line" {
    color: "red"
    margin: 2
}

// MENU

$ menucreate one metamodel root = menu {
    label: "Create"
    menuitem {
        label: "Intentional element"
        action: "click" {
            create { $IntentionalElement } "consume"
        }
    }
    menuitem {
        label: "Actor"
        action: "click" {
            create{ $Actor } "consume"
        }
    }
}

// GRAPH

$ rootnode one metaobject Actor = node($*self)
$ node one metaobject Actor = nodeexpandable($*self)
$ nodeexpandable one metaobject Actor = rectangle {
    border: "line" {
        color: "transparent"
    }
    var[view] expand = true
    action: "doubleClick" {toggle{val expand}}
    if { guard { condition: val expand
        ellipsis {
            // avoid painting behind the circle
            boxV { space { height: 10 } boxH { space { width: 10 }
                free { node($ActorIntention.range) }
            }}
            border: "line" {
                stroke: "dashed"
                color: "gray"
            }
        }
    }}
    nodecollapsed($*self)
    align: "top_left"
}
$ nodecollapsed one metaobject Actor = circle {
    value_force($name)
```

```
    background: "white"
    bitt: "shape" {
        for: $ElementLink.domain
        both_sides: true
    }
    bitt: "shape" {
        for: $ElementLink.domain
        both_sides: true
        object: if { guard { condition: not { val expand } $ActorIntention.range }}
    }
}

$ rootnode one metaobject IntentionalElement = if {
    guard {
        condition: empty($ActorIntention.domain)
        node($*self)
    }
}
$ node one metaobject IntentionalElement = near {
    boxV {
        value($<IntentionalElement.importance>)
        boxH { spring value($<IntentionalElement.importanceQuantitative>) spring }
        font: font {
            size: 8
        }
    }
    if { guard { condition: not { match { "AND" value($<IntentionalElement.decompositionType>) } } }
        value($<IntentionalElement.decompositionType>)
    }}
    boxV {
        border: IntentionalBorder($<IntentionalElement.type>)
        value_force($name)
        bitt: "rectangular" {
            for: $ElementLink.domain
            both_sides: true
        }
    }
}
$ IntentionalBorder no metaobject <IntentionalElement.type> = &noBorder
$ IntentionalBorder one metaproperty <IntentionalElement.type> = if {
    guard {
        condition: match { $*self "Goal" }
        "rounded" {
            apexLeft: 10
            apexRight: 10
        }
    }
    guard {
        condition: match { $*self "SoftGoal" }
        "rounded" {
            apexLeft: 10
            apexRight: 10
            apexTop: -3
            apexBottom: -3
        }
    }
    guard {
        condition: match { $*self "Task" }
        "pointing" {
            apexLeft: 10
            apexRight: 10
        }
    }
    guard {
        condition: match { $*self "Resource" }
        "line" {
```

```
            margin: 3
        }
    }
    guard {
        condition: match { $*self "Belief" }
        "rounded" {
            apexLeft: 10
            apexRight: 10
            apexTop: 3
            apexBottom: 3
        }
    }
    guard {
        &noBorder
    }
}

$ node one metarole ElementLink = edge {
    targetShape: if {
        guard {
            condition: match { value($<ElementLink.type>) "Contribution" }
            "arrow"
        }
        guard {
            condition: match { value($<ElementLink.type>) "Dependency" }
            "sequence" {
                "void" {
                    radius: 16
                    draw: true
                }
                "bent" {
                    inside: false
                    filled: true
                }
            }
        }
        guard {
            condition: match { value($<ElementLink.type>) "Decomposition" }
            "sequence" {
                "void" {
                    radius: 16
                    draw: true
                }
                "bar" { width: 3 }
            }
        }
    }
    stroke: if {
        guard { condition: [
            match{ value($<ElementLink.type>) "Contribution" }
            bool($<ElementLink.contribution.correlation>)
            ]
            "dashed"
        }
        guard {
            "solid"
        }
    }
    if { guard { condition: match { value($<ElementLink.type>) "Contribution" }
        boxV {
            value($<ElementLink.contribution.type>)
            boxH {spring value($<ElementLink.contribution.quantitative>) spring}
            color: "transparent"
            decorator_position: 0.9
            decorator_align: "center_target"
        }
```

```
    }}
}

// PROPERTIES

$ property_tr one metaobject Actor = property_tr_URNElement($*self)

$ property_tr one metaobject IntentionalElement = [
    property_tr_URNElement($*self)
    tr { "Type" combobox {
        editable: false
        selected: value($<IntentionalElement.type>)
        action: "validate" {update{$<IntentionalElement.type>}}
        enumString(meta{$<IntentionalElement.type>})
    }}
    tr { "Decomposition" combobox {
        editable: false
        selected: value($<IntentionalElement.decompositionType>)
        action: "validate" {update{$<IntentionalElement.decompositionType>}}
        enumString(meta{$<IntentionalElement.decompositionType>})
    }}
    tr { "Importance" boxV {
    combobox {
        editable: false
        selected: value($<IntentionalElement.importance>)
        action: "validate" {update{$<IntentionalElement.importance>}}
        enumString(meta{$<IntentionalElement.importance>})
    }
    textfield {
        value($<IntentionalElement.importanceQuantitative>)
        action: "validate" {update{$<IntentionalElement.importanceQuantitative>}}
    }
    }}
    if { guard { condition: not { empty($<ActorIntention>.domain) }
        tr { "Actor" panel {
            button {
                label: "Remove"
                tooltip: "Remove from the actor"
                action: "click" {
                    delete_role{ $ role <ActorIntention>.domain }
                    "consume"
                }
            }
        }}
    }}
]

$ property_tr one metaobject ElementLink = [
    property_tr_URNElement($*self)
    tr { "Link" combobox {
        editable: false
        selected: value($<ElementLink.type>)
        action: "validate" {update{$<ElementLink.type>}}
        enumString(meta{$<ElementLink.type>})
    }}
    if { guard { condition: match{ value($<ElementLink.type>) "Contribution" }
        tr { "Contribution" boxV {
            boxH { "Type" combobox {
                editable: false
                selected: value($<ElementLink.contribution.type>)
                action: "validate" {update{$<ElementLink.contribution.type>}}
                enumString(meta{$<ElementLink.contribution.type>})
            }}
            boxH { "Quantity" textfield {
                value($<ElementLink.contribution.quantitative>)
                action: "validate" {update{$<ElementLink.contribution.quantitative>}}
```

```
            }}
            checkbox {
                label: "Correlation"
                value($<ElementLink.contribution.correlation>)
                action: "validate" {update{$<ElementLink.contribution.correlation>}}
            }
        }}
    }}
]
```

UCM Editor

```
notation "UCM map"
root @URN@UCM
import {
    "Common URN"
}
model { UCMmap }
functor "graphpanel"
requires {
    "option: auto-edit"
}

description s{{{
Represents a single UCM map.
}}}

// MENU

$ menucreate one metamodel root = menu {
    label: "Create UCM element"
    menuitem {
        label: "Resp."
        tooltip: "Responsibility reference"
        action: "click" {create{$RespRef}}
    }
    menuitem {
        label: "Stub"
        action: "click" {create{$Stub}}
    }
    menuitem {
        label: "Start and End"
        action: "click" {create{$StartPoint} create{$EndPoint}}
    }
    menu {
        label: "Fork/Join"
        menuitem {
            label: "AND fork"
            action: "click" {create{$ForkAND}}
        }
        menuitem {
            label: "AND join"
            action: "click" {create{$JoinAND}}
        }
        menuitem {
            label: "OR fork"
            action: "click" {create{$ForkOR}}
        }
        menuitem {
            label: "OR join"
            action: "click" {create{$JoinOR}}
        }
    }
    menuitem {
        label: "EmptyPoint"
        action: "click" {create{$EmptyPoint}}
    }
    menuitem {
        label: "Waiting"
        action: "click" {create{$WaitingPlace}}
    }
    menuitem {
        label: "Timer"
        action: "click" {create{$Timer}}
    }
    menuitem {
        label: "Component"
```

```
        action: "click" {create{$ComponentRef}}
    }
}

// GRAPH

$ rootnode one metaobject PathNode = if { guard {
    condition: empty($<ComponentRef.nodes>.domain)
    node($*self)
}}

$ noConnect one metaobject @CommonMetaObject = true
$ noConnect one metaobject Connect = false
$ unconnected no metaobject @CommonMetaObject = true
$ unconnected one metaobject @CommonMetaObject = true
$ unconnected one metaobject PathNode = [
    noConnect($<NodeConnection>.domain)
    noConnect($<NodeConnection>.range)
]

$ node one metaobject PathNode = if { guard {
    condition: unconnected($*self)
    rendernode($*self)
}}

$ rendernode one metaobject PathNode = near {
    value($name)
    rendernodeshape($*self)
}

$ rendernodeshape one metaobject PathNode = boxH {
    "N" id
    bitt: "rectangular" {
        for: $NodeConnection.domain
        both_sides: true
    }
}

$ rendernodeshape one metaobject StartPoint = circle {
    space {
        width: 6
        height: 6
    }
    background: "black"
    bitt: "shape" {
        for: $NodeConnection.domain
        both_sides: true
    }
    bitt: "shape" {
        for: $InBinding.range
    }
}

$ rendernodeshape one metaobject EndPoint = rotate {
    rectangle {
        background: "black"
        space { width: 8 height: 25 }
        bitt: "shape" {
            for: $OutBinding.range
        }
    }
    bitt: "rotated" {
        for: $NodeConnection.domain
        both_sides: true
    }
}
```

```
$ rendernodeshape one metaobject EmptyPoint = circle {
    space {
        width: 4
        height: 4
    }
    bitt: "shape" {
        for: $NodeConnection.domain
        both_sides: true
    }
}

$ rendernodeshape one metaobject RespRef = rotate {
    "X" {
        font: font {
            bold: true
            size: 15
        }
    }
    bitt: "rotated" {
        for: $NodeConnection.domain
        both_sides: true
        direction: "center"
    }
}

$ rendernodeshape one metaobject WaitingPlace = "waiting" {
    bitt: "rectangular" {
        for: $NodeConnection.domain
        both_sides: true
    }
}

$ rendernodeshape one metaobject Timer = image {
    resource: "urn:image:timer"
    bitt: "circular" {
        radius: 16
        for: $NodeConnection.domain
        both_sides: true
    }
    bitt: "circular" {
        radius: 16
        for: $<NodeConnection.timer>.domain
    }
}

$ rendernode one metaobject Connect = near {
    $NodeConnection.domain
    $NodeConnection.range
}

$ rendernodeshape one metaobject JoinAND = rendernodeAND($*self)
$ rendernodeshape one metaobject ForkAND = rendernodeAND($*self)
$ rendernodeAND one metaobject PathNode = rotate {
    rectangle {
        background: "black"
        space { width: 6 height: 20 }
    }
    bitt: "rotated" {
        for: $NodeConnection.domain
        both_sides: true
        direction: "right"
    }
}
$ rendernodeshape one metaobject JoinOR = rendernodeOR($*self)
$ rendernodeshape one metaobject ForkOR = rendernodeOR($*self)
```

```
$ rendernodeOR one metaobject PathNode = rotate {
    circle {
        background: "black"
        space { width: 3 height: 3 }
    }
    bitt: "rotated" {
        for: $NodeConnection.domain
        both_sides: true
        direction: "right"
    }
}

$ rendernodeshape one metaobject Stub = rhomb {
    if {
        guard { condition: bool($<Stub.synchronizing>) "S" }
        guard { condition: bool($<Stub.blocking>) "B" }
        guard { " " }
    }
    border: "line" {
        stroke: if {
            guard { condition: bool($<Stub.dynamic>)
                "dashed"
            }
        }
    }
    bitt: "shape" {
        for: $NodeConnection.domain
        both_sides: true
    }
    bitt: "shape" {
        for: $PluginBinding.domain
    }
}

$ rootnode one metaobject ComponentRef = if {
    guard {
        condition: empty($<ComponentRef.parent>.range)
        node($*self)
    }
}

$ node one metaobject ComponentRef = boxV {
    boxH {
        if { guard { condition: match { "Actor" kind($*self) }
            image {
                resource: "urn:image:actor"
            }
        }}
        value_force($name)
    }
    free {
        $<ComponentRef.nodes>.range
        $<ComponentRef.parent>.domain
    }
    border: nodeborder($<ComponentRef.def>.range)
}
$ kind one metaobject ComponentRef = $<ComponentRef.def>.range
$ kind one metaobject Component = value($kind)
$ kind no metaobject @CommonMetaObject = "none"
$ nodeborder no metaobject Component = "line" {
    stroke: "dotted"
    margin: 3
}
$ nodeborder one metaobject Component = if {
    guard { condition: match { "Process" kind($*self) }
        "parallelogram" {
```

```
            margin: 3
        }
    }
    guard { condition: match { "Object" kind($*self) }
        "rounded" {
            apexLeft: 10
            apexRight: 10
            margin: 3
        }
    }
    guard {
        "line" {
            margin: 3
        }
    }
}

$ node one metarole NodeConnection = edge {
    rotate { ">" decorator_align: "center" }
    router: "curved"
    if {
        all: true
        guard { condition: not {empty($label)}
            boxH {
                "[" value($label) "]"
                decorator_position: 0.2
            }
        }
        guard { condition: not {empty($InBinding.range) empty($OutBinding.range)}
            space {
                bitt: "center" {
                    for: [ $InBinding.domain $OutBinding.domain ]
                }
            }
        }
        guard {
            space {
                bitt: "center" {
                    for: $<NodeConnection.timer>.range
                }
            }
        }
    }
}
$ node one metarole <NodeConnection.timer> = edge {
    targetShape: "triangle"
    color: "gray"
    "int" {
        font: font {
            italic: true
            size: 0.8
        }
        color: "gray"
    }
}

// PROPERTIES

$ property_tr one metaobject Stub = [
    property_tr_URNElement($*self)
    tr { "Dynamic" checkbox {
        selected: bool($<Stub.dynamic>)
        action: "validate" {
            update{$<Stub.dynamic>}
        }
    }}
```

```
    tr { "Synchronizing" checkbox {
        selected: bool($<Stub.synchronizing>)
        action: "validate" {
            update{$<Stub.synchronizing>}
        }
    }}
    tr { "Blocking" checkbox {
        selected: bool($<Stub.blocking>)
        action: "validate" {
            update{$<Stub.blocking>}
        }
    }}
]

$ property_tr one metaobject ComponentRef = [
    property_tr_URNElement($*self)
    if { guard { condition: not { empty($<ComponentRef.parent>.range) }
        tr { "Parent" panel {
            button {
                label: "Remove"
                tooltip: concat {
                    "Remove from " id($<ComponentRef.parent>.range)
                }
                action: "click" {
                    delete_role{ $ role <ComponentRef.parent>.range }
                    "consume"
                }
            }
        }}
    }}
    $<ComponentRef.def>.range
]
$ property_tr no metaobject Component = tr { "No component"
    button {
        label: "Create"
        action: "click" {grv{s{{{
            def ctx = context.getParentContext()
            while (ctx.getObject().isEmpty()) {
                ctx = ctx.getParentContext()
            }
            def obj = ctx.getObject().getHead()
            def comp = concretemodel.createObject(metamodel.getMOByName("Component"))
            def mr = metamodel.getMOByName("ComponentRef.def").narrow2MetaRole()
            concretemodel.createRole(mr, obj, comp)
        }}}}}
    }
}
$ property_tr one metaobject Component = [
    tr { "Kind" combobox {
        editable: false
        selected: value($<kind>)
        action: "validate" {update{$<kind>}}
        enumString(meta{$<kind>})
    }}
    tr { "Protected" checkbox {
        selected: bool($<Component.protected>)
        action: "validate" {update{$<Component.protected>}}
    }}
]

$ properties many metaobject PathNode = boxV {
    "Connect nodes"
    concat { id(list {
        separator: ", "
    })}
    button {
```

```
        label: "Create connect"
        enabled: unconnected($*self)
        action: "click" {grv{s{{{
            def mr = metamodel.getMOByName("NodeConnection").narrow2MetaRole()
            def c = concretemodel.createObject(metamodel.getMOByName("Connect"))
            context.getObject().iterator().toList().each{ it ->
                concretemodel.createRole(mr, c, it)
            }
        }}}}}
    }
}

$ property_tr one metaobject PathNode = [
    property_tr_URNElement($*self)
    if { guard { condition: not { unconnected($*self) } tr {
        "Connect"
        button {
            label: "Delete"
            tooltip: "Remove all connect elements"
            action: "click" {grv{s{{{
                def obj = self.getHead()
                def mr = metamodel.getMOByName("NodeConnection").narrow2MetaRole()
                def mc = metamodel.getMOByName("Connect")
                obj.getAllConcreteRolesDomainCO(mr).each{ it ->
                    if (it.hasType(mc)) it.delete()
                }
                obj.getAllConcreteRolesRangeCO(mr).each{ it ->
                    if (it.hasType(mc)) it.delete()
                }
            }}}}}
        }
    }}}
]

$ property_tr one metaobject NodeConnection = [
    tr { "Label" boxH {
        textfield {
            value($label)
            action: "validate" {update{$label}}
        }
        button {
            label: "Delete"
            visible: not{empty($label)}
            action: "click" {delete{$label}}
        }
    }}
]

$ property_tr one metaobject PluginBinding =
    tr { "ID" textfield{
        value($id)
        action: "validate" {update{$id}}
    }}
```

Integrated URN editor

```
notation "URN editor"
root @URN
model { root }
import {
    "GRL graph"
    "UCM map"
}
functor "urneditor"
requires {
    "option: auto-edit"
    "option: no-auto-create"
}

description s{{{
Editor for URN.
Both UCM and GRL can be edited.
}}}

define modelBorder = "rectangle" {
    color: "lightGray"
    stroke: "dashed"
    margin: 4
}

define modelFont = font {
    bold: true
    italic: true
}

// MAIN EDITOR

$ urneditor one metamodel root = panelBorder {
    var models
    var models_selection
    var selection

    // Model panel
    left: urneditor_models($*self)

    // Main graph and properties
    center: split {
        position: 0.7
        first: boxV {
            boxH { name(val models) }
            graph($*self)
        }
        second: boxV {
            "Properties"
            properties_urn(val selection)
        }
    }
}

$ urneditor_models one metamodel root = boxV {
    tree {
        showRoot: true
        editable: false
        label: name($*self)
        tree_model($*def)
        action: "select" {grv{s{{{
            def sel = event.getSelection().collect{
                it.getContext().getObject().iterator().toList()
            }.flatten() as Set
            context.setVariable("models_selection", sel)
            "consume"
```

```
        }}}}}
        action: "contextualMenu" {
            menuitem {
                label: "View selected"
                action: "click" { set{ val models value: val models_selection } }
            }
            menuitem {
                label: "Hide selected"
                action: "click" {grv{s{{{
                    def selection = context.getVariable("models_selection")
                    if (selection == null) return
                    def old = context.getVariable("models")
                    if (old == null) return
                    old -= selection
                    context.setVariable("models", old)
                    "consume"
                }}}}}
            }
            menuitem {
                label: "Create GRL"
                action: "click" {create{$GRL@GRLgraph} "consume"}
            }
            menuitem {
                label: "Create UCM"
                action: "click" {create{$UCM@UCMmap} "consume"}
            }
        }
        action: "doubleClick" {grv{s{{{
            def selection = context.getVariable("models_selection")
            if (selection == null) return
            def old = context.getVariable("models")
            if (old != null) {
                selection += old
            }
            context.setVariable("models", selection)
            "consume"
        }}}}}
    }
    space { height: 10 }
    var show_definitions = false
    checkbox {
        label: "Definitions"
        action: "validate" {toggle{val show_definitions}}
    }
    if { guard { condition: val show_definitions
        buildtree_def(val models_selection)
    }}
}

$ tree_model one metaobject @CommonMetaObject = none
$ tree_model one metamodel root = tree {
    label: name($*self)
    $*def
}
$ tree_model one metaobject UCM@Stub = $UCM@PluginBinding.range

$ buildtree_def many metamodel @CommonMetaObject = "Many models are selected" {
    font: font {
        italic: true
    }
}
$ tree_def_label one metaobject @CommonMetaObject = name($*self)

$ graph one metamodel root = graph {
    animate: true
    rootnode(val models)
```

```
    action: "select" {grv{s{{{
        def sel = event.getSelection().collect{
            it.getContext().getObject().iterator().toList()
        }.flatten() as Set
        context.setVariable("selection", sel)
        "consume"
    }}}}}
}

// GENERIC PROPERTIES

$ name no metaobject @CommonMetaObject = "noname" {
    font: font {
        italic: true
    }
}
$ name one metaobject root = concat { "URN " id }
$ name one metaobject GRL = concat { "GRL " id }
$ name one metaobject UCM = concat { "UCM " id }
$ name one metaobject URNmodelElement = concat { id " " value_force($name) }

// GRAPH

$ rootnode one metamodel root = concat {
    "Root " id
    font: &modelFont
    border: &modelBorder
}

$ rootnode one metamodel GRL = <model> {
    <metamodel>: $GRL
    boxV {
        concat {
            name($*self)
            font: &modelFont
        }
        border: &modelBorder
        free {
            rootnode($*def)
        }
        action: "contextualMenu" {
            menucreate($*self)
        }
    }
}

$ rootnode one metamodel UCM = <model> {
    <metamodel>: $UCM
    boxV {
        concat {
            name($*self)
            font: &modelFont
        }
        border: &modelBorder
        free {
            rootnode($*def)
        }
        action: "contextualMenu" {
            menucreate($*self)
        }
        bitt: "rectangular" {
            for: $UCM@PluginBinding.range
        }
    }
}
```

```
$ isBindingSelected no metaobject @CommonMetaObject = false
$ isBindingSelected one metaobject @CommonMetaObject = false
$ isBindingSelected one metaobject UCM@PluginBinding = grv {
    updateOn: val selection
    s{{{
        def s = context.getVariable("selection")
        return s != null && s.contains(self.getHead())
    }}}
}
$ isBindingSelected one metaobject UCM@InBinding = $UCM@<InBinding.binding>.range
$ isBindingSelected one metaobject UCM@OutBinding = $UCM@<OutBinding.binding>.range
$ bindingColor one metaobject @CommonMetaObject = if {
    guard { condition: isBindingSelected($*self)
        "red"
    }
    guard { "orange" }
}
$ node one metarole UCM@PluginBinding = edge {
    targetShape: "triangle"
    color: bindingColor($*self)
    value($id)
    space {
        bitt: "center" {
            for: [ $UCM@<InBinding.binding>.range $UCM@<OutBinding.binding>.range ]
        }
    }
}

$ node one metarole UCM@InBinding = edge {
    color: bindingColor($*self)
    stroke: "dashed"
    "in" {
        bitt: "rectangular" {
            for: $UCM@<InBinding.binding>.domain
        }
    }
}
$ node one metarole UCM@OutBinding = edge {
    color: bindingColor($*self)
    stroke: "dashed"
    "out" {
        bitt: "rectangular" {
            for: $UCM@<OutBinding.binding>.domain
        }
    }
}
$ node one metarole UCM@<InBinding.binding> = edge {
    color: "yellow"
    stroke: "dotted"
}
$ node one metarole UCM@<OutBinding.binding> = edge {
    color: "yellow"
    stroke: "dotted"
}

// PROPERTIES

$ properties many metaobject @CommonMetaObject = "Select one object" {
    font: font {
        italic: true
    }
}

// PROPERTIES (IN MODEL)

$ metaself one metaobject @MetaObject = $*self
```

```
$ properties_urn one metaobject @CommonMetaObject = <model> {
    <metamodel>: metaself(meta{$*def.domain})
    <model>: $*def.domain
    properties($*self)
}
$ properties_urn one metaobject GRL@GRLmodelElement = <model> {
    <metamodel>: $GRL
    <model>: $*def.domain
    properties($*self)
}
$ properties_urn one metaobject UCM@UCMmodelElement = <model> {
    <metamodel>: $UCM
    <model>: $*def.domain
    properties($*self)
}
$ properties_urn many metaobject @CommonMetaObject = properties($*self)
$ properties_urn many metaobject GRL@GRLmodelElement = <model> {
    <metamodel>: $GRL
    <model>: $*def.domain
    properties($*self)
}
$ properties_urn many metaobject UCM@UCMmodelElement = <model> {
    <metamodel>: $UCM
    <model>: $*def.domain
    properties($*self)
}
```