Tallinn University of Technology
Institute of Cybernetics

A Framework for Design and Implementation of Visual Languages

Master thesis

Ando Saabas

Supervisor:
Professor Enn Tõugu

A thesis submitted in conformity with the requirements for the degree of Master of Science.

Tallinn 2004

Abstract

This thesis proposes a methodology for designing and implementing visual languages. It defines the abstract syntax of the visual languages that can be modeled in the framework and suggests to give the semantics of the language through a set of Java classes which are annotated with specifications in order to enable program synthesis. The specification language together with its semantics has been given. A markup language has been described which allows defining a concrete syntax for the developed visual languages. The fully functional prototypical implementation of the framework has also been presented, together with some conducted experiments.

Keywords: visual languages, program synthesis, specification languages.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The aim of this thesis is to develop a framework for designing and implementing visual languages. A strong emphasis is on providing the means for giving the syntax and semantics of the developed language. One of the main goals is to enable defining concrete and formal semantics for the designed languages, so that schemes – sentences in a visual language – could be compiled into executable code.

Such a framework could be used for rapid development of domain specific visual languages and CAD/CASE tools.

In this introductory chapter, we give a background and motivation for our work, and briefly describe our proposed solution.

1.1 Background

Visual languages have entered the mainstream in computing, as visual programming and visual user interfaces are becoming increasingly popular and common in several domains. Visual languages have become the most common way of doing GUI programming [9], there have been a plethora of visual languages developed as problem-oriented tools for specific domains, for example languages for solving certain engineering problems, query languages for spatial databases etc. One of the most important usage of visual languages has been for visual specifications and graphical presentation of data. There has been a big success for visual specification languages such as UML and SDL, both of which have gained wide acceptance and popularity and are now considered as industry standards (see also chapter 6 for an overview). There has not, however, been a major breakthrough for an all-purpose visual language.

Theoretical research about visual languages has mostly been about parsing visual languages, i.e. the development of a grammatical theory of visual language syntax and semantics. Lately the focus has also shifted to other areas of visual computing and the theory of visual languages is not modeled on standard formal language theory only [31].

A lot of effort in the development in visual language theory has gone into the research of the dynamic and interactive aspects of visual languages, since most visual language applications are inherently dynamic and interactive. So it is often preferred that the semantic effects of program edits (and runs) should be automatically displayed, as immediate visual feedback (sometimes called WYSIWYG, or *what you see is what you get*) is one of the most important features of programming in multiple dimensions [8].

There has also been an interdisciplinary and cognitive approach to the theory of visual languages [31], namely:

- Research on what makes one visual language formalism better than the other and when is visual language more appropriate than purely textual notation.

- Research on what makes one diagram better than the other for communicating one particular message.

These areas require empirical research and the developing of cognitive models of how humans interact with visual notations. This research is enhancing the understanding of existing visual notations and provides a basis for the design of new visual notations and strategies for data visualization.

A big trend in visual language research is towards application specific languages, for example query languages for spatial databases, diagrammatic notations for engineering, languages for specifying visualization graphics depicting scientific data etc (one of the most popular tools for industry level measurement analysis and data presentation is a graphical development environment LabVIEW [21]). Another field of research, intertwined with the above, is the problems of ascribing formal semantics to visual notations [31]. One of the main complication in this area is that domain-specific notation and knowledge should usually be interpreted appropriately - but as every domain has its own notation, which in turn can often be ambiguous, it can lead to several problems.

1.2 Motivation

It has been recognized for many years that there is a vital difference between an application's domain and its code [22]. They are different worlds each having its own language, experts, ways of thinking etc. The software developer has to build a bridge between these worlds, solving problems in both worlds [36].

Typically the problem has to first be solved in the domain world, but usually with little or no tool support. Modeling tools such as UML do not help much in this area, since they do not relate directly to the application domain but to the implementation – UML is mainly meant for visualizing code. Furthermore, usually a very small percentage of the finished code can be automatically generated from UML diagrams. So the developer has to solve the problem at least twice – first in domain terms (usually either in his head or on paper), and then in code terms.

One of the solutions to this problem is domain specific languages [52]. With domain specific languages, it is often possible to solve the problem (almost) only in domain terms. Excellent examples of domain specific languages are TEX, Yacc, SQL, Make, HTML, LabView – through bringing the (often declarative or visual) programming language close to the problem domain, they significantly reduce the time and effort needed for developing the solution.

Unfortunately, not every domain has such well-established tools, so the developer still often has to write their custom code in a general purpose language. As mentioned, the ideal would be that the solution could be developed only once, as a model of the particular domain, from which the full product could be automatically generated. The problem is, for this goal to be achieved, one would need a modeling language of the particular domain, a tool for building models and then a methodology and tools for generated the final product from the model (code, computational results etc). Building all these tools from scratch is a monumental effort (but one which still often has to be undertaken). However, using an effective framework could significantly simplify the task of building such tools. An expert of the particular domain would build the modeling tools in the framework and the solutions could be developed using those tools, incrementally improving them as needed.

There have been several attempts to provide such frameworks, all with their own particular emphasis (for an overview, see chapter 6), still it seems to be not a completely mature field.

1.3 Problem statement and proposed solution

The objective of this work is to design and implement a framework for creating visual specification-languages, where the visual specification (a scheme) could be directly translated into executable code. This research is largely a continuation of the work that has been done on the NUT system [50].

In the suggested framework, a new visual language is designed (compositionally) on top of an implemented ontology of a problem-domain. The semantics of the ontology is given by a set of classes and methods which describe the objects of the domain, and their relations. This is done through adding specifications to the classes in a meta-language as to what their actual computational meaning is.

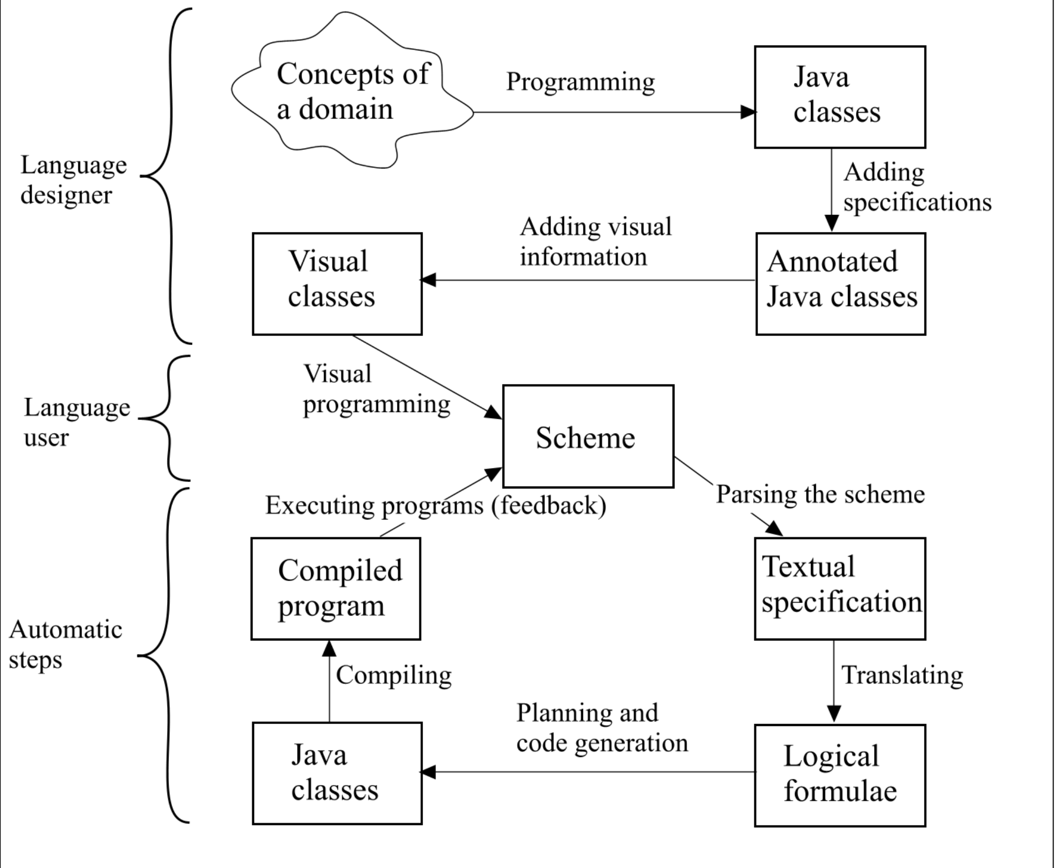


Figure 1.1: A visual language life cycle

We propose to implement the ontology by first writing the necessary classes and methods in the Java programming language, and then annotating them with specifications, i.e. pre- and postconditions stating what these classes or methods can compute, and on which conditions. We have chosen the Java programming language as the imperative part of the ontology and as the target language for the final code generation. There are several reasons for doing so. Java has become one of the most widely used programming languages, being quite simple to learn, robust, architecture neutral and most importantly, a strictly object-oriented language.

The scheme in figure 1.1 describes the general approach we take for developing a visual language and using it for programming.

The first three steps described in the scheme are taken by the developer of the language who is responsible for writing the necessary Java classes, annotating them and adding visual extensions. Note however that these three steps do not have to be in that chronological order. In practice, there can be a large degree of parallelism and iteration between these steps.

When the classes capturing the problem domain have been built by the developer, the language user can then use them for visual programming, arranging and connecting objects to create a scheme. Manipulating the scheme – a visual representation of a problem, is the central part of the user's activities.

When the scheme has been built by the user, parsing, planning and compiling are automatic steps taken by the computer. The compiled program then provides an implementation for the problem specified in the scheme and the results it provides can be fed back into the scheme, thus providing interactive properties.

1.4 Organization of the thesis

The rest of the thesis can roughly be divided into 3 main parts. The first part (chapter 2) concentrates on the visual language – we describe the abstract syntax and semantics of the visual language, discuss its expressiveness and implementability and propose a markup language for defining the concrete syntax of a developed language. In the second part (chapter 3), we concentrate on the specification language for Java classes, give the semantics of the language and discuss the Java code generation. The relationship between these two parts will be described as a translation from the visual specification into the textual specification. The third part (chapters 4 and 5) describes the implementation of the framework and the experiments that were conducted.

The related work will be discussed in chapter 6 and the conclusions will be drawn in chapter 7.

Chapter 2

The visual language

2.1 Defining the visual language

Defining the syntax and semantics of a visual language is a somewhat more complicated task than doing the same for textual languages, where the tools and methods have been long established. The latter can not be said about visual languages, up to this day there is no generally accepted framework for defining the syntax and semantics of visual languages.

There are various techniques that have been proposed for defining visual languages, which differ significantly in the way they conceptualize a visual notation, as well as in the way they describe its syntactic structure [23]. However most of them seem to have a rather marginal impact on practice [32]. Below, we will give an overview of some of the more influential techniques that have been proposed.

Janneck and Esser suggest in [23], to take a predicate-based approach by defining an abstract syntax of a picture (a visual sentence) to be an attributed graph and then to restrict the admissible graph structures and/or attributes by defining a set of predicates over the graph. Their approach is based on the abstract syntax and semantics framework suggested by Erwig in [14].

Atomic relational grammars, described by Wittenburg in [53] extend traditional grammars into the domain of multidimensional languages. A sentence of a multidimensional language is a finite graph of tokens connected by edges representing arbitrary relations. This is in contrast with a textual language, where a sentence is a finite list of symbols. Atomic relational grammars employ the normal grammatical mechanisms for specifying a language: sets of terminal and non-terminal symbols, a start symbol and a set of productions. In addition, an ARG defines a set of relations and a set of syntactic attributes for each non-terminal; they appear in the constraint expressions of the productions of a grammar; the constraints define the topology of the subgraphs of symbols derived by the productions.

Salter in [39] outlines a framework for syntax definition designed to enable the construction of grammatical formalisms which are suitable for defining the syntax of visual languages. It is however very general, and does not offer any immediately applicable techniques.

Picture layout grammars, introduced by Golin and Reiss in [18] can be used to specify syntax of visual languages in much the same way that context-free grammars are used to define textual programming languages. It is based on attributed multiset grammars where the right hand side of productions are considered to be an unordered collection of symbols. The language created by such a grammar is a set of multisets. For a visual program, this abstract structure corresponds to the decomposition of a picture (scheme) into subshapes and the productions correspond to picture composition operators. This technique seems to be most suitable for specifying geometry-based visual languages [11].

Besides technical issues, one of the main concerns when defining a visual language is finding a balance between the expressiveness and implementability of the language. Languages that are too expressive tend to be very difficult to formalize and therefore to implement [49]. But by restricting the expressiveness of the language, we also restrict its usability.

Considering the programming language paradigms – visual imperative languages tend to be too difficult to use for programming - control structures, algorithms etc are much more easily representable in textual form. As explained in figure 1.1 we take the approach to mix the two in software development, a textual programming language is used to describe the components, and visual language is for expressing the structure and relations, i.e. the architecture.

2.1.1 Syntax

We do not define the concrete syntax for specifying visual languages here but rather take the approach suggested by Erwig in [14] – we give an abstract syntax of the language. The framework presented here is meant for fast prototyping of various problem-oriented visual languages, so the concrete syntax(the look of objects, types of connections etc) can be arbitrary. This means that we have to abstract from the choice of icons or symbols and from geometric details like size and position of objects – the so called graphical attributes. Also, we do not give the abstract syntax of our visual language by a grammar. This would raise the need to take into consideration some context information which would unnecessarily complicate the definitions of transformations [14]. Furthermore, as has been demonstrated in [13], one can have a (de)compositional/recursive view of graphs without resorting to grammars.

On an abstract level, we consider a scheme (an expression in the visual language) to be a *simple graph* with some additional information about its nodes.

Definition 2.1.1. *A scheme G is a tuple (O, P, E, g) where O is a set of objects, P is a set of ports, E is a set of unordered pairs of elements of P and the total mapping $g : P \to O$ defines the object to which a particular port belongs.*

Definition 2.1.2. *An object O is a triple (N, C, A) where N is the name of the object, C is the type of the object, and A is a set of object attributes.*

Definition 2.1.3. *An object attribute A is a pair (N_a, V) where N_a is the name of the attribute and V is the value of the attribute.*

It is worth noting that although syntactically we do not allow loops (ports connected to themselves), or multiple connections between the same two ports, we do so only to keep the syntax somewhat cleaner, semantically it would make no difference if we allowed the latter connection types. The reason behind this will be explained later.

By restricting the language to the set of simple graphs it might seem we loose much in the sense of expressiveness of the language. However, in the following sections we will show that the expressiveness will not be lost, at the same time we can keep the language clean and easily implementable.

2.1.2 Semantics

When we consider a particular problem-domain, a scheme representing something in this domain typically has what one could call the "real" semantics, the meaning of the scheme that a specialist of that domain recognizes and understands. Our obvious aim is to represent this knowledge (at least to some extent) in a computer. For this, we have to reason about the semantics of schemes on several different levels.

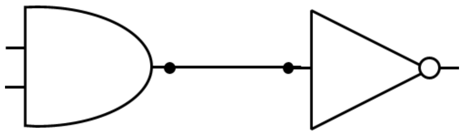
Firstly, one can consider the *shallow semantics* of a scheme (see also [50]) – the textual representation of the graph underlying the scheme. On this level, all deeper information about the scheme is disregarded. To be more precise – it is hidden in the types(classes) of objects. The textual representation of the graph could essentially be in an arbitrary form, for example XML. The shallow semantics however is only useful as a medium into a language where we could further reason about the "real" meaning of the scheme. We consider the shallow semantics of a scheme as a function that translates a scheme into the specification language described in chapter 3.

Definition 2.1.4. *The shallow semantics of schemes is a function $\mathcal{SS}$ which, for each scheme G, returns a string including exactly the following:*

- *Class obj*; *whenever G includes an object named obj of type Class.*

- *obj.x* $:=$ *v*; *whenever an attribute x of object obj has the value v in scheme G.*
- *obj1.x* $=$ *obj2.y; whenever there is an edge between ports x and y, and these ports are associated with objects obj1 and obj2, respectively.*

Example. Below, we have a simple scheme – two logic gates connected in a logical circuit.



The shallow semantics would translate the scheme into the following specification:

```
And and;
Not not;
not.in = and.out;
```

Such a specification clearly describes a graph with two vertices and an edge between when. However, the specification obtains a deeper meaning when we consider the types *Not* and *And* of objects *not* and *and*. According to the canonical meaning of such types, the above scheme actually specifies a function which takes two arguments (the two inputs of the AND gate) and returns an answer, realizing an NAND gate.

This is what we call the *deep semantics* of a scheme, namely a program or a set of programs that can be synthesized from the scheme, solving the problem(s) specified in the scheme.

Definition 2.1.5. *The deep semantics of schemes $\mathcal{DS}$ is a composition of the shallow semantic function $\mathcal{SS}$ and of the function $\mathcal{S}$ which defines the logical semantics of textual specifications ($\mathcal{DS} = \mathcal{S} \circ \mathcal{SS}$)*

The semantic function $\mathcal{S}$ is defined in chapter 3.5.

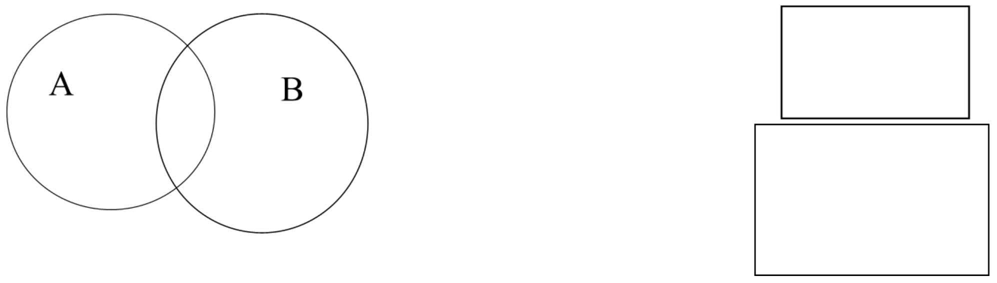


Figure 2.1: Spatial relations – *intersect* and *above*

2.2 Expressiveness of the language

The classification of visual languages given in [11] divides them into 2 main subsets - connection-based and geometry-based, reflecting the two basic modalities used to compose visual sentences, i.e. by connecting or spatially arranging graphical objects. Our visual language framework obviously belongs to the first subset. So it may seem that by just allowing connecting ports of objects, we limit the set of visual languages that can be modeled in the framework. For example, how could one express spatial relationships like intersection or inclusion? The approach we take is to look at geometry-based visual expressions simply as graphs (figures 2.1 and 2.2). This way, one could consider geometry-based languages as a subset of graph-based languages and model relations like intersect, inside, above by connecting two objects through ports of those types. Of course this does not mean that there has to be a *visual* edge between the two ports. In the framework, an edge can be implicit, for example it is created when an object is put above another object.

Another question might arise from the fact that a scheme is an undirected graph, so it might be difficult to model directed relations, for example generalization, aggregation etc. This, in fact, can be dealt with in a very straightforward manner. Firstly, it could be done in the same way as we model the spatial relations, namely connecting two opposing ports, like a parent and child port of two objects. But a much more powerful and elegant solution to

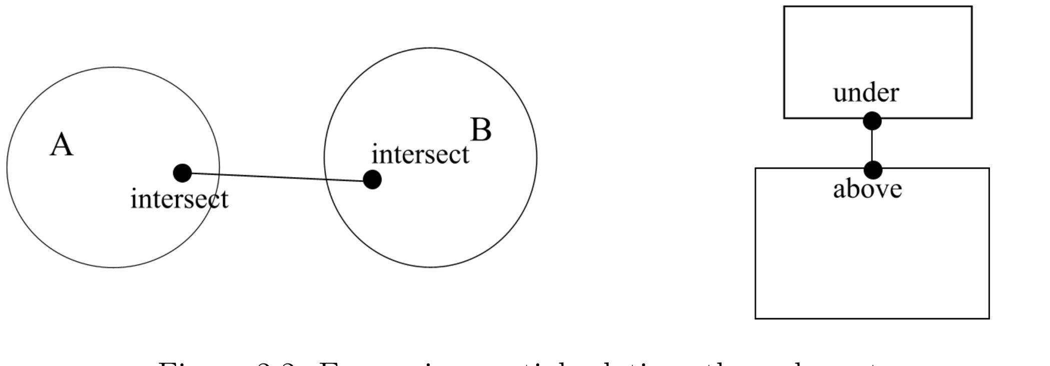


Figure 2.2: Expressing spatial relations through ports

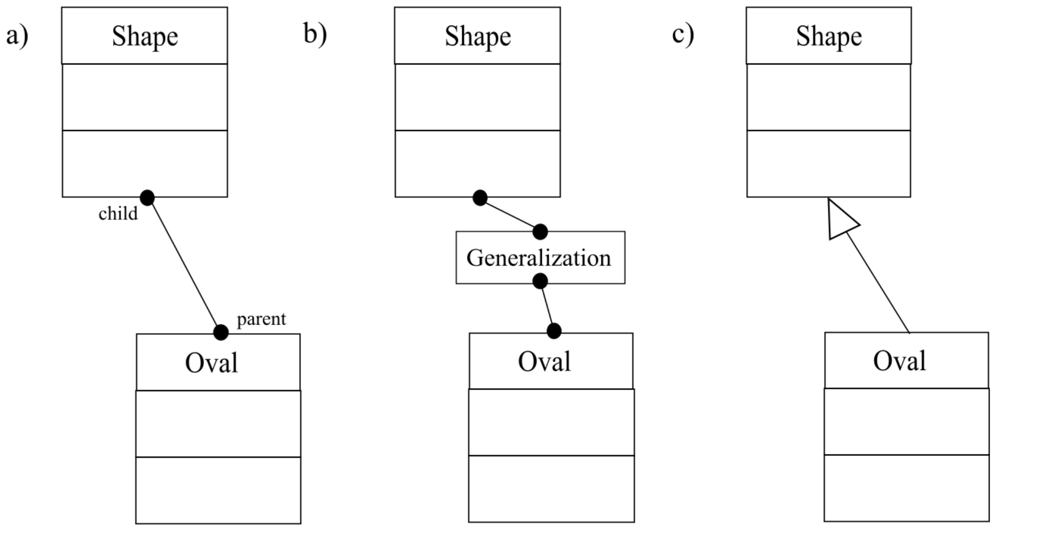


Figure 2.3: Modeling directed relations

this would be defining a class which represents the particular relations; this way, an ”edge” between ports would not only denote their equivalence, but instead could have a much deeper meaning, captured in the class specifying the relation. In examples b) and c) of figure 2.3 the generalization relations are defined as classes.

So we have deliberately kept from extending the syntax with spatial or directed relationships and have tried to keep the language as simple as possible, at the same time, it can be shown that the expressiveness of the language is not lost.

2.2.1 Polymorphism

In the NUT system [50], a port can be defined to have type *any*, meaning that the port is actually typeless until it is connected to another port. After being connected, it obtains its type from the port it is connected to. This provides a kind of polymorphism, where one does not need to know concrete types of all ports during the time of specifying the components.

Here we extend this notion of polymorphism, allowing a port to be specified both nameless and typeless. Such a port (named *any*) would obtain a name and a type only after it is connected to another port.

This can be especially useful when a class has many different types of (directed) relations. Contrary to the NUT system, where a relation would always be defined by the ports between which the relation is drawn, we can have an opposite approach – the relation defines the ports, as is the case with typical directed diagrams such as data- and control-flow diagrams, class diagrams etc. Of course, this only applies when we use "typed" relations (i.e. classes acting as a relations). Typically we would need to define a separate port for each type of relation. Using name polymorphism just a single port can be defined, and one can let the typed relation and its direction determine the connection type.

In the example presented in figure 2.4, we have three classes A, B and C, where C is a class defined as a relation and the classes are specified with the following ports:

```
A: Any any;

B: Any any;

C: int x; String y;
```

The specification of the scheme b) is

```
A.x = C.x;
B.y = C.y;
```

and the specification of scheme c) is

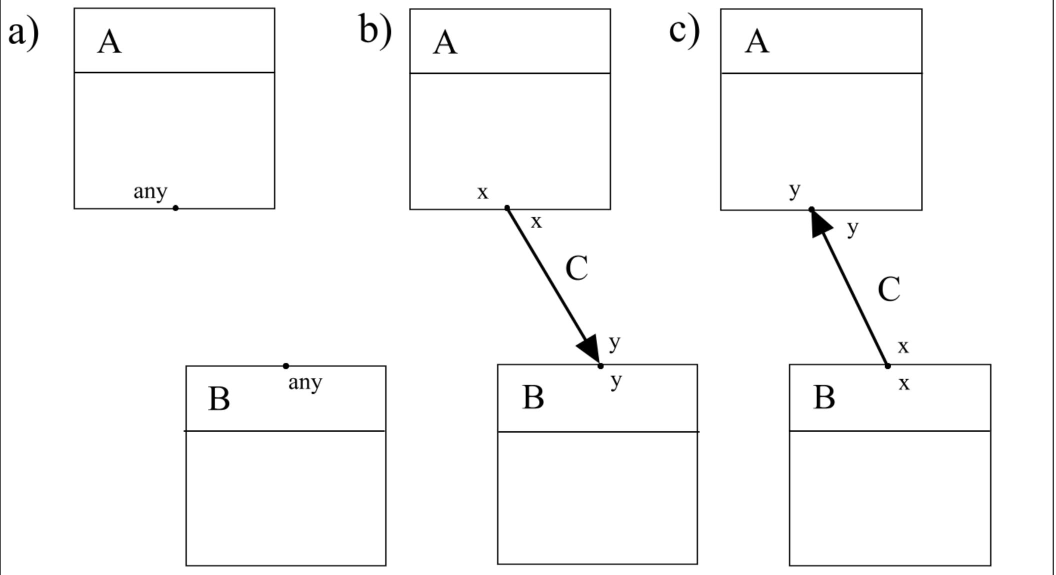


Figure 2.4: Examples of port polymorphism

```
A.y = C.y;
B.x = C.x;
```

In the first case, the *any* port of class *A* was connected to port named x : *int*, so the port *any* is "cast" into x : *int*. A similar thing happens in the second case, where the port is cast into y : *String*. Even if we added new kinds of relations, we do not have to define additional ports. If an *any* port is connected to more than one port, it obtains a specific name for each connection instance.

Without name polymorphism, the above would have required 2 ports, and 2 more ports for each additional directed relation type. Using name polymorphism, we can always use a single port for cases like that.

2.2.2 Hierarchical composition

A scheme consists of a set of connected objects. Each object can consist of other objects, organized as a scheme. This way, a scheme can include subschemes, which in turn can include subschemes etc. Such hierarchical composition, very typical in modeling and software development ([4],[2]), allows the scheme to be viewed on several different levels of abstraction,

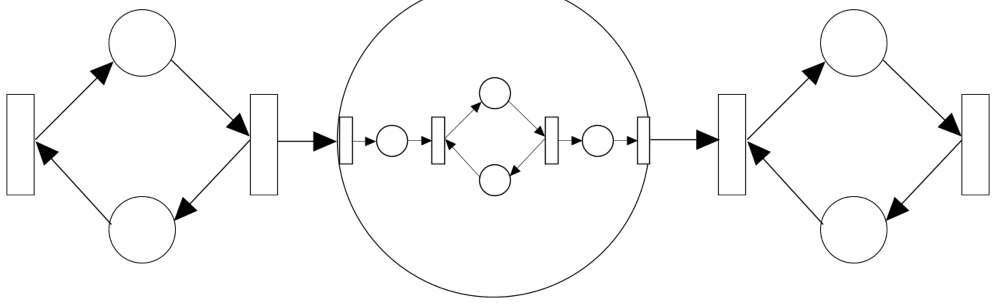


Figure 2.5: Example of hierarchical composition – a hierarchical Petri net

providing the separation of concerns to identify, encapsulate, and manipulate the parts of scheme that are relevant to a particular issue.

2.3 Defining the visual classes

2.3.1 Package format

As was discussed in section 1.3, building visual classes in the described framework includes

- writing Java classes
- annotating the classes with specifications so that they can be used in program synthesis
- adding visual information to the classes - how an instance of a particular class looks and interacts with other instances.

While the annotations are included in Java classes, the visual extension are written in separate files. In a way, this visual specification, or a shallow visual class as we call it, is a separate entity, which could also be used independently from the Java class. One can say that the visual class further *specifies* the Java class, while the Java class *realizes* the concept described by the visual class.

Information added to annotated Java classes to make a visual class (figure 2.6) is the following:

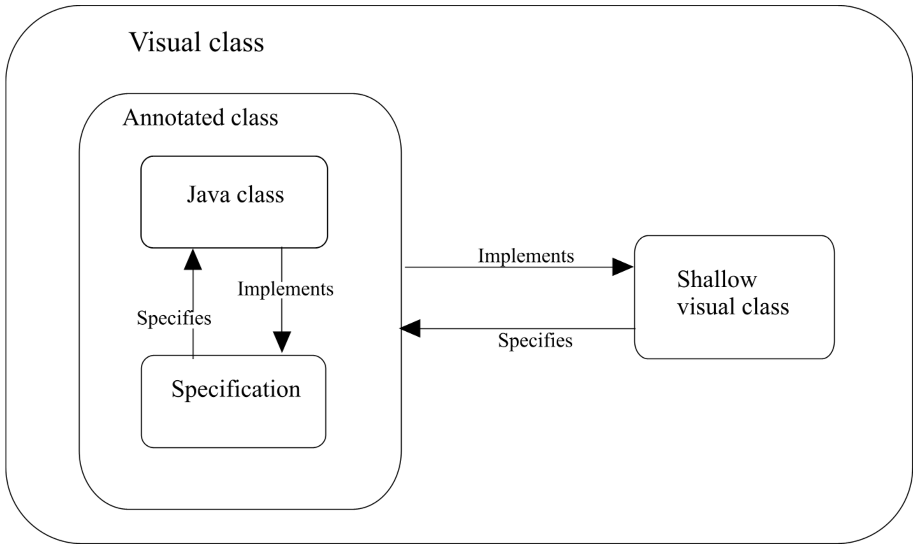


Figure 2.6: The visual class

- *Look* of the class, its shape, size and other graphical attributes
- *Ports* – how the objects can be connected to other objects
- *Fields* – the attributes which can be viewed and changed in the visual editor
- *Interactive properties* – how its look, position etc is affected by the changes in object attributes and vice versa.

We use XML [38] as the language to define the visual properties of the classes. The choice of using XML was rather imminent – XML has become a widely used standard, being easy to both read and write and carrying formal syntax. At the same time, there are good XML parsers available, making loading and writing information easy in the developed application.

Note that in the following when specifying the package format, we use XML Schema (instead of BNF) to specify the structure of the package. XML Schemas provide a means for defining the structure, content and semantics of XML documents and was approved as a W3C Recommendation in 2001

[41]. Since it was designed specifically to define XML structure, using this format is in this case more natural than using BNF.

Package

A package forms a full visual description of the problem domain in which the problems are to be solved. It consists of visual classes that form certain concepts in a particular problem domain. For example the classes(concepts) Capacitor, Resistor, Inductor and Power Supply when grouped together form a package – an ontology for simulating electrical circuits.

A package definition includes its name and description together with the list of classes. It has the following form.

```
<xs:element name="package">
  <xs:complexType>
   <xs:element name="description" type="xs:string">
   <xs:element ref="class" maxOccurs="unbounded"/>
   <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

Shallow visual classes

The purpose of a shallow visual class is to specify how objects of that class look and interact in the visual editor.

The name of a visual class should be the same as the corresponding Java class name. The class attributes such as fields and ports should also typically be bound to the respective attributes in the Java class specification.

The definition of a class includes its name, a brief informal description of the class's purpose (which can be shown as a tooltip in the visual editor). The icon attribute in the definition specifies the file name of the icon to be shown in the palette.

The definition also includes the graphics, port and field definitions, i.e. how an object looks like, how it can be connected to other objects and which attributes the class has – they can be shown and changed in the scheme.

The format is the following:

```
<xs:element name="class">
 <xs:complexType>
  <xs:element minOccurs="0" maxOccurs="1" ref="description" />
  <xs:element minOccurs="0" maxOccurs="1" ref="icon" />
  <xs:element ref="graphics" use="required"/>
  <xs:element ref="ports" use="required" />
  <xs:element ref="fields" minOccurs="0"/>
  <xs:attribute name="name" use="required"/>
  <xs:attribute name="type" use="optional">
   <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:enumeration value="relation" />
     <xs:enumeration value="class" />
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
 </xs:complexType>
</xs:element>
```

The *type* attribute of a class can be be used to define the class as a relation, thus forcing it to obtain specific properties as described in section 2.2. When a class is defined as a relation, it must have at least 2 ports. The first declared port becomes the starting point and second port the end point for drawing the relation. The relation can be added to the scheme by clicking on a port of another object; the first port of the relation object is then connected to that port, and the object itself becomes freely rotatable around the point defined by the port. As explained in section 2.2, this will give us the means to design new explicit relationships as classes.

Graphics of class

This part of the visual class specification describes how the objects of that class will look in the scheme editor window. Besides typical shapes and attributes such as size and colour, developers are allowed to make use of more advanced graphical features like anti-aliasing, transparency etc.

```
<xs:element name="graphics">
 <xs:complexType>
  <xs:element ref="bounds" />
```

```
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element ref="line" />
    <xs:element ref="text" />
    <xs:element ref="dot" />
    <xs:element ref="rect" />
    <xs:element ref="oval" />
    <xs:element ref="arc" />
   </xs:choice>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

All elements through which the graphics is defined (rectangle, oval etc) can have their own color, stroke, and transparency. Stroke defines a basic set of rendering attributes for the outlines of graphics primitives, in our case, the width of the line with which the object is drawn . Transparancy can vary from non-transparent to completely see-through(values 0-100).

A *graphic* definition can consist of the following attributes:

- *Bounds*, which set the size of the object. An object is always considered to be the size of its bounds, independent of the visual look of the object. So a mouse-click in the scheme window will only be registered as a click on the object, if it is inside the bounds of this object. The bounds are always a rectangle, but they may be tilted in the scheme editor window.

The actual look of the object is defined by the following fields:

- *Line*, defined by a beginning and an end point. The optional attributes are colour, transparency and stroke. The default values for these attributes are black, 0 and 1 respectively. Similar attributes and default values apply to other graphical elements.

- *Oval*, defined by its surrounding rectangle. An oval, just like arc and rectangle may be defined as filled.

- *Dot*, which paints one pixel to the desired colour.

- *Text*, which draws a string in specified style, font and size.

- *Arc*, which draws an outline of an elliptical arc defined by its surrounding rectangle.
- *Rect*, which draws a rectangle specified by the point of its upper right corner, width and height.

The following schema describes the structure of an element `rect`. The other graphical attributes have similar definitions.

```
<xs:element name="rect">
 <xs:complexType>
  <xs:attribute name="x" type="xs:integer" use="required" />
  <xs:attribute name="y" type="xs:integer" use="required" />
  <xs:attribute name="width" type="xs:integer" use="required" />
  <xs:attribute name="height" type="xs:integer" use="required" />
  <xs:attribute name="colour" type="xs:integer"/>
  <xs:attribute name="transparency" type="xs:integer">
   <xs:simpleType>
    <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="100"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="filled">
   <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:enumeration value="true" />
     <xs:enumeration value="false" />
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
 </xs:complexType>
</xs:element>
```

Ports

Ports define the interface through which objects (and their attributes) can be connected to each other, describing most of the behavioral properties of the language. The structure of port definitions is the following.

```
 <xs:element name="ports">
  <xs:complexType>
```

```
   <xs:element maxOccurs="unbounded" ref="port" />
  </xs:complexType>
 </xs:element>

 <xs:element name="port">
  <xs:complexType>
   <xs:element name="open" minOccurs="0" maxOccurs="1" />
    <xs:complexType>
      <xs:element ref="graphics" />
    </xs:complexType>
   </xs:element>
   <xs:element name="closed" minOccurs="0" maxOccurs="1" />
    <xs:complexType>
     <xs:element ref="graphics" />
    </xs:complexType>
   </xs:element>
   <xs:attribute name="name" type="xs:string" use="required" />
   <xs:attribute name="type" type="xs:string" use="required" />
   <xs:attribute name="x" type="xs:integer" use="required" />
   <xs:attribute name="y" type="xs:integer " use="required" />
   <xs:attribute name="connection">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="single" />
      <xs:enumeration value="area" />
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
   <xs:attribute name="binding">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="strict" />
      <xs:enumeration value="line" />
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
  </xs:complexType>
 </xs:element>
```

Every port is defined by its name and its position on the object. For every port x defined in the class, there should typically be a variable x defined in the specification of the corresponding Java class, or the port name must be *this* (meaning the object itself can be connected to other ports) or *any* (see section 2.2.1). The type of the port also has to be set – only ports with matching types can be connected to each other (unless one of the ports has type *any*).

To give developers greater flexibility when defining the language syntax, ports can have their own predefined graphics, both for an open and closed (i.e. connected to another port) port. So a port can a have a different look when it is connected to another port.

When a port's graphics is undefined, its default look is a circle with a radius of 5 pixels, filled when its closed.

Typically, a port is a connection area, meaning that a port can be connected to any number of other ports. Optionally, a port may be set as single so that it can accept only one connection. This may be useful when we explicitly want to avoid certain types of ambiguity in the visual language being defined.

A port being strict means that it can be connected to other strict ports by placing one port directly over the other port. This is a useful feature by which we gain the graphical attributes for arranging objects spatially, so it is to some extent possible to create geometry-based visual languages in our framework.

Fields

Fields attribute lists the fields that can be set/shown in the visual editor. A field is defined by its name, type and default value. For every field x, there should be a variable x defined in the class's meta-interface. The type and default value of a field can be set in the package.

Just like ports, fields can include a *graphics* element. This gives us the means to provide simple visual feedback in the editor – to define the graphical attributes to be shown when a particular field is known, and when its not.

The syntax of the field declaration is as follows

```
<xs:element name="fields">
 <xs:complexType>
```

```
   <xs:sequence>
    <xs:element maxOccurs="unbounded" ref="field" />
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="field">
  <xs:complexType>
   <xs:element name="default" minOccurs="0" maxOccurs="1" />
    <xs:complexType>
     <xs:element ref="graphics" />
    </xs:complexType>
   </xs:element>
   <xs:element name="known" minOccurs="0" maxOccurs="1" />
    <xs:complexType>
     <xs:element ref="graphics" />
    </xs:complexType>
   </xs:element>
   <xs:attribute name="name" type="xs:string" use="required" />
   <xs:attribute name="type" type="xs:string" use="required" />
   <xs:attribute name="value" type="xs:string" use="required" />
  </xs:complexType>
 </xs:element>
```

If no fields are specified in the class definition, the variables that are shown/editable in the scheme editor are those declared in the specification that are of primitive types or string, or an array of the forementioned types.

Chapter 3

Synthesis of programs

So far, we have given the syntax and shallow semantics of the visual language, as well as defined the format for describing the entities in the language.

We would, however like to be able to give deeper semantics to the languages developed in the framework, so that schemes could be directly translated into executable code. We propose to give the functionality of visual classes via Java classes and methods that realize the concepts visualized by the shallow visual classes. While Java classes themselves can describe their functionality, it is unfortunately not enough to perform automated synthesis of full programs. For this, there has to be some glue language to support automated composition of software. In this chapter we will go through these issues, first describing the program synthesis methodology that we use and then define the specification language that can be used to extend Java source files to support this methodology.

3.1 SSP and ESSP

Below we will briefly describe structural synthesis of programs, or SSP, as put forward by Tyugu and Mints in [37]. For a thorough overview of SSP and related topics, see [26].

SSP is based on the idea that programs can be constructed taking into account only their structural properties. This means that only the structure of a program is synthesized, and the full program is built from preprogrammed components, whose precise behaviour would be difficult to describe in logic. A preprogrammed module is supplied with an axiom stating under which condition it can be applied, and what values it computes. The axiom, however, does not describe the relationship between input and output values. So instead of a general form of a specification of a module $\forall x(P(x) \rightarrow \exists y R(x, y))$, one can use only $\forall x(P(x) \rightarrow \exists y R(y))$, which in turn can be reduced to $(\exists x P(x)) \rightarrow (\exists y R(y))$. As a consequence, the axioms can be represented in a propositional language, considering $\exists y R(y)$ as a propositional variable R [34].

3.1.1 The logical language

The logical formulae in the language are the following.

1. Propositional variables denoted by capital letters $A, B, C, ...$ etc.

2. Unconditional computability statements $A_1 \& ... \& A_n \rightarrow B$

3. Conditional computability statements $(\underline{A}_1 \rightarrow B_1) \& ... \& (\underline{A}_n \rightarrow B_n) \rightarrow (\underline{C} \rightarrow D)$, or alternatively $(\underline{A}_1 \rightarrow B_1) \& ... \& (\underline{A}_n \rightarrow B_n) \& \underline{C} \rightarrow D$. Note that we use $\underline{\mathrm{A}}$ to denote a conjunction $A_1 \& ... \& A_n$.

The computational meaning of the formulae is the following:

1. Propositional variables correspond to object variables in the underlying problem domain.

2. Unconditional computability statements denote that a value of an object variable b be can be computed knowing the object variables $a_1, .., a_n$.

3. Conditional computability statements express the computability of variable d from variables c, depending on other computations.

SSP uses intuitionistic logic, which guarantees the constructiveness of proofs. The logical language described contains only conjunctions and implications, so it is sufficient to use only introduction and elimination rules for

Table 3.1: Conventional inference rules using proof terms.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \wedge B} \wedge I \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash fst\ M : A} \wedge E$$

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash snd\ M : B} \wedge E$$

$$\frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash \lambda a.b : A \to B} \to I \qquad \frac{\Gamma \vdash f : A \to B \quad \Sigma \vdash a : A}{\Gamma, \Sigma \vdash fa : B} \to E$$

these connectives. Since each formula in the logical languages has a computational meaning, we can associate a proof term with each derivable formula (which corresponds to a program).

Program extraction occurs at each inference step. The realizations of axioms are given, and the realizations of assumptions $\underline{A}$ are object variables $\underline{a}$. Realization of a derived formula is built step by step from realizations of premises.

Note that in case of conventional inference rules (figure 3.1), program extraction is somewhat complicated due to steps needed for composition and decomposition of data structures that correspond to realization of conjunctions, which leads to very inefficient programs.

In SSP it is dealt with in a straightforward way, considering the conjunction of variables of arbitrary length as a single unit.

Table 3.2: SSP admissible rules using proof terms.

$$\frac{\Gamma, \underline{a : A} \vdash b : B}{\Gamma \vdash \lambda \underline{a}.b : \underline{A} \to B} \to I \qquad \frac{\Gamma \vdash f : \underline{A} \to B \quad \Sigma \vdash \underline{a} : A}{\Gamma, \Sigma \vdash f\underline{a} : B} \to E$$

$$\frac{\Delta \vdash f : \underline{(\underline{A} \to B)} \wedge \underline{C} \to D \quad \underline{\Gamma, \underline{a : A} \vdash b : B} \quad \underline{\Sigma \vdash c : C}}{\Delta, \Gamma, \Sigma \vdash f(\underline{\lambda \underline{a}.b}, \underline{c}) : D} \to EE$$

Table 3.3: Disjunction elimination in ESSP

$$\frac{\vdash M : \underline{W} \to A \vee B \quad \underline{\Gamma \vdash N : W} \quad \Sigma, u : A \vdash N_1 : C \quad \Delta, w : B \vdash N_2 : C}{\underline{\Gamma}, \Sigma, \Delta \vdash (case\ M\underline{N}\ of\ inl^B u \Rightarrow N_1 | inr^A w \Rightarrow N_2) : C} \vee E$$

The main idea of SSP is to use computability information present in the axioms to direct the search. Instead of using conventional rules of intuitionistic propositional calculus (figure 3.1), the so-called admissible rules are used (figure 3.2). Every application of an admissible elimination rule corresponds exactly to an application of a preprogrammed module represented by an axiom.

Surprisingly this implicative fragment of intuitionistic propositional calculus(IPC) is actually intuitonistically complete, meaning that it is possible to reduce an arbitrary intuitionistic formulae (including formulae with disjunction) to the logical language of SSP by introducing new variables [37]. This, however is relatively cumbersome. For example since modules can throw exceptions, the possibility to specify alternative outcomes is needed. In SSP, disjunctions have to be encoded in modules with subtasks. This can lead to large amounts of additional modules.

For the above reason, Lämmermann extended SSP in [26] introducing disjunction into the logic, making it possible to specify branching in a straightforward way. He allows to explicitly specify alternative outcomes by writing disjunctions on the right hand side of the axiom (figure 3.3).

3.1.2 Proof search algorithm

The proof search algorithm used in SSP was described in [48]. The proof search strategy combines an assumption driven forward search with goal driven backward search. Forward search is used to select computability statements. The goal-driven backwards search is used to select subtasks. If the forward search cannot continue, a subtask of a conditional computability statement, whose unconditional input variables have been computed, is selected. If the proof of the subtask succeeds, the next subtask of the same statement is selected. Otherwise, a subtask of another computability statement is selected. Subtasks are selected this way until either the goal is reached or no further subtask selection is possible. This proof search algorithm is extended, when disjunctions are allowed in the logical language. The extended proof search algorithm dynamically rewrites logical specifications containing disjunction connectives. A logical axiom with a disjunction is rewritten into the form of SSP's specification language if and only if during proof search the axiom has to be applied. The proof will then be continued with the new set of formulae. For each rewritten axiom, a realization deploying the module in the rewritten form has to be synthesized. This means that the modules for the rewritten specifications do not have to be supplied by the user, as was the case with SSP.

3.2 Extending Java classes with meta-interfaces.

In order to use Java classes for generating programs from schemes, we need to extend the classes with specifications (meta-interfaces). In our implementation, meta-interfaces are included in special comments in Java files, between `/*@` and `@*/`. The purpose of a meta-interface is either to specify how a given class can be computationally used in the package or to introduce a new component (class), which is composed of already specified components (we use it for hierarchical composition of visual classes). The specification language is in many ways similar to the one proposed in [26]. The goals of the two languages are however somewhat different, which requires some changes to the language syntax and its extensions.

The specification language proposed here has a straightforward translation into the logical language described in the previous section. The semantics of the language will be presented through describing this translation.

3.3 The language core

The core specification language allows declaring variables, constants, axioms and bindings. The meaning of these entities will be described below. The specification is defined as follows:

```
MetaInterface ::= 'specification' SpecificationName
                     '{' Specification '}'
Specification ::= (VariableDeclaration | ConstantDeclaration |
                   ControlVariableDeclaration | Axiom | Binding)';'
                     [Specification]
```

where `SpecificationName` is the name of the specification being declared and has to be the same as that of corresponding Java class.

3.3.1 Variable Declarations

To keep the language intuitive for the programmer, we try to keep the syntax of the specification language as Java-like as possible. Variable declarations follow the Java syntax.

```
VariableDeclaration ::= Type IdentifierList';'
IdentifierList ::= Identifier [',' IdentifierList]
```

`Identifier` is the name of the interface variable being declared, and `Type` can be either

- a primitive type of the Java programming language
- a class in the Java programming language or
- a meta-interface already described in the package.

Example.

```
specification Person {
  String name;
  int age;
}
```

3.3.2 Control variables

Axioms are used to specify when a method can be applied (what are its input parameters) and what it computes. Java methods can, however have void input and/or void output. An axiom specifying such an axiom would have an empty left and/or right side. This would lead to a problem, where we would be unable to guide the synthesis process, for example in the case where we want that a method with void input is applied after a method with void output is applied. To force applications of realizations when certain conditions are met (like in the above case), we introduce special variables (or control variables) into the specification language.

A control variable does not have a computational meaning, but it must be derivable when the axiom (see below) is used that has a control variable as one of the precondition. Control variables have to be declared via type *void*. The declaration of a control variable has the form

```
ControlVariableDeclaration ::= 'void' ControlVariableList';'
ControlVariableList ::= Identifier [',' ControlVariableList]
```

3.3.3 Constant declarations

Logically, a constant declaration is an axiom with an empty left-hand side, applicable without any preconditions. Constant declarations allow us to explicitly use values of the underlying programming language (in our case, Java). The specification of a constant has the form

```
ConstantDeclaration ::= Variable ':=' value';'
```

where `Identifier` has the form

```
Variable ::= Identifier | Identifier'.'Variable
```

and `value` refers to a constant in the Java programming language.

Example.

```
String[] s;
double pi;
s := {"text1", "text2"};
pi := Math.PI;
```

3.3.4 Binding

We use bindings to show that the realization of one component is the same as the realization of the other component. The syntax of bindings is as follows

```
Binding ::= Variable '=' Variable';'
```

So the specification `a.x = b.y` means that the realization of the variable `a.x` is the same as the realization of the variable `b.y`. This introduces compound names to the specification language and provides a possibility to handle hierarchical data structures.

Going back to the shallow semantics of our visual language, the connection of two ports means that there is a logical binding between components represented by these ports. This is also the reason why we do not allow loops and multiple connections between two ports in the visual language syntax. A loop on port x would have the semantics $x = x$, which is always trivially true, and multiple edges between ports x and y would mean that instead of a relation $x = y$, we would have a set of relations $x = y, x = y, ...$, which in our logical language are semantically equivalent.

3.3.5 Axioms

Axioms are logical formulae specifying the possibilities of computing provided the realizations of the components. In our language, an axiom is a declaration of computability, so axioms specify all the possible computations an object provides.

Axioms are written in the form of an implication. On the left hand of the implication the preconditions of the axioms are listed, while the right hand side of the implication lists the postconditions of the axiom. The realization of the axiom (a method of the class) is specified in the curly brackets. The axiom states that the realizations can be applied, if all of its preconditions are derivable.

Example. The following specification states that knowing a file name, we can read the file into a string via a method `readFile`. It also states that the length of a string `aString` can be computed in the specified class, and this computability statement is realized by the method `calcLength`. The specification together says that knowing a file, we can find its length.

```
String aString, fileName;
int length;
fileName -> aString{readFile};
aString -> length {calcLength};
```

The precondition of an axiom can be comprised of interface variables, control variables and subtasks or it can be empty, meaning that it can be applied without any condition.

The postcondition of an axiom is typically an interface variable corresponding to the value which the realization of the axiom (a Java method) returns. The postcondition can include one or more control variables - when an axiom is applied it means that these variables are derivable. The disjunction symbol '|' is used in the postcondition to specify possible alternative outcomes of a method (for example throwing exceptions).

```
Condition ::= (Variable | Subtask) [',' Condition]
Precondition ::= [Condition]
Postcondition ::= Variable [',' ControlVariableList ]
                   ['|' Postcondition]

VariableList ::= Variable [',' VariableList]
UnimplementedAxiom ::= [VariableList] '->' Variable '{spec}'

ImplementedAxiom ::= Precondition '->' Postcondition
                   '{'Realization'}'
Axiom ::= (ImplementedAxiom | UnimplementedAxiom)';'
```

where `Realization` denotes a preprogrammed module (a Java method) and `spec` is a keyword stating that the axiom is unimplemented – its body has to be synthesized.

Subtasks

Preconditions in the form of an implication denote a subgoal in an axiom. These functional parameters, which we call subtasks are like ordinary input parameters, but they also behave like goals (see next section). In a sense they are like unimplemented axioms so their realization has to be synthesized according to the specification, but they can be invoked only by the realization of the axiom where they occur as input parameters. Subtasks have the form

```
Subtask ::= '['VariableList  '->' Variable ']'
```

3.3.6 Goals

A goal specifies a program to be synthesized. A goal has the form

```
Goal ::= [Assumptions] '->' Goals
Assumptions ::= Variable [',' Assumptions]
Goals ::= Variable [',' Goals]
```

A goal states that assuming the identifiers given in `Assumptions`, synthesize a program which computes the variables corresponding to the identifiers given in `Goals`.

Example. Consider the specification in the previous section. For this, we could write a goal `fileName -> length`. This would force the synthesis of a program, which takes a string as an argument (the file name), and returns an integer (the length of the file).

3.4 Extensions to the core language

We extend the core language with several constructs - a "syntactic sugar" added to the core language, to make writing specifications easier for the developer. These extensions can be syntactically rewritten into the core-language, in fact that is what we do when parsing a specification. In many cases, using the extensions can make the specification an order of magnitude smaller and also easier to read, as well as reducing the amount of work needed to be done programming the Java classes.

3.4.1 Equations

Equations have the form

```
Equation ::== Expression '=' Expression';'
Expression  ::=  [ '-' ] Term [ ( '+' | '-' ) Term ]
Term  ::=  Factor [ ( '*' | '/' ) Factor ]
Factor  ::=  Primary [ '^' Primary ]
Primary  ::=  Number | Variable |'(' Expression')'  |
              Function-name '(' Expression ')'
Function-name ::= 'sin' | 'cos' | 'tan' |  'log'
```

We can use equations to bind variables with each other without the need to specify an axiom for each case. For example, instead of writing the specification

```
m, a -> F { calcForce}
F, a -> m {calcMass}
F,m -> a {calcAcceleration}
```

and adding the required methods for calculating the force, mass and acceleration of an object, we can use a single equation `F = m * a`.

3.4.2 Aliases

We allow the user to create lists of components via the `alias` construction. The syntax of aliases is the following

```
Alias ::= 'alias' Identifier '=' '(' VariableList');'
```

Example.

```
int x;
int y;
alias coordinates = (x, y);
```

We can then bind such data structures together, so that

`A.coordinates = B.coordinates` would unfold to

```
A.x = B.x
A.y = B.y .
```

Extensive use of this feature can be made when specifying visual classes – if more than one variable needs to be bound via a port, we create a data structure pointing to these variables.

The *alias* construction should generally be used for lightweight structures only, more complex structures should be presented as a class hierarchy. When all the components of an *alias* construction have the same type, it can be regarded as an array during the synthesis process, in this case, it can be bound to an an array, or be used as an identifier in an axiom.

3.4.3 Wildcards

Wildcards can be used in specifications when values have to be computed from, or propagated to all similar objects in the specification. We might, for example, need to synchronize the time in each object, or calculate the average on all the objects in the specification. It could be done by writing a separate axiom for each object. This will however quickly become cumbersome, since the specification for the above case should be updated each time an object is removed or added. Using wildcards, this could be prevented by writing the needed axiom(s) only once.

Identifiers that use wildcards have the following form:

```
Token ::= '*' | Identifier
Wildcard ::= Token['.'Wildcard]
WildcardVariable ::= Wildcard'.'Identifier
```

Such variables can be used as normal variables, but only in axioms (not in bindings or equations).

Example.

```
Object o1, o2, o3;
clock.time -> *.currentTime {sync};
```

The above specification is semantically equivalent to the following specification, stating that knowing the time, we can calculate the time in each object declared.

```
Object o1, o2, o3;
clock.time -> o1.currentTime {sync};
clock.time -> o2.currentTime {sync};
clock.time -> o3.currentTime {sync};
```

When wildcards are used on the left side of the implication, the input parameter of the method implementing the axiom must be an array of objects of this type.

Lets look at an example similar to the above, considering `currentTime` to be of type `int`.

```
int averageTime;
Object o1, o2, o3;
*.currentTime -> averageTime {average};
```

In the above case, the signature of the method has to be

`int average(int[] parameter)`.

This specification can be rewritten into

```
int averageTime;
Object o1, o2, o3;
alias x = (o1, o2, o3);
x -> averageTime {average};
```

Note that when using wildcards, only objects of the same type can be pointed to via a wildcard. Consider for example the following specification.

```
specification A {
  int x;
}

specification B {
  String x;
}

specification C {
  A a;
  B b;
  -> *.x {m}
}
```

Parsing this specification would return an error, since the x in class A and class B are of different types.

3.5 Semantics of the specification language

We now proceed to describing the logical semantics of the specification language, i.e. how the specification is translated into the logical language described in section 3.1. The translation is quite straightforward, with two major steps to be taken. Firstly, before applying the semantic function, the specification is rewritten into the core language via the translation of equations, axioms containing wildcards and aliases. Secondly, the hierarchical structure of the full specification is flattened; this is done as a part of the semantic function.

As a reference – a full definition of the logical semantics of the NUT language was given by Uustalu in [51]. Also, Lämmermann gives the logical semantics of the specification language he presents in [26].

3.5.1 Syntactic rewriting of extensions

Below we will use $SExp$ to denote the set of specification expressions, SP to denote the set of valid classes with meta-interfaces and I to denote the set of interface variables. We will use

$Ty : I \rightarrow SP$	as a function which returns the type of a given variable.
$sp_d : SP \rightarrow \mathcal{P}(I)$	as a function which returns the set of variables declared in a given class specification.

The function rewriting a given specification expression is the following.

Definition 3.5.1. *The function* $\mathcal{RW}$

$$\mathcal{RW} : SExp \rightarrow (SP \rightarrow \mathcal{P}(SExp))$$

rewrites a specification expression in a given context, returning a set of specification expressions.

The specification expression *sexp* in a given specification is replaced by the set of expressions generated by $\mathcal{RW}$.

Wildcards

Before we define the rewrite function for wildcard expressions, we first need a function to unroll the wildcard identifiers. Let s be a *non-empty* variable, p be a token, v be an identifier and *spec* be a specification. Let *head* be a function that returns the first token of a (wildcard)identifier. Then the matching function unrolls the identifiers as follows.

$$\begin{aligned}
&match(p.s, spec) = \\
&\quad = \begin{cases} \{x.match(s, Ty(x)) \mid x \in sp_d(spec) \wedge head(s) \in sp_d(Ty(x))\} & \text{if } p = \text{'*'} \\ p.match(s, p) & \text{otherwise} \end{cases} \\
&match(v, spec) = \\
&\quad = v
\end{aligned}$$

Note that this functions returns a tree-like structure, where each path from root to leaf denotes a valid variable. The structure is well-formed, if all the leaves are of the same type. Without getting into further details, let $matched(s, spec)$ denote the set of all such identifiers obtained from the well formed structure $match(s, spec)$.

To rewrite the axioms with wildcards on the right-hand side in a specification *spec*, we generate a set of new axioms, the postconditions of which are drawn from the set $matched(Post_*, spec)$, where $Post_*$ is the postcondition which includes wildcards. Let R_* denote a well formed construct on the right-hand side of an axiom which includes an identifier with wildcards and let _ denote an arbitrary well-formed syntactic construct in the current context. Then

$$\mathcal{RW}(\_ \texttt{ -> } R_*)(spec) \quad = \quad \{\_ \texttt{ -> } R_*[Post_* \leftarrow x] \mid x \in matched(Post_*, spec)\}$$

When an axiom includes wildcards in the left-hand side of the axiom, we add an alias construction to the specification, and then rewrite the axiom.

Let Pre_* be the precondition which includes wildcards, let x be a fresh variable and *list* a comma-separated list of variables from the set $matched(Pre_*, spec)$. We then rewrite the axiom, replacing the precondition Pre_* with variable x.

$$\mathcal{R}W(L_* \texttt{ -> } \_\,)(spec) \;=\; \{\texttt{alias}\ x \texttt{ = } (list)\} \cup \{L_*[Pre_* \leftarrow x] \texttt{ -> } \_\,\}\ .$$

Equations

Equations are rewritten into axioms for each variable in the equation (of course only under the assumption that the equation is solvable for this particular variable). Let A range over well-formed arithmetic expressions, its arguments denoting the variables used in the expression.

$$\begin{aligned}
&\mathcal{R}W(A_a(x_1, ...x_k) \texttt{ = } A_b(x_{k+1}, ..., x_n))(spec) = \\
&= \;\; \{x_2, ..., x_n \;\; \texttt{->} \;\; x_1 \;\; \texttt{\{}A_1(x_2, ..., x_n)\texttt{\}}\} \\
&\cup \;\; ... \\
&\cup \;\; \{x_1, ..., x_{n-1} \;\; \texttt{->} \;\; x_n \;\; \texttt{\{}A_n(x_1, ..., x_{n-1})\texttt{\}}\}
\end{aligned}$$

Aliases

Aliases are a special case in the extensions – part of it is rewritten into the core language and part of it is not. When two *alias* constructs are bound together, all of their components are bound together as explained in section 3.4.2, these new bindings are added to the specification. Note however the main construct is not removed, and its semantics will be given in the following sections.

3.5.2 Unfolding

Specifications typically have a hierarchical structure. While this is very convenient during the time of specifying the components, this structure needs to be flattened when translating the specification into the logical language in order to simplify the planning process. The specification will then be represented by a graph where the variables and methods realizing axioms are nodes, and edges represent relations between them.

Example. The following is an example where instances of one class are declared and used in another class.

```
specification ClassDiagram {
  UMLClass c1, c2;
  String dir;
  boolean done
  c1.parent = c2.name;
  c1.written, c2.written, dir -> done {copy};
}

specification UMLClass {
  String name;
  String parent;
  String[] attrs;
  boolean written;
  name, attrs -> written {writeFile};
}
```

After unfolding this specification we obtain the following relations:

```
ClassDiagram.c1.parent = ClassDiagram.c2.name;
ClassDiagram.c1.name, ClassDiagram.c1.attrs ->
      ClassDiagram.c1.written {ClassDiagram.c1.writeFile};
ClassDiagram.c2.name, ClassDiagram.c2.attrs ->
      ClassDiagram.c2.written {ClassDiagram.c2.writeFile};
ClassDiagram.c1.written, ClassDiagram.c2.written,
      ClassDiagram.dir -> ClassDiagram.copy {ClassDiagram.copy};
```

As can be seen, the specification is flattened by using compound names. In this sense, the structure is still present, but we can regard it simply as a graph.

The full definition of the unfolding function will be given in the next section together with the semantic function.

3.5.3 The logical semantics of specifications

The logical semantics of specifications is given by a translation from the specification into the logical language.

In defining the semantics, we will use the following syntax:

SP	denotes the set of valid classes with meta-interfaces
W	denotes the set of valid classes.
t	ranges over the set of types of interface variables.
v	ranges over the set of identifiers of specification variables.
L, R	ranges over well-formed syntactic constructs in the axioms in the left and right hand side respectively
o	ranges over the set of compound names.
m	ranges over the method names of the class which realize the axioms.
r	ranges over realizations of interface variables
$value$	ranges over values in the Java programming language.
$select_n$	is a function that returns the n-th element of the alias construction.
$alias$	is a constructor function for creating an alias structure.

In the following we let ':' to denote realizations. So for example $r_t : v$ means that r of type t is the realization of the interface variable v. *Constants* are treated as functions producing the value assigned to the constant variable. *Axioms* are realized by a pair – a method, and the object on which the method can be invoked.

We also make use of the function $sp : W \rightarrow \mathcal{P}(SExp)$ which returns the set of all expressions declared in a class specification.

The unfolding of a specification is done via the $unfold$ function. Unfolding is done during the translation of the specification. The translation process is started by unfolding the initial specification, by applying the semantic function on each specification expression in the specification. The unfold function is then called recursively when a component is declared in the specification.

$$unfold(t, o) = \{\mathcal{S}[\![sexp]\!](o) \mid sexp \in sp(t)\}$$

Definition 3.5.2. *Let LL be the set of valid formulae in the logical language and Var be the set of variables. The semantic function*

$$\mathcal{S} : SExp \rightarrow (Var \rightarrow LL)$$

is defined as follows

$$\begin{array}{rcl}
\mathcal{S}[\![t\ v]\!](o) & = & \begin{cases} r_t : o.v \cup unfold(t, o.v) & if\ t \in SP \\ r_t : o.v & otherwise \end{cases} \\
\mathcal{S}[\![t\ v_1, ..., v_n]\!](o) & = & \mathcal{S}[\![t\ v_1]\!] \cup ... \cup \mathcal{S}[\![t\ v_n]\!] \\
\mathcal{S}[\![o']\!](o) & = & o.o' \\
\mathcal{S}[\![L_1, .., L_n]\!](o) & = & \mathcal{S}[\![L_1]\!](o) \wedge ... \wedge \mathcal{S}[\![L_2]\!](o) \\
\mathcal{S}[\![L\ \texttt{->}\ R\{m\}]\!](o) & = & (r_t^o, m) : \mathcal{S}[\![L]\!](o) \supset \mathcal{S}[\![R]\!](o) \\
\mathcal{S}[\![\texttt{alias}\ v\ \texttt{=}\ \{v_1, ..., v_n\}]\!](o) & = & alias : \mathcal{S}[\![v_1]\!](o) \wedge ... \wedge \mathcal{S}[\![v_n]\!](o) \supset \mathcal{S}[\![v]\!](o) \\
 & \cup & select_1 : \mathcal{S}[\![v]\!](o) \supset \mathcal{S}[\![v_1]\!](o) \\
 & \cup & ... \\
 & \cup & select_n : \mathcal{S}[\![v]\!](o) \supset \mathcal{S}[\![v_n]\!](o) \\
\mathcal{S}[\![v \texttt{:=}\ value]\!](o) & = & value : \top \supset \mathcal{S}[\![v]\!](o) \\
\mathcal{S}[\![L\ \texttt{->}\ R]\!](o) & = & \mathcal{S}[\![L]\!](o) \supset \mathcal{S}[\![R]\!](o) \\
\mathcal{S}[\![[L\ \texttt{->}\ R]]\!](o) & = & \mathcal{S}[\![L\ \texttt{->}\ R]\!](o) \\
\mathcal{S}[\![\ \texttt{->}\ R]\!](o) & = & \top \supset \mathcal{S}[\![R]\!](o)
\end{array}$$

Note that the bindings are not present here in the semantics, this is because rewriting the equations already covers this case – a binding is rewritten into two axioms.

3.6 Java code generation

When ta specification has been translated into the logical language, the planning algorithm can be applied on it. The planning algorithm attempts to find a proof of the derivability of the goal variables. From that proof, an algorithm can be extracted. Below, we will discuss generating Java source code from the extracted algorithm.

Rewriting classes and applying methods

Firstly, the annotated Java classes are rewritten – all used interface variables except the control variables and aliases are added to the class as public instance variables. In case there is a naming collision with variables already declared in the Java class, the interface variables are renamed before adding them to the rewritten class.

Consider the following example.

```
public class Person {
  /*@
  specification Test {
    String name, address;
    name -> address {findAddress};
  }
  @*/

  String findAddress {String x) {
    ...
  }
}
```

During code generation, this class is rewritten into the following class, where the interface variables `name` and `address` are added to the new class as instance variables:

```
public class _Person_ {
  String name, address;

  String m {String x) {
    ...
  }
}
```

The bulk of the algorithm is usually made up of applications of axioms, which correspond to applications of methods in the Java programming language. When generating the code, each time a variable is inferred in the planning step, this step is added to the generated source code as an assignment. This means that all of the intermediate values are computed and saved in the instance variables. So all used interface variables become instance variables of Java objects and obtain concrete values when they are computed by the synthesized algorithm.

The generated code is *single assignment* [40], each variable is always assigned to only once.

Example. If we have a specification

```
int a, b, c, d;
a, b -> c {m1};
c -> d {m2};
```

and the goal of this specification is to compute d knowing a and b, then the synthesized program is

```
c = m1(a,b);
d = m2(c);
```

An alternative code satisfying the specification would be

```
d = m2(m1(a,b));
```

The advantage of the the first approach is that since all intermediate values are calculated and assigned to instance variables, the user could browse through them to see the data flow in the scheme or propagate the calculated values to the scheme.

Another case where this approach becomes advantegous is when we need to specify a method, which has to use a variable if its derivable, but can do without it if its not. Although such behavior can not be specified directly in SSP, we can push such specifications to Java sourcecode level, by adding a check to the Java method whether the variable is null or not (derivable). Then it is highly desirable that all intermediate values have been saved in instance variables.

The third case where this approach becomes useful is when applying the "compute all" strategy. To be more specific, in some cases it is not possible (or just too difficult) to state the goal in the specification in the form *assumptions* $\rightarrow$ *goals*. Instead, we would like to compute everything that can be computed based on the specification. Although expressing the goal "compute everything that can be computed" is not expressible in SSP (see [33]), we can add another planning mode that will go as far as it can with the planning process, and when it can not go any further (derive new variables), we declare the planning process to be successful. Then the user can see what exactly was calculated, and whether the result satisfies the user.

The generated program is added to the class as a method `compute`. The parameters of the method are the variables given as assumptions in the goal (if the goal is not specified, the method is parameterless).

Aliases

When two alias constructs are bound to each other, it means that all their components are bound to each other (this binding already happens on the preprocessing level).

If an alias construction only includes variables of one type, it can be regarded as an array (so it can for example be bound to an array, or given as input to a method with an array as one of its input parameters).

Example. In the following specification, an alias construct is bound to an array.

```
Obj[] d;
Obj a,b,c;
alias x = (a, b, c);
x = d;
```

The code generated from this specification (assuming variable d to be known) would be:

```
a = d[0];
b = d[1];
c = d[2];
```

Similarly, in the following specification a wildcard construction (which is translated into an alias) is given as input to the method.

```
Obj x, y, z;
int a;
*.b -> a {m}
```

Assuming that the variable b in components x, y and z has type *int*, the generated code is

```
int[] i = new int[3]
i[0] = x.b;
i[1] = y.b;
i[2] = z.b;
a = m(i);
```

Subtasks

Axioms can have subtasks as preconditions; they specify functional parameters of modules. The realization of a subtask is not a preprogrammed, but a synthesized module. Since Java does not have higher-order functions nor function pointers, the input parameter has to be an object which includes an instance method that implements the subtask. Although we can not know beforehand which class the subtask will be associated with, we can make use of Java polymorphism to overcome this. Therefore subtasks have to implement the subtask interface, which has the form

```
public interface Subtask {
   public Object[] run (Object[] input) throws Exception
}
```

The parameters of the subtask are passed through an array of objects. The drawback of this approach is that the primitive types have to be converted to objects, which will slow down the execution of the synthesized program. One way to address this problem would be adding subtask interfaces for primitive types, which could improve performance when the input of a subtask consists of a list of variables of primitive types. Also, in the future versions of Java, the casting operation will not be necessary, since from the version 1.5, Java supports autoboxing/unboxing [30], so the conversions are done automatically.

The actual subtask program generation is similar to the generation of the main program.

Equations

The code for realizing equations is very straightforward, the only exception is when the equation includes variables with different levels of precision (for example doubles and integers). In this case, an equation might be generated where an expression including double variables is assigned to the integer variable. In such a case the Java compiler will not compile this code, because a possible loss of precision can occur. Therefor, the expressions have to be cast into the correct type when generating Java source code.

Chapter 4

Implementation

The framework design presented in the previous chapters was implemented as a prototype; this chapter will give a brief overview of the implementation.

One of the goals of this work was to build a modular and extendable system, which could be used as a testbed for both the visual language framework and the Java synthesis mechanisms. Most of the design of the framework presented so far in this thesis has been implemented.

The framework was implemented using the Java programming language [24]. The choice was rather obvious, Java being a solid tool for software development, being platform independent, very robust and reliable and having excellent libraries (including libraries for user interface development). Most importantly of course, Java fully supports reflection and can dynamically (at runtime) load and use classes that were not loaded at the program start time. This gives us an excellent opportunity to make use of the generated Java code. We can compile it during runtime by calling the Java compiler, and then by using a customized classloader to load the classes, run the synthesized code, and extract the computed values.

The implementation can be divided into 2 main projects which are independent of each other (but are at the same time interoperable) - the scheme editor and the synthesizer. The scheme editor realizes the visual language design presented in chapter 2. The synthesizer, which includes a parser-translator, a planner, an optimizer, a code generator and an equation solver

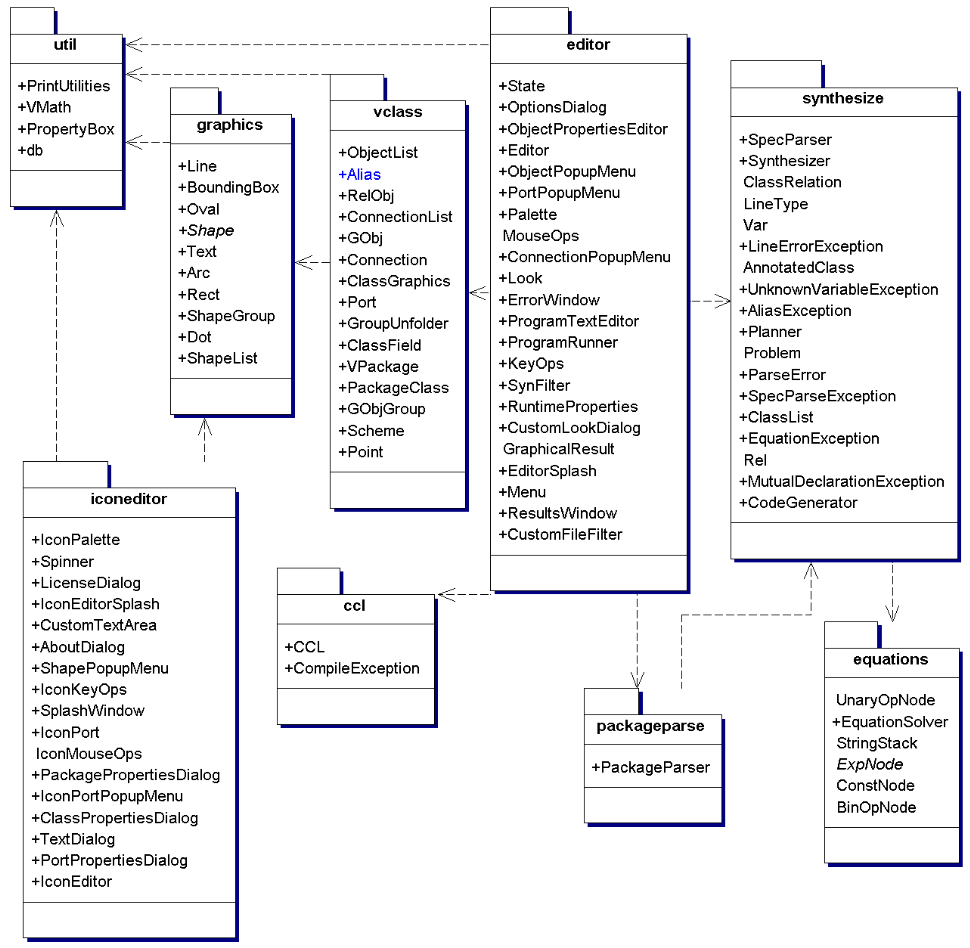


Figure 4.1: The package diagram

implements the Java program synthesis methodology presented in chapter 3. The interfaces between them are a class loader and -executor, which compile the synthesized program, execute it and send the results back to the scheme editor.

Figure 4.1 gives a rough overview of the implementation, showing the package diagram of the full system.

4.1 Scheme Editor

The scheme editor was built to implement an editor for the visual language described in chapter 2. It was implemented using Java Swing library [15]. The scheme editor provides the user interface for drawing visual specifications, and a parser for parsing the schemes into the specification language described in section 3.2. The design of the scheme editor is represented in figure 4.3.

The editor is fully syntax directed (unlike the Vilpert framework [45]). It means that the correctness of the scheme is forced during editing – drawing syntactically incorrect diagrams (for example a relation that is not connected to ports) is not possible. We believe it is the better approach – firstly, it allows us to skip the nontrivial task of developing a correctness checking parser for the visual language and secondly, we believe that a syntax directed editor is more usable.

The scheme editor uses visual packages which follow the format described in section 2.3.1. Visual packages are loaded and parsed via the SAX XML parser [35]; the scheme editor controls are customized according to the package – tool palette is built to include icons to access the classes in the package etc.

The classes and packages itself can be built using an icon editor developed in cooperation with Aulo Aasma(see [1]). It is a lightweight vector-based drawing tool, which lets the user to draw images of shallow visual classes, adding ports and fields to them etc. The shallow visual classes drawn in the icon editor can then be saved in the package format and directly be used by scheme editor.

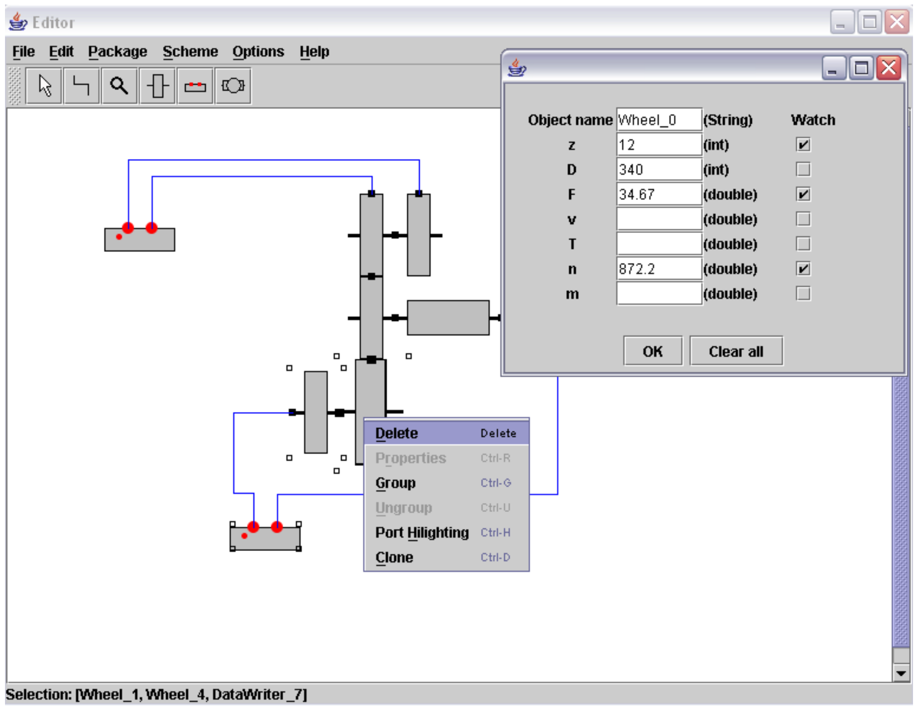
The scheme editor itself supports typical features needed for manipulating graphical objects, for example

- resizing objects
- grouping objects
- cloning objects or parts of scheme
- zooming in and out of the scheme
- saving and loading schemes

- switching packages during scheme drawing.

Figure 4.2 gives a sample screenshot of the scheme editor window. The main window includes a scheme, and two pop-up windows. The pop-up window on the right displays the editable attributes of an object and the pop-up window on the left displays the actions that can be invoked on the selected objects.

The `SpecGenerator` class takes care of parsing the scheme and outputting a specification suitable for Java synthesis. As mentioned however, the scheme editor can be used completely separately from the synthesizer. By defining a custom `SpecGenerator`, one could parse the scheme into an arbitrary format, which could for example take into account the positions and sizes of objects (in addition to their attributes and connections etc) and pipe the result to another program.



Object name	Wheel_0	(String)	Watch
z	12	(int)	✓
D	340	(int)	☐
F	34.67	(double)	✓
v		(double)	☐
T		(double)	☐
n	872.2	(double)	✓
m		(double)	☐

Selection: [Wheel_1, Wheel_4, DataWriter_7]

Figure 4.2: Scheme editor window

4.1.1 Visual feedback in scheme editor

An important feature of visual languages (often stressed as a major advantage over textual languages – see for example [8]) is immediate visual feedback – automatic display of semantic effects of program edits.

There are several ways how to provide visual feedback in the presented framework. Firstly the language developer could include visualizing components as a part of the semantics of certain domain concepts. Those components could make use of Java Swing or AWT API, collecting data from other components, and visualizing it as needed. This, however would require more extensive effort from the developer, but of course the expressive power of such visualization technique is only limited by the power of the Java language itself.

Another, simple yet quite effective and useful way to provide visual feedback is through using the `known` and `default` elements in the `field` definition in package format. These elements allow to define graphics to be displayed according to whether a particular interface variable is known or not. This means that a scheme can change when variables are computed (for example displaying the computed value, colouring part of an object in a different colour etc).

Also, work is under way on extending the package format, where the graphical attributes of visual classes could be bound to the values of their interface variables, for example the `width` attribute of the rectangle could be bound to the `width` attribute of the object, so that when a new value is computed, the look of the object changes accordingly.

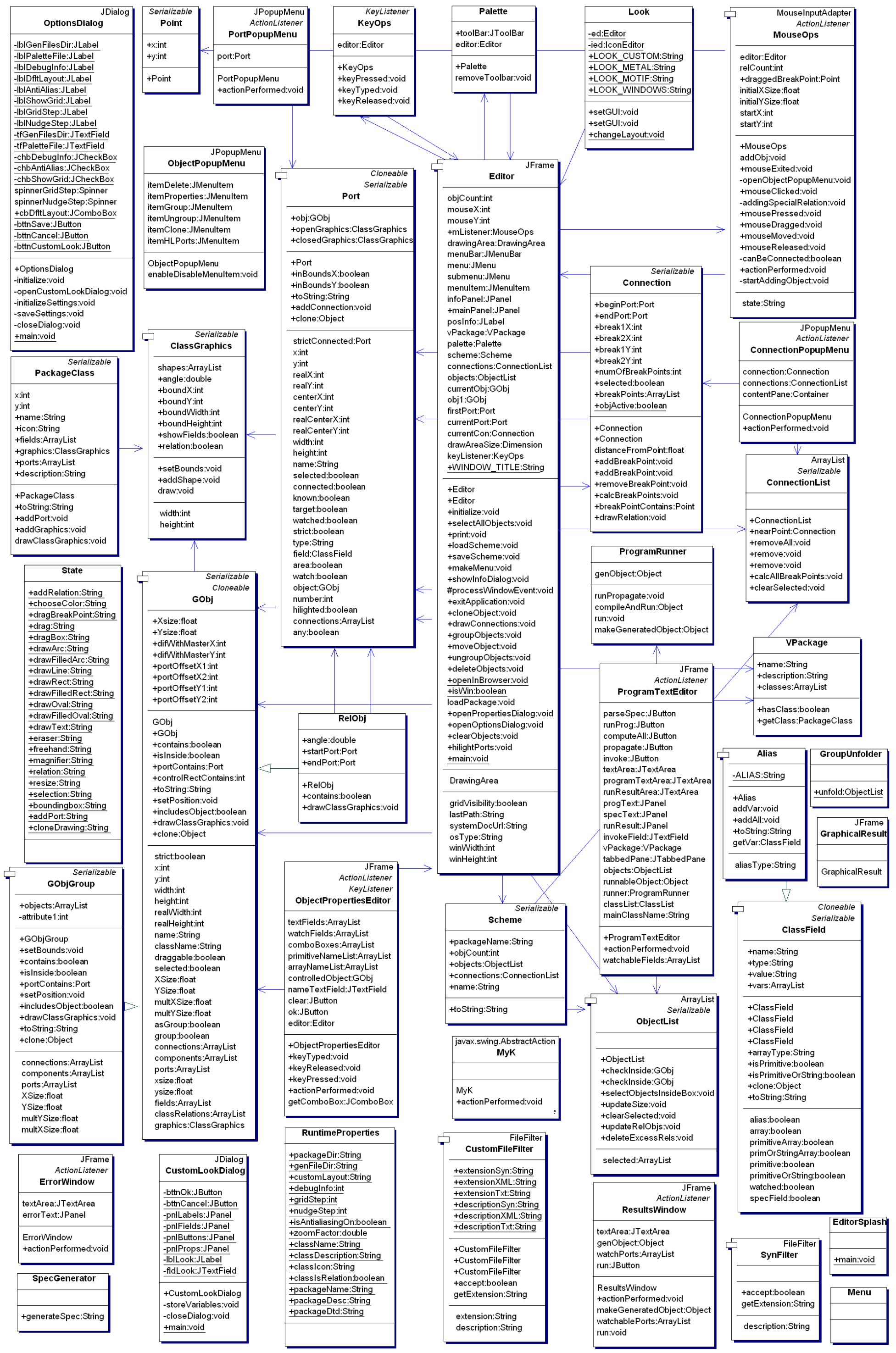


Figure 4.3: Class diagram of the scheme editor

4.2 Synthesizer

The synthesizer with its modules is designed and built as a completely separate library, usable independently from the scheme editor.

It can take over where the scheme editor left off. An interface between the synthesizer and scheme editor can pipe the generated specification to the synthesizer. The output of the scheme editor, a textual specification in the form described in section 3.2 is piped to the synthesizer, which parses the specification recursively, unfolding it and translating it into the logical language as described in section 3.1.

The graph built in this way is then submitted to the planner, which tries to find a proof for the problem specified. The steps in the proof form an algorithm, which will in turn be submitted to the code generator. The generated code can be compiled at runtime, and loaded back to the scheme editor via a custom classloader. The class can then be run, and the values read from it. Below, each major component of the synthesizer will be described in more detail. Its design is represented in figure 4.5.

4.2.1 Parser and translator

The unfolding and semantic functions described in section 3.5 are implemented in the `Parser` class. Starting with its input – the topmost specification in the aggregation hierarchy, the specifications are recursively parsed, meaning that each time when a new declaration is encountered in the specification, the parser checks whether an annotated Java class by this name exists in the path. If this is the case, the current file is pushed onto the stack, and the parser will start traversing the found file. This continues until the full specification hierarchy is parsed. Then, the translation function can translate the specification into a form suitable for planning. As explained in section 3.5 it builds a flat graph, the nodes of which are instances of `Var` and `Rel` classes (variables and relations). Each variable that occurs in the specification becomes an instance of `Var` and likewise relations become instances of

`Rel`. Each variable has a list of references to the relations where they occur as inputs, similarly, each relations has a list of references to variables which occur on the left hand side of the axiom represented by the relation and also a list of references to the variables which are postconditions of that axiom. Additionally, the relation instances are all initialized with an *input variable counter*, which is initially set to be number of input variables in the axiom, and a similar counter for subtasks. Such information and structure will later simplify the planning process. This graph – an instance of class `Problem` – can then be submitted to the planner.

4.2.2 Equation solver

The synthesizer includes an equation solver which realizes the extension to the core specification language as described in section 3.4.1. It is a simple analytical solver, applicable for equations which have one occurance of each variable.

When the parser encounters an equation in the specification, it passes it to the equation solver. The equation is parsed by the solver and the expression tree obtained is annotated – every node in the tree except leaves are annotated with a kind of an infix representation of its left and right branch(see figure 4.4). Then, all the equations for each separate variable can be obtained by traversing the tree and constructing the expression on the way. Consider for example the tree in the figure, representing an equation $a + b = c - d * 5$. When finding a solution for variable d, we would cover the path from the root to the d leaf, combining the annotations on all the opposing subtrees. First, we would take the expression on the left subtree

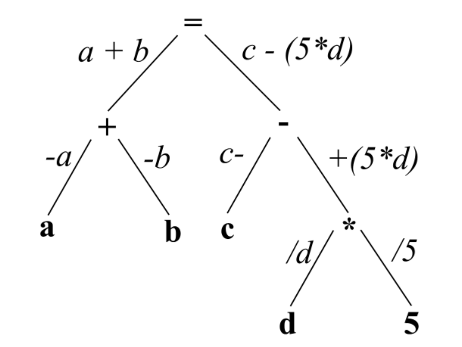


Figure 4.4: Annotated expression tree

$(a+b)$, move to the right, combine it with the expression at the left subtree (obtaining $c-(a+b)$), move to the left, combine it with the right subtree (obtaining $(c-(a+b))/5$). We would then reach variable d, thus the solution is $d=(c-(a+b))/5$. Similarly, we can find the correct equations for all the other variables.

4.2.3 Planner and optimizer

When the specification has been parsed and a graph representing the problem created, it can be submitted to the planner, which attempts to find a proof that the goal variables are derivable from the assumption variables in the specification. The planner has been developed together with Pavel Grigorenko [19]. The planning algorithm itself was first described in [48].

The planner starts working knowing the `Problem` graph and the set of assumption variables. The algorithm works by trying to derive goal variables by applying computability statements. A computability statement is applicable, when its input variable counter is zero. This means that the axioms with an empty left hand side can immediately be applied. The variables occurring in the right hand side of applied axioms are marked as computable variables, and the axiom is discharged. These variables(just as the assumption variables) can be used to reduce the variable counter of relations which have them as input variables. This continues until either all goal variables are marked as computable, or no further steps are possible (see algorithm in table 4.1). So the linear planning process is essentially a reachability check on the graph represented by the `Problem` instance.

If no further steps are possible in the linear planning process, then subtask selection is applied. It means that a subtask of a conditional computability statement, whose unconditional input variables have been derived is selected. For the new subtask, a sub-goal is generated and the planner tries to solve it. If it succeeds, and the computability statement does not have additional subtasks, the statement can be applied. Otherwise, the next subtask of this

Table 4.1: Linear planning algorithm

```
new variables derived = true
while not goal variables known && new variables derived do
  new variables derived = false
  for each known variable do
    for each of its relation do
      decrement the counter of relation by 1
      if counter of the relation == 0
        remove relation from relation list
        if not all of its output variables already found
          add output variables of relations to known variables
          add relation to algorithm
          new variables derived = true
        fi
      fi
    od
    remove variable from list of known variables
  od
od
```

statement is selected. If the solving of a subtask failed, a subtask of another axiom is selected. New subtasks are selected and new sub-goals generated until the goal is reached, or no further steps can be taken (see also [25]).

The algorithm guarantees that each variable is computed only once and each relation is applied only once in a particular subprogram.

Optimizer

The assumption driven forward search might generate an algorithm which computes variables that are not needed for computing the goal. Consider for example the specification

```
x -> y{m1};
x -> z{m2};
```

From the goal $x \rightarrow y$, the assumption driven forward search might derive the algorithm

```
z = m2(x);
y = m1(x);
```

It is obvious that the first step in the algorithm is unnecessary. In the worst case, the generated code could become extremely bloated.

The `Optimizer` class takes the generated algorithm, and optimizes it to only include necessary steps towards computing the goal. Starting with the set of goal variables and working bottom up, it checks whether the relation computes variables necessary for the goal. If this is the case, the input variables of the relation are added to the set of goal variables, otherwise, the relation is removed from the algorithm. This, although not guaranteeing optimal solution to the problem, removes all unnecessary steps from the synthesized algorithm.

4.2.4 Code generator

The `CodeGenerator` takes a list of `Rel` instances – an algorithm – and generates the Java classes which realize that algorithm. The main strategy for code generation was discussed in section 3.6. All the classes in the the aggregation hierarchy are copied to the `generated_files` directory and renamed adding underscores to the file (eg `_FileName_.java`). The main generated file (which is the topmost class in the aggregation hierarchy of a specification and which might include classes realizing subtasks) is also added to the directory, on this class's instance the generated `compute` method will be invoked.

4.2.5 Dynamically loading generated programs

Two classes, `CCL` and `ProgramRunner` provide the interface between the synthesizer and the scheme editor. Typically, when generating source code at runtime, using its functionality from within the generating program itself would be very inconvenient, requiring establishing an auxiliary protocol for realizing this. Fortunately, we can make use of Java's built in support for reflection (see [12]). The reflection API represents the classes and objects in the current Java virtual machine. Using the API, it is possible to create an instance of a class whose name or contents is not known at loadtime and also get and set values of fields or invoke methods which are not known at loadtime.

When the Java code has been generated by the `CodeGenerator`, the name of the file is passed to the compiling classloader (`CCL`). The compiling classloader uses Java `Runtime` library, spawning a `Javac` compiler process, which will attempt to compile the generated code. If the compilation is successful, the `CCL` class provides a custom classloader, which attempts to read the class file, create a `Class` instance and load it. The loaded class can then be submitted to the `ProgramRunner`, which will invoke the main method of the class (`compute`). When the method has finished running, all computed values can be read from the classes, and sent to the scheme as necessary. 12345678 1 xx 2 xx 3 xx 4 xx

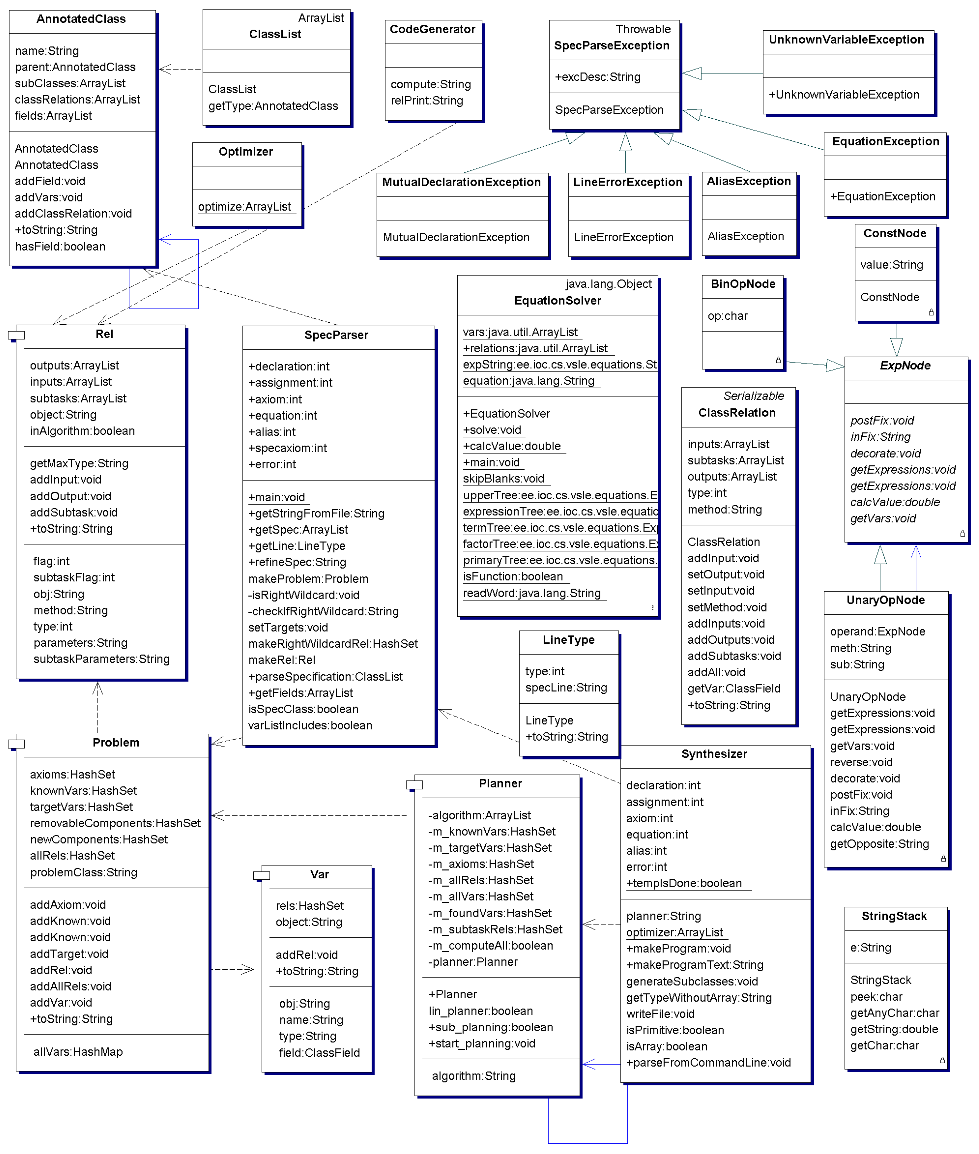


Figure 4.5: Class diagram of the synthesizer

Chapter 5

Experiments

Several experiments were conducted with the implemented system, to validate the design decisions and test its viability as a framework for CAD/CASE tool development. In this chapter, we will give an overview of some of the experimental packages that were developed. The experiments presented here are tests for functionality and applicability of the framework and its implementation. Tests for efficiency and speed of planning and code generation, or comparisons of performance of generated programs versus handcoded programs were not considered. Although there is a lot of room for optimizations in these areas, the development of the most efficient planner was not our goal and furthermore, in typical application areas the performance barrier should not be a problem.

UML diagrams

One of the most popular visual languages is UML or Unified Modeling Language (see also the overview in section 6.1.1). Among different CASE tools, UML diagram editors are very widely used in software industry.

Thus it was quite natural to try to implement the functionality of some typical UML diagrams in the developed framework. UML class diagram, use case diagram and activity diagram functionality was implemented. Below we will briefly report these experiments.

Class diagrams

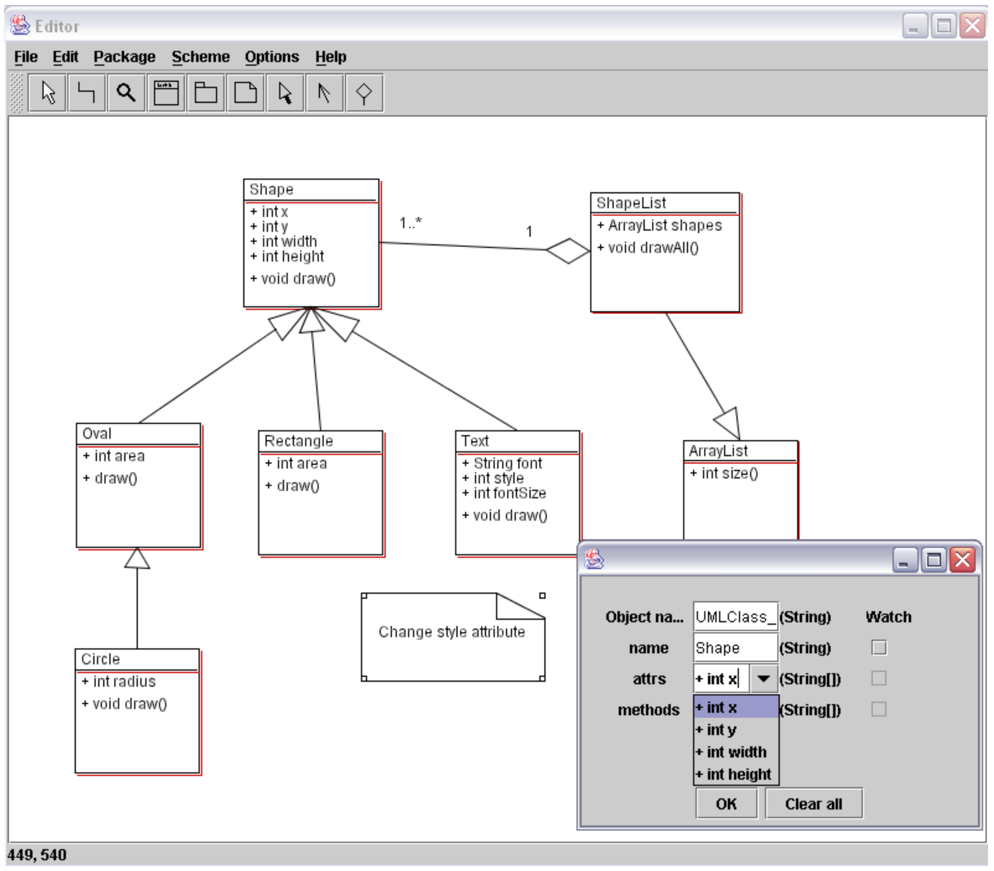
Class diagrams are the backbone of almost every object-oriented method, describing the static structure of a system. Classes represent an abstraction of entities in the system which have common characteristics (to avoid confusion with regular Java classes or visual classes, we will call them UML classes from now).

Implementing UML class diagram functionality in the framework was quite straightforward. First, it required drawing the diagram entities like UML classes and packages. Since class diagrams have several different types of directed relations, they also had to be defined as visual classes (association, aggregation, generalization). A sample screenshot of the defined package in use is presented in figure 5.1.

Most UML tools allow generating source code from class diagrams – the diagram can be turned into a set of Java or C++ classes, which correspond to the structure presented in the diagram. This offered us a chance not only to test the scheme editor for its applicability, but also to make use of our synthesis techniques. In order to implement this functionality in the framework, semantics had to be given to UML classes and generalization relations via writing Java classes providing the necessary methods. Then we could use the synthesizer to synthesize a program, which exports the class diagram as Java source files.

As an example, we present the specification of the UML class, which is needed for code generation.

```
/*@
  specification UMLClass  {
    String name;
    String parent;
    String[] attrs;
    String[] methods;
    void done;
    [done -> done], name -> done {writeClass};
  }
@*/
```



Object na...	UMLClass_	(String)	Watch
name	Shape	(String)	☐
attrs	+ int x	(String[])	☐
methods	+ int x + int y + int width + int height	(String[])	☐

Figure 5.1: A class diagram of graphics package, modeled in the scheme editor

Note that the empty and always solvable subtask (empty in the sense that it doesn't compute anything) is added to the axiom in order to force its application after all other computations (which do not require subtasks as input) are complete. This is to ensure that the class is written into a file only when all its components that can be computed, actually are computed.

Activity diagrams and use case diagrams

Activity diagrams describe the workflow behavior of a system. The diagrams describe the state of activities by showing the sequence of activities performed. They can show activities that are conditional or parallel [3].

The use case diagrams are used to identify the primary elements and

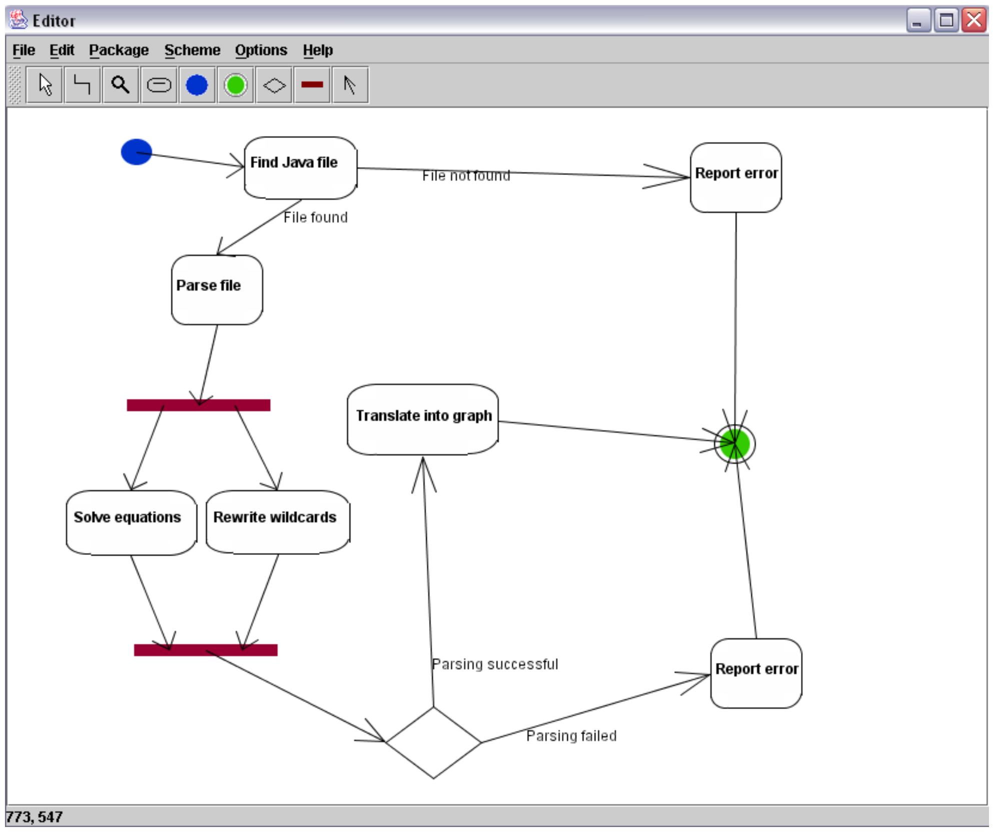


Figure 5.2: An activity diagram of the parsing process

processes that form the system. The primary elements are termed as "actors" and the processes are called "use cases." The use case diagram shows which actors interact with each use case [3].

Implementing those diagrams was quite similar to implementing the class diagrams, and much of the code could be reused. As with class diagrams, features such as defining classes as relations and the ability to add labels to directed relations turned out to be very useful. Sample screenshots of the two packages in use can be seen in figures 5.2 and 5.3.

Gearbox

Another conducted experiment was developing a simple CAD tool for calculating the kinematics of a gearbox, a problem from mechanical engineering

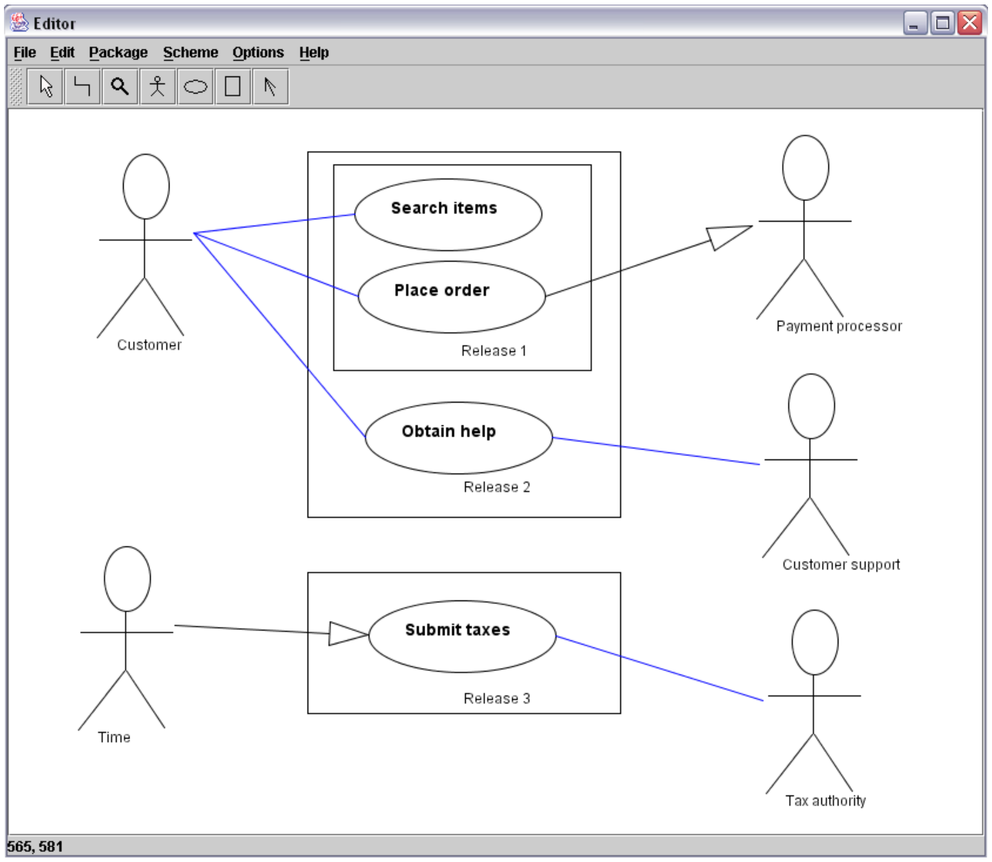


Figure 5.3: A sample use case diagram modeled in the scheme edito

domain (an example taken from the lecture notes of professor Ahto Kalja). Although this is a small example, it is interesting in the sense that we can make use of several extensions that were made both to the visual and the specification language, for example spatial relationships, equations and aliases.

The main entity in the chosen problem domain is a gear – a toothed wheel designed to transmit torque to other gears. We consider the case of a spur gear, which is a flat wheel that has teeth projecting radially in the plane of the wheel. Spur gears can only be fitted on parallel axis.

Gears can be connected to each other both axially and tangentially (through their teeth). In the scheme editor, it can be done by either placing two gears next to each other, or above each other – these are typical spatial relationships. When objects are placed next to each other, a relation is established between the touching ports, this is done automatically by the scheme editor. Immediate visual feedback is given – the edges are drawn in a different shape and colour, to denote that they have been connected.

Aliases were used in the example to group attributes which can be passed between objects together. For example, when the gears are connected to each other tangentially, it means that their tangential speed, module of the teeth, and tangential force are equal. To avoid defining three ports for the three attributes, they can be grouped together by an alias construction, and bound by just a single port.

The specification of the gear is the following.

```
class Gear {
  /*@
  specification Gear {
    int z, D;
    double v, F, T, n, m;
    F=1000*T/(0.5*D);
    D=m*z;
    v=3.14*n*D/1000;
    alias axial = (n, T);
    alias tang = (v, m, F);
  }
  @*/
}
```

What is also interesting in this example, is that the main entity – the gear, is only defined via equations, i.e. no Java source code is needed to describe its full functionality. Without using equations, the specification would have required nine axioms, and nine methods implementing these axioms. This is a good indication of the usefulness of the equation solver in certain cases.

Note that after code generation the seemingly empty Java class will come into use, since the needed interface variables are added to the class via class rewriting, so that intermediate values can be saved into them (as described in section 3.6).

Other entities in the package were a *Motor* and a *DataWriter* class. The *Motor* class was added to have the ability to automatically generate values as needed, for example gradually increasing the speed of the gear. The *DataWriter* class can be used to save values in a file in the CSV (comma separated values) format. When its input is connected to a port of a gear it can take those values and save them to a file.

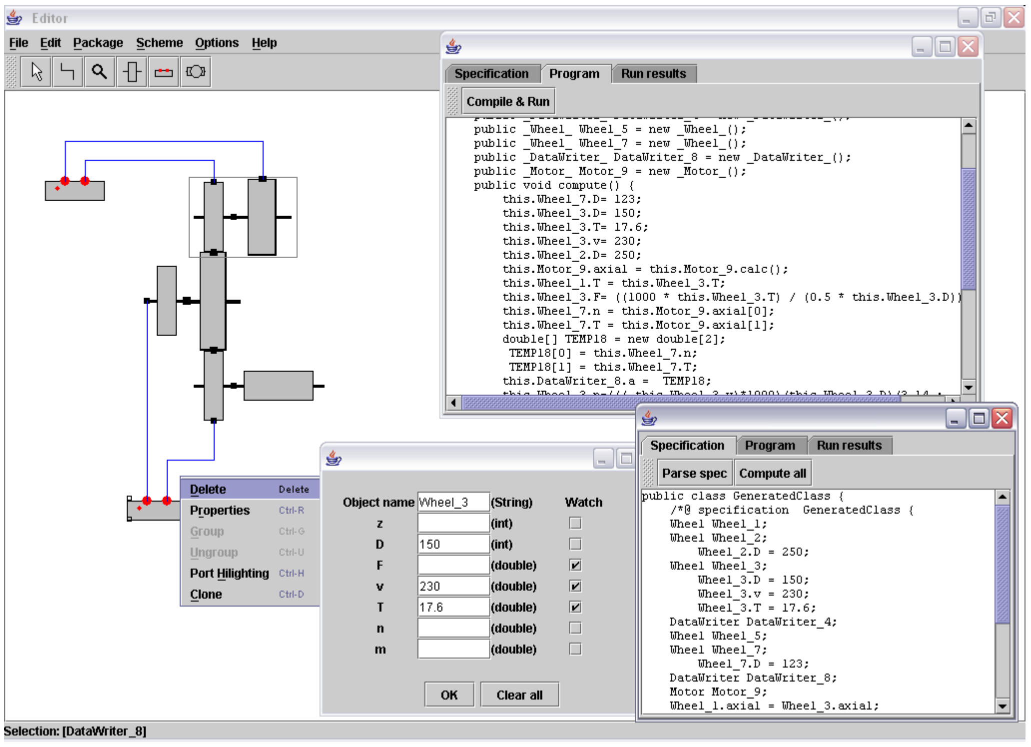
This actually indicates quite well how convenient it is to use existing Java libraries to give semantics for components. Instead of using the Java IO package we could as easily have uploaded the data to an ftp site, written it to a database, sent the data via e-mail, or established an interface with a mathematical analysis program like Mathematica to analyze the results(for example via JLink [43]). This could be done in just a few lines of code, by writing a method that calls the particular library, and adding a specfication to it. This is a clear advantage over a custom scripting language (like in the case of MetaEdit+ [44]), where all that functionality should be written from scratch.

The screenshot in figure 5.4 shows a scheme of a gearbox drawn in the framework. It displays the main window, together with the object property window, the textual specification of the scheme and the synthesized code window.

Conclusions based on experiments

In general, the framework and the implementation seem to be quite flexible and conceptually powerful enough for implementing more advanced domain-specific visual languages.

It allows rather effective code reuse. For example, the full package defining the class diagram functionality was about 200 lines of XML code. The classes realizing the functionality for creating class templates for the diagrams were 40 and 12 lines, respectively. The activity- and use case diagram package sizes were around 240 and 140 lines. So, each package comprises around 1 -



```
    public _Wheel_ Wheel_5 = new _Wheel_();
    public _Wheel_ Wheel_7 = new _Wheel_();
    public _DataWriter_ DataWriter_8 = new _DataWriter_();
    public _Motor_ Motor_9 = new _Motor_();
    public void compute() {
        this.Wheel_7.D= 123;
        this.Wheel_3.D= 150;
        this.Wheel_3.T= 17.6;
        this.Wheel_3.v= 230;
        this.Wheel_2.D= 250;
        this.Motor_9.axial = this.Motor_9.calc();
        this.Wheel_1.T = this.Wheel_3.T;
        this.Wheel_3.F= ((1000 * this.Wheel_3.T) / (0.5 * this.Wheel_3.D))
        this.Wheel_7.n = this.Motor_9.axial[0];
        this.Wheel_7.T = this.Motor_9.axial[1];
        double[] TEMP18 = new double[2];
         TEMP18[0] = this.Wheel_7.n;
         TEMP18[1] = this.Wheel_7.T;
        this.DataWriter_8.a =  TEMP18;
```

Object name	Wheel_3	(String)	Watch
z		(int)	☐
D	150	(int)	☐
F		(double)	✓
v	230	(double)	✓
T	17.6	(double)	✓
n		(double)	☐
m		(double)	☐

```
public class GeneratedClass {
    /*@ specification  GeneratedClass {
    Wheel Wheel_1;
    Wheel Wheel_2;
        Wheel_2.D = 250;
    Wheel Wheel_3;
        Wheel_3.D = 150;
        Wheel_3.v = 230;
        Wheel_3.T = 17.6;
    DataWriter DataWriter_4;
    Wheel Wheel_5;
    Wheel Wheel_7;
        Wheel_7.D = 123;
    DataWriter DataWriter_8;
    Motor Motor_9;
    Wheel_1.axial = Wheel_3.axial;
```

Figure 5.4: A scheme of a gearbox

2% of the total size of the application. Furthermore, note that most of this code does not need to be written by hand, but is generated by the icon editor when the classes are drawn.

For comparison – Tuovinen reports in his PhD thesis([46]) on implementing the typical UML diagrams in the VILPERT framework (described in more detail in section 6.2.3), where the language specific code was about 20% of the full application code (3700 lines of code). The two frameworks themselves are comparable in bulk code size.

Considering the UI, there is of course still a lot of work to be done. The hierarchical composition is only partly implemented, via the possibility to group objects and the option of defining new classes as a composition of other classes. This has to be extended and made more straightforward. Also, the interface between the scheme editor and the synthesizer needs to be improved, to make it more convenient and user-friendly.

Among missing features is the possibility of editing object attributes directly on the canvas(right now this has to be done through object property window). Although a seemingly minor detail, it makes using the UI much more convenient in many cases. It is also very easily implementable, since values of attributes can be displayed on screen already now, so making them editable should be quite straightforward.

The language for defining packages will probably need to be extended, to allow a wider variety of features to be implemented. These requirements will hopefully become imminent when experimenting further with the system and trying to implement more sophisticated CAD or CASE tools.

Chapter 6

Overview of related work

In this thesis, we bring together several different areas of computer science – visual languages, specification languages and program synthesis. In this section, we will give an overview of some of the related work that is most relevant considering the topic of this thesis.

6.1 Specification languages

6.1.1 UML

UML, the Unified Modeling Language [6], is used to model object-oriented (software) systems, offering a graphical notation and some general concepts to visualize, specify, construct and document software. It is derived primarily from three notations: Booch's OOD (Object-Oriented Design), Rumbaugh's OMT (Object Modeling Technique), and Jacobson's OOSE (Object-Oriented Software Engineering). UML was developed in the mid-90's and standardized by the Object Management Group in 1997. It is probably the most widely used visual specification language for software development and accepted as an industry standard.

UML is based on different views(models) of the system. It defines twelve types of diagrams, divided into three categories. Four diagram types represent static application structure; five represent different aspects of dynamic behavior, and three represent ways you can organize and manage your application modules [20].

Structural Diagrams include:

- Class diagram, which shows a set of classes, interfaces and their relationships,
- Object diagram, which is the a static snapshot of instances of classes in class diagrams,
- Component diagram, showing the organization and dependencies among the set of components addressing the static implementation view of a system.
- Deployment diagram, which shows the configuration of run-time processing nodes and components that live on them.

Behavior diagrams include the

- Use case diagram which is used to model the behavior of a system, showing a set of use cases and actors and their relationships
- Sequence diagram that emphasizes the time-ordering of messages sent between objects
- Activity diagram addressing the dynamic view of a system. It shows the flow from activity to activity within a system
- Collaboration diagram which shows the organization of objects sending and receiving messages.
- Statechart diagram, which is a state machine, consisting of states, transitions, events and activities.

Model management diagrams include :

- Packages
- Subsystems

- Models.

The multiple of diagrams allows the users to model the system from different viewpoints, allowing an easy separation of concerns. While in some ways it is a good approach, there are, however, drawbacks. As specifications consist of only loosely coupled models, tied together by very few and semantically weak rules it means that it is usually not possible to specify how these models should work together. This leads to several deficiencies in the UML – use case models cannot express state-dependent system behavior adequately, let the system initiate an interaction or express structure between use cases. Moreover, UML cannot model the behavior of high-level system components such as subsystems. For a thorough study these deficiencies, see [16].

As has been pointed out in [28] other problems can arise from its lack of precise semantics. UML's semiformal semantics can not always give a unique and meaningful sense to all concepts. This means that there are no sound basis for tools that may perform advanced operations on a UML model, eg concreteness checking, simulation or code generation.

Still, for the practical use of a semiformal requirements specification language, UML is highly effective and as an industry standard has at the moment no alternatives.

6.1.2 SDL

SDL, the Specification and Definition Language is a formal and visual modeling language standardized by the International Telecom Union. It is primarily intended for the specification of real-time, event-driven, interactive applications in telecom, distributed and embedded systems. The language is mainly based on the concept of extended finite state machines and consists of the following components [10]:

- *Structure* is the hierarchy of systems, blocks, processes and procedures. Each SDL process type is defined as a nested hierarchical state machine. Each substate machine is implemented in a procedure. Procedures can be recursive, and there scope can be defined as local or global. Memory spaces are separate in SDL, i.e. data is local to a process or a procedure. The static structure of the system is defined in terms of blocks and channels, the dynamic structure of a system

- *Communication.* There are two basic communication mechanisms in SDL - asynchronous signals and synchronous remote procedure calls. Both of these mechanisms can carry parameters to interchange information between SDL processes (and also between an SDL system and its surrounding environment). Signal and process priorities are however not in the scope of SDL.

- *Behavior* of a system is described in the processes. Processes can be created at system start, and created and terminated during runtime. Also, more than one instance of a process can exist.

- *Data.* SDL accepts two ways of describing data, abstract data type, and ASN.1 (abstract syntax notation). Abstract data types have no specified data structure - it specifies a set of values, a set of operations allowed and a set of equations these operations must fulfill.

- *Inheritance.*

Compared to UML, SDL is more suited to describe behavior and communication (since thats what it was mostly designed for), another advantage is that formal semantics of SDL provide application designers with capabilities for model testing and automatic code generators. SDL is not as good however when it comes to type level, it is weak if reuse, extensibilty and typing are needed [28]. The latest version of SDL is, however adddressing some of these issues.

6.1.3 ADORA

ADORA (Analysis and Description of Requirements and Architecture) is an approach to object oriented modeling developed in University of Zürich as an alternative to UML. It is based on object modeling and hierarchical decomposition, using an integrated model. The ADORA language is intended to be used for requirements specifications and high-level, logical views of software architectures [17]. The main contributions of the ADORA language are the integration of different aspects of a system into one model and a concept for systematic hierarchical decomposition of models [5]. In ADORA, the model is decomposed recursively from objects to objects, so in theory it gives an opportunity for object modeling on all levels of the hierarchy and abstraction. Another feature of ADORA is its ability to visualize a model in its context – a component can always be visualized in its surrounding hierarchical context.

ADORA vs UML

One of the main differences between Adora and UML is that ADORA model integrates all modeling aspects(structure, data, behavior etc) in one coherent model [5]. In UML, the model consists of a set of diagrams that are only loosely coupled. This approach makes it very easy to achieve separation of concerns, but on the other hand, it makes it difficult to make the model consistent. The authors of ADORA claim to have achieved consistency by using a decomposition mechanism where the model is decomposed recursively into smaller parts. The separation of concerns is then achieved using a view concept, i.e. there are different views illustrating different aspects of the system. Readability of the diagram is achieved by selecting the right level of abstraction. Another major difference is that unlike UML (and most object-oriented modeling methods) ADORA does not use class diagrams as the cornerstone to the model. Instead, ADORA uses abstract, prototypical objects, so the abstract object is a placeholder for a concrete object instance.

6.2 Frameworks

6.2.1 The NUT System

The NUT system ([50], [47]) is a programming tool supporting declarative programming in a high-level language, automatic program synthesis and visual specification of classes. It has been developed in the Institute of Cybernetics, Tallinn and KTH, Stockholm. Structural Synthesis of Programs (SSP), which uses intuitionistic propositional logic, is the central part of of the system.

In the NUT system, specifications of classes are used for automatic construction of programs using the SSP method. Just like the Amphion system, NUT restricts its attention to constructing programs from pre-programmed modules, rather than from primitive instructions of a programming language. The NUT specification language is an object-oriented language extended with features for program synthesis, the pre-programmed modules are methods of classes supplied with specifications.

6.2.2 Amphion

The Amphion system [29] synthesizes programs by composing existing subroutines. It provides automation of the software development process and improves the productivity and quality of software engineering.The Amphion system framework uses automated theorem proving techniques developed by Waldinger. In this sense it is similar to the NUT system (and the approach presented in this thesis), where software is also constructed from preprogrammed modules.

Amphion's main application domain has been planning and interpreting space-science observations. In Ampion, software requirements are specified in a graphical notation. The graphical specification is translated into a first-order logic theorem and a program is developed based on the logical form of the specification using the automated deduction system.

6.2.3 Vilpert

Vilpert, or visual language expert is an object oriented framework for implementing visual languages in the Java language [46]. Implementing a visual language with Vilpert means generating a language analyzer based on a formal syntactic specification and implementing a graphical editor for manipulating the programs. It has a language specification sub-framework that is based on the grammatical model called atomic relational grammars [45]. The diagrams are drawn in free order (not dictated by a syntax directed editor). The language analyzer can then be used to interpret the drawing, so the correctness of a diagram is not enforced during editing.

In theory, semantics could also be give to the language (although it is not particularly straightforward). To define the (operational) semantics of the language, the grammar writer can define additional (semantic) methods and attributes for the language which are to be invoked on the parse tree after parsing.

6.2.4 MetaEdit

MetaEdit+ ([54], [44]) is a tool for creating CASE tools that comprise a domain specific visual language. To specify a language, the user specifies (with the method workbench toolset) the concepts of the domain, the rules for using and composing the concepts, the graphical notation that corresponds to the concepts, and a set of generators (specified in a scripting language) that transform models into some external format (code, documentation, data dictionary, etc.). The goal is to create a complete CASE tool tailored for a specific domain and a specific development process. The heart of the approach is the creation of a metamodel that specifies the domain specific languages. The elements of the GOPRR metamodeling language are graph, object, property, relationship, and role. A model is a graph consisting of objects with properties and relationships to other objects. Objects may have roles in the relationships that they participate in. The method workbench of MetaEdit+ provides tools for specifying all these elements of a domain specific language. The system then derives from the specification a set of (syntactic) wellformedness rules that the specifier can tailor (choose which rules to include in the final method). The specifier can also create more complex (e.g. semantic) checks on the models using the scripting language provided by the tool. The specification of the generators for the target language is also based on writing processors in the scripting language for the models created with the target language. MetaEdit+ can support only those kind of languages that can be expressed using the GOPRR metamodeling language, i.e. graphs of objects.

6.3 Java behavioural specification languages

6.3.1 JML

There is a lot of ongoing work to create a behavioral interface specification language for Java. One of the main efforts in this field is JML, or Java Modeling Language [27], started in Iowa State University. JML assertions are written as special comments in Java code, either after `//@` or between `/*@...*/` JML annotations are used to augment the syntactic interface of Java code, the class or interface's method signatures, attribute types etc, to more precisely indicate the correct usage of the API. The central ingredients of JML specification are preconditions, postconditions and class and interface invariants. They are expressed as boolean expressions in JML's syntax.
Example [27].

```
public class IntMath {              //1
  /*@ requires y >= 0;              //2
      assignable \nothing;          //3
      ensures 0<= \result           //4
  */                                //5

  public static int isqrt(int y) {  //7
    return (int)Math.sqrt(y);       //8
  }                                 //8
}                                   //9
```

The above specification describes a Java class that contains one static method. C-style comments starting with `/*@` are annotations. On line 2, the keyword `requires` denotes a precondition, saying what must be true about the arguments. On line 3, the condition states that this method, when called, does not assign to any locations and on line 4, the postcondition `ensures` says that if the precondition is true, the method must terminate normally in the state satisfying the postcondition.

JML annotations can have several uses, there are for example tools for [7]

- runtime assertion checking and testing, i.e. running Java code and testing for violations of JML assertions
- static checking and verification
- generating specifications.

While syntactically JML is quite similar to the specification language described in this thesis, their goals are very different – checking the correctness of programs versus enabling automated composition of software. However it would be quite interesting to use these two languages together – while our method for synthesis of software guarantees the correctness of the program structure (with respect to the specification), we can not guarantee the correctness of each separate module, which are written by hand. These modules could be proved correct via the help of JML, thus guaranteeing the correctness of whole program. There have been similar attempts before, for example in [42] Tammet used first-order annotations of axioms in SSP to prove the correctness of some synthesized set manipulation programs.

Chapter 7

Conclusions and future work

In this thesis we have presented a framework for designing and implementing visual specification languages. The work concentrated on two very different fields, visual languages and program synthesis. One of the main goals of the work was to bring these two together, to enable compiling visual languages into executable code.

In chapter 2 we proposed the visual language. The abstract syntax and the semantics of the language was given. The semantics of the visual language was described on two different levels. Firstly, we considered the shallow semantics, which is the textual representation of the graph underlying the scheme. Secondly, we defined the deep semantics which was defined as a set of programs that can be automatically derived from a a scheme.

To enable giving concrete syntax to the visual languages developed in the framework, we proposed a markup language in XML format through which the language entities can be described.

In chapter 3 we briefly describe SSP, a method for synthesizing full programs from preprogrammed modules. We considered the case where these modules are Java classes and proposed a specification language that can be used to annotate Java classes to enable program synthesis. The semantics of the specification language is given through its translation into logic, and the code generation is described.

A big part of the thesis work was implementing the framework design as a fully functional prototype. This work has been reported in chapter 4. Among the developed modules was the scheme editor, which implements the visual language design presented in chapter 2 and the synthesizer, which implements the Java program synthesis methodology. These two modules were bound together via an interface which allows to send the specification obtained from the scheme to the synthesizer, and when the program has been generated, to compile and run it to send the results back to the scheme editor.

The experiments conducted with the system were reported in chapter 5. The experiments included implementing several UML diagrams in the framework, namely use case-, class- and activity diagrams. From the experiments we concluded that the framework design and implementation are quite flexible and conceptually powerful enough for implementing more advanced domain-specific visual languages.

7.1 Future work

We believe that the solution we have proposed is in many cases a viable option for CAD and CASE tool design and development.

Future efforts should be concentrated on improving the implementation of the the framework. The next items in the improvements request-list are fully functional hierarchical composition, tool support for incremental design of packages, a more advanced equation solver and an improved interface between the two major components of the system – the scheme editor and the synthesizer.

Further work should also include experiments with some more sophisticated examples. This would give a good indication of the shortcomings of the system and give us some more insight as what to should be improved or which features added.

Raamvärk visuaalsete keelte projekteerimiseks ja realiseerimiseks.

Ando Saabas

Resümee

Tarkvaraarenduse puhul on paljudel juhtudel seos probleemivaldkonna ja kasutatava programmeerimiskeele vahel väga väike. Enamasti on need on kaks eri maailma, millel mõlemal oma mõisted, mõttemallid ja eksperdid. Nende ühendamine, loomaks valdkonnaspetsiifilist tarkvara, tähendab üldjuhul seda, et arendaja peab probleemi lahendama kaks korda – esiteks konkreetse valdkonna terminites ja teiseks koodi maailmas.

Ideaaljuht oleks aga selline, kus arendaja loob lahenduse vaid ühe korra ja seda probleemvaldkonna (nt. visuaalse) mudelina. Selle mudeli põhjal oskaks arvuti siis juba automaatselt lõppprodukti genereerida, olgu selleks siis programmikood, arvutustulemused vmt. Paraku peavad mudeli loomiseks olema vastavad vahendid – modelleerimiskeel, arenduskeskkond ja metoodika mudelist lõppprodukti genereerimiseks. Modelleerimisvahendid nagu UML sellisel juhul palju ei aita, sest UML ei seostu mitte probleemvaldkonnaga, vaid koodiga, on ta ju mõeldud koodi visualiseerimiseks.

Käesolev magistritöö keskendubki neile probleemidele, pakkudes välja raamvärgi visuaalsete spetsifitseerimiskeelte kiireks ja efektiivseks loomiseks. Rõhk on asetatud sellele, et loodavale keelele oleks võimalik anda konkreetne ja formaalne semantika selleks, et ta oleks kompileeritav – automaatselt tõlgitav käivitatavasse koodi. Selle kaudu seob töö kahte väga erinevat teemat, visuaalseid keeli ja programmide sünteesi.

Antud lähenemises defineeritakse visuaalne keel kompositsiooniliselt lähtudes probleemvaldkonna ontoloogiast. Valdkonna mõistete semantika esitatakse läbi Java klasside, milles kirjeldatakse nende funktsionaalsus. Selleks, et Java klasse oleks võimalik kasutada programmide automaatseks sünteesiks, lisatakse neile spetsifikatsioonid – meetodite eel- ja järeltingimused, mis määravad, kuidas konkreetset klassi arvutuslikult kasutada saab – mida ta arvutab, mis

tingimustel ning millised on tema seosed teiste klassidega. Lisaks mõistete funktsionaalsuse kirjeldamisele, peab olema neile võimalik määrata ka väljanägemise ja reeglid (visuaalseks) sidumiseks teiste komponentidega.

Niimoodi defineeritud mõistetest/komponentidest on kasutajal võimalik kokku seada skeem ehk siis kasutada ülalkirjeldatud viisil defineeritud keelt visuaalseks programmeerimiseks, seejuures nii, et loodud skeemid on võimalik automaatselt teisendada kompileeruvaks Java koodiks.

Töös on esitatud loodavate keelte abstraktne süntaks (see on antud kui märgistustega lihtgraaf); keele semantika on esitatud kahel tasemel – madala taseme semantika kui skeemigraafi tekstiline esitus (mille põhjal toimub automaatne programmide genereerimine) ning süvasemantika kui programmide kogum, mis spetsifikatsiooni põhjal võimalik genereerida. Loodavale keelele konkreetsüntaksi defineerimiseks on töös välja pakutud XML-il põhinev märgistuskeel.

Programmide sünteesi metoodika põhineb SSP-l (Structrural Synthesis of Programs). Esitatud on spetsifitseerimiskeel Java klasside kirjeldamiseks, toodud selle keele semantika ning antud ülevaade spetsifikatsioonide põhjal Java koodi genereerimisest.

Töös kirjeldatakse ka täisfunktsionaalset prototüüpi, mis realiseerib toodud raamistiku disaini. Programm koosneb kahest põhikomponendist – skeemiredaktorist, mis realiseerib visuaalse keele disaini ja sünteesimootorist, mis võimaldab automaatset programmide genereerimist spetsifikatsioonidest.

Töös on esitatud ka eksperimendid loodud prototüübiga, loodud on näiteks paketid UML keele klassiskeemide, kasutuslooskeemide ja tegevusskeemide loomiseks.

Bibliography

[1] Aulo Aasma. *Visuaalsete keelte konstruktor*. Diploma thesis, Tallinna Tehnikaülikool, 2004.

[2] M. Aksit. Separation and composition of concerns in the object-oriented model. In *ACM Computing Surveys*, volume Vol. 28, 1996.

[3] Scott W. Ambler. *Agile Model Driven Development with UML 2*. Cambridge University Press, 2004.

[4] Don Batory, Jia Liu, and Jacob Neal Sarvela. Refinements and multi-dimensional separation of concerns. In *Proceedings of the 9th European software engineering conference*, pages 48 – 57, 2003.

[5] Stefan Berner, Nancy Schett, Xia Yong, and Martin Glinz. An experimental validation of the ADORA language. Technical Report No. 99-07, Universität Zürich, 2000.

[6] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2000.

[7] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Wrik Poll. An overview of JML tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, 2003.

[8] Margaret Burnett. Software engineering for visual programming languages. In *Handbook of Software Engineering and Knowledge Engineering*, volume 2, 2001.

[9] S. K. Chang, Margaret M. Burnett, Stefano Levialdi, Kim Marriott, Joseph J. Pfeiffer, and Steven L. Tanimoto. The future of visual languages. In *IEEE Symposium on Visual Languages*, 1999.

[10] International Engineering Consortium. Specification and description language (SDL). http://www.iec.org/online/tutorials/acrobat/sdl.pdf, (05.2004).

[11] G. Costagliola, A. Delucia, S.Orefice, and G. Polese. A classification framework to support the design of visual languages. *Journal of Visual languages and computing*, Vol. 13:573–600, 2002.

[12] Bruce Eckel. *Thinking in Java.* Prentice Hall, 2000.

[13] Martin Erwig. Functional programming with graphs. In *2nd ACM SIGPLAN Int. Conf. on Functional Programming*, 1997.

[14] Martin Erwig. Abstract syntax and semantics of visual languages. *Journal of visual languages and programming*, Vol. 9:461–483, 1998.

[15] Amy Fowler. A Swing architecture overview. http://java.sun.com/products/jfc/tsc/articles/architecture/, (05.2004).

[16] Martin Glinz. Problems and deficiencies of UML as a requirements specification language. In *IEEE Tenth International Workshop on Software Specification and Design*, 2000.

[17] Martin Glinz, Stefan Berner, Stefan Joos, Johannes Ryser, Nancy Schett, Reto Schmid, and Yong Xia. The adora approach to object-oriented modeling of software. *13th International Conference on Advanced Information Systems Engineering*, 2001.

[18] E. J. Golin and S. P. Reiss. The specification of visual language syntax. In *Proc. 1989 IEEE Workshop on Visual Languages Rome*, 1989.

[19] Pavel Grigorenko. *Program Synthesis in Java environment.* Diploma thesis, Tallinna Tehnikaülikool, 2004.

[20] Object Management Group. Introduction to OMG's unified modeling language. http://www.omg.org/gettingstarted/what_is_uml.htm, (05.2004).

[21] National Instruments. LabView overview. http://www.tinkersguild.com/sample/SponsorAds/NatInstruments/labview.htm, (05.2004).

[22] Michael Jackson. *Software Requirements and Specifications : A Lexicon of Practice, Principles, and Prejudices.* Addison-Wesley, 1995.

[23] Jorn W. Janneck and Robert Esser. A predicate-based approach to defining visual language syntax. In *Symposia on Human-Centric Computing, IEEE Computer Society*, 2001.

[24] James Gosling Ken Arnold. *The Java programming language.* Addison-Wesley, 1997.

[25] Sven Lämmermann. *Automated composition of Java software.* Lic. thesis, KTH, Stockholm, 2000.

[26] Sven Lämmermann. *Runtime service composition via logic-based program synthesis.* PhD thesis, KTH, Stockholm, 2002.

[27] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR #98-06v, Department of Computer Science, Iowa State University, 2003.

[28] Philippe Leblanc and Ileana Ober. Comparative case study in SDL and UML. In *33rd International Conference on Technology of Object-Oriented Languages and Systems*, 2000.

[29] Michael R. Lowry, Andrew Philpot, Thomas Pressburger, and Ian Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *Proceedings of the 8th International Symposium Methodologies for Intelligent Systems*, 1994.

[30] Qusay H. Mahmoud. Programming with the new language features in J2SE 1.5. http://java.sun.com/developer/technicalArticles/releases/j2se15langfeat/, (05.2004).

[31] Kim Marriott. The future of visual languages: Visual language theory. In *IEEE Symposium on Visual Languages*, 1999.

[32] Kim Marriott, Bernd Meyer, and Kent B. Wittenburg. A survey of visual language specification and recognition. In K. Marriott and B. Meyer, editors, *Theory of Visual Languages*. Springer-Verlag, 1998.

[33] Mihhail Matskin and Henryk Jan Komorowski. Partial structural synthesis of programs. *Fundamenta Informaticae*, Vol. 31(2):125–144, 1997.

[34] Mihhail Matskin and Enn Tyugu. Strategies of structural synthesis of programs and its extensions. *Computing and Informatics*, Vol. 20:1 – 25, 2001.

[35] David Megginson. SAX - The Simple API for XML. Presentation at XML Developers' Day, Seattle, 1998.

[36] MetaCase. Domain-specific modeling: 10 times faster than UML. http://www.metacase.com/papers/Domain-specific_modeling_10X_faster_than_UML.pdf, (05.2004).

[37] Grigori Mints and Enn Tyugu. Justifications of the structural synthesis of programs. *Science of Computer Programming*, 2(3):215–240, 1982.

[38] Liam Quin. Extensible Markup Language (XML). http://www.w3.org/XML/, (05.2004).

[39] Ian Salter. A framework for formally defining the syntax of visual languages. In *Proceedings of the 1993 IEEE Workshop on Visual Languages*, 1993.

[40] Sven-Bodo Scholz. Single assignment C — functional programming using imperative style. In *Proceedings of the 6th International Workshop on Implementation of Functional Languages*, 1994.

[41] C. M. Sperberg-McQueen and H. Thompson. XML Schema. http://www.w3.org/XML/Schema, (05.2004).

[42] Tanel Tammet. First order correctness proofs for propositional logic programming. Technical Report TR CS14/90, Institute of Cybernetics, Tallinn, 1999.

[43] Wolfram Technologies. Java toolkit: J/Link. http://www.wolfram.com/solutions/mathlink/jlink/, (05.2004).

[44] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+: Defining and using domain-specific modeling languages and code generators. In *OOPSLA 2003 demonstration*, 2003.

[45] Antti-Pekka Tuovinen. VILPERT - visual language expert. In *Proceedings of the Sixth Fenno-Ugric Symposium on Software Technology FUSST'99*, 1999.

[46] Antti-Pekka Tuovinen. *Object-oriented engineering of visual languages.* PhD thesis, University of Helsinki, 2002.

[47] Enn Tyugu. Using classes as specifications for automatic construction of programs in the NUT system. *Journal of Automated Software Engineering*, Vol. 1:315 – 334, 1994.

[48] Enn Tyugu and Mait Harf. Algorithms of structured synthesis of programs. *Programming and computer software*, 6:165–175, 1980.

[49] Enn Tyugu and Ando Saabas. Problems of visual specification languages. In *Proc 35th International Conference on IT + SE*, 2003.

[50] Enn Tyugu and Rando Valt. Visual programming in NUT. *Journal of visual languages and programming*, Vol. 8:523 – 544, 1997.

[51] Tarmo Uustalu. *Aspects of Structural Synthesis of Programs.* Lic. thesis, KTH, Stockholm, 1995.

[52] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices 35*, 2000.

[53] Kent B. Wittenburg. Predictive parsing for unordered relational languages. In Harry Bunt and Masaru Tomita, editors, *Recent advances in parsing technology.* Kluwer academic publishers, 1996.

[54] Zheying Zhang. Defining components in a metacase environment. In *Conference on Advanced Information Systems Engineering*, 2000.