TALLINN UNIVERSITY OF TECHNOLOGY
Facility of Information Technology
Department of Computer Science
Chair of Theoretical Informatics

Pavel Grigorenko

PROGRAM SYNTHESIS IN JAVA ENVIRONMENT

Thesis for degree of Bachelor of Science

Supervisor: Enn Tõugu

Tallinn 2004

Abstract

Automatic program synthesis module has been developed in the present work. It is a part of a framework for automated composition of Java programs using a visual specification language. The synthesizer module, uses a deductive program synthesis method, namely Structural Synthesis of Programs (SSP). The framework introduces metainterfaces as logical specifications of Java classes.

Metainterfaces contain variables and axioms of the class which denote the logical relations between the components of the specification. Metainterfaces are transformed into logical specification in the form of a graph. Synthesis is based on usage of this graph. The generated Java source code can be compiled at runtime and loaded into the framework.

Table of Contents

List of Figures

1 Introduction

The present work includes two parts – theoretical part about program synthesis and experimental part resulting in a program synthesizer. This work uses results of previous research by Lammermann [14], [15] and Saabas [1] on Structural Synthesis of Programs and its application in Java programming language. Completely new result is a new Java synthesizer with subtasks.

1.1 Motivation and goal

Over the past few years, Java programming language has become very popular. Our ambition is to develop a framework for automated composition of Java programs using visual specification-language and automatic deductive program synthesis. Deductive program synthesis uses logical specifications that represent relations between given inputs and outputs. We introduce *metainterfaces* as logical specifications which expand the capabilities of Java classes.

The goal of the present work was to design and implement an algorithm for planning with subtasks, to describe the whole synthesizer's architecture and to perform an experiment of synthesis with subtasks.

1.2 Background

Deductive program synthesis has been studied for many years and has been used in practice since '70s in several programming tools of the PRIZ family. The NUT [7] system was designed as object-oriented environment with features of automatic

program synthesis for constructing programs from the specifications and preprogrammed modules.

At present time, the research group in Institute of Cybernetics develops a framework for automated composition of Java programs using visual specification-language and deductive program synthesis. The framework can be widely applied to the different kinds of design and composition problems, for instance, composition of web services or software/hardware codesign.

1.3 Method

To describe program synthesis, formal methods have been used in the theoretical part of the work.

In the experimental part – software development and experimental validation of program synthesis for Java are the methods used.

2 Program Synthesis

2.1 Different approaches

Program synthesis is a method of software engineering used to generate programs automatically. Or another definition can be – program synthesis is the task of formally deriving programs from specifications. There are three different approaches to program synthesis:

- **Transformational synthesis**
  In transformational program synthesis given logical specification has to be rewritten until it is possible to execute it as a constructed program [19].

```
                rewriting
Specification --------------> program
```

- **Inductive Synthesis**
  Inductive program synthesis constructs programs from input/output examples [20].

  From the example

  ```
  "1 gives 1, 2 gives 4, 3 gives 9, etc. "
  ```

  it is possible to derive an arithmetic program:

  ```
  y = x * x
  ```

where `x` is input and `y` is output of the program. If this program is incorrect, then one can say that the specification is wrong, or at least incomplete.

- **Deductive synthesis**

  To derive a program, deductive approach uses a logical specification that represents a relation between a given input and the desired output. A specification can be stated as a theorem that asserts the existence of an output that meets the specified conditions. This theorem should be proved automatically. The generated proof becomes the basis for a program that can be extracted from the proof. The key ideas of this approach are presented in [18].

One of the methods of the deductive program synthesis is more deeply described in the next section.

2.2 Structural Synthesis of Programs

A formal foundation of Structural synthesis of programs (SSP) was developed by Mints and Tyugu [8]. SSP uses the deductive program synthesis approach. The idea of SSP is that programs can be constructed using their structural properties. The SSP uses intuitionistic propositional calculus (IPC).

2.2.1 Intuitionistic Propositional Calculus

In order to guarantee constructiveness of proofs, the SSP uses intuitionistic logic. It is sufficient to use introduction and elimination rules for implication and conjunction, because these are the only connectives of the logical language of SSP (see 2.2.2). In this case we have to accept conjunctions of propositional variables and of simple implications as formulae of our logic as well. These conjunctions will appear in the process of derivation. The inference rules of the Intuitionistic propositional calculus for implications, conjunctions and disjunctions are the following.

Conjunction:

$$\frac{A \qquad B}{A \wedge B}(\wedge+) \qquad\qquad \frac{A \wedge B}{A}\ \frac{A \wedge B}{B}(\wedge-)$$

Introduction Elimination

Disjunction:

$$\frac{A}{A \vee B}(\vee+) \qquad\qquad \frac{\begin{array}{cc} & A \quad B \\ A \vee B & \vdots \quad \vdots \\ & C \quad C \end{array}}{C}(\vee-)$$

Introduction Elimination

Implication:

$$\frac{\begin{array}{c} A \\ \vdots \\ B \end{array}}{A \supset B}(\supset +) \qquad\qquad \frac{A \qquad A \supset B}{B}(\supset -)$$

Introduction Elimination

Figure 2.1. IPC inference rules.

Any formula of IPC has a computational realization with a structure corresponding to the structure of the formula.

Program extraction occurs step by step in the following way. Realizations of axioms are given. Let *a* be the realization of a formula *A* and *b* the realization of a formula *B*. Then $A \wedge B$ has a realization *(a, b)*. $A \vee B$ has a realization *if "first" then a else b*. The realization of $A \supset B$ is a function *f(x)* that computes *b* from *a*.

Realizations of derived formulas are built as follows:

Conjunction:

$$\frac{a : A \qquad b : B}{(a,b) \quad A \wedge B} \qquad \frac{(a,b)\, A \wedge B}{a : A} \qquad \frac{(a,b)\, A \wedge B}{b : B}$$

Introduction Elimination

Disjunction:

$$\frac{a : A}{\textit{if } A \textit{ then } a \textit{ else } b : \; A \vee B} \qquad \frac{\begin{array}{ccc} & A & B \\ A \vee B & \vdots & \vdots \\ & f : C & g : C \end{array}}{\textit{if } A \textit{ then } f(a) \textit{ else } g(a) : C}$$

Introduction Elimination

Implication:

$$\frac{\begin{array}{c} x : A \\ \vdots \\ b : B \end{array}}{b(x) : \; A \supset B} \qquad \frac{a : A \qquad b(x) : A \supset B}{b(a) : B}$$

Introduction Elimination

Figure 2.2. IPC inference rules with realizations.

Calculus that is based on conventional inference rules of IPC as presented above is practically unsuitable for program synthesis for the following two reasons. First, the extracted program will contain unnecessary steps of composition and decomposition of data structures that correspond to the realizations of conjunction appearing in a derivation. Second, the derivation steps are small and it is difficult to perform a goal-oriented search using the conventional inference rules. In the next section we introduce new inference rules and describe a calculus immediately suitable for program synthesis.

2.2.2 Language and Inference Rules of SSP

SSP uses implicative fragment of IPC (without disjunctions) and only restricted nestedness of formulas [11], [12].
The Logical language of SSP has the following kinds of formulae:

1. *Propositional variables* match the objects from the specification of the problem. They are denoted by identifiers with capital letters, for example, A, B, C…
   If there is an object variable *a* in the specification of a problem, then we denote the corresponding propositional variable in the logical language by *A*. It expresses computability of a value of the variable *a*.
2. *Unconditional computability statements* express the computability of the object variable corresponding to the right hand side of the statement from values which correspond to the propositional variable on the left hand side of the statement. $A_1 \wedge \ldots \wedge A_k \supset$ B.
3. *Conditional computability statements,* for example, $(A \supset B) \wedge C \supset D$ express computability of the object variable d from c depending on the computation of the object variable b from a.

SSP uses admissible rules for deriving new formulas instead of using the conventional rules of the Intuitionistic propositional calculus. Any admissible elimination rule represents a fragment of derivation, and corresponds precisely to application of one program method. This guarantees efficient proof search.
Let D, E, F, G, H be conjunctions of propositional variables. Then the admissible inference rules of SSP are:

- Implication Elimination

$$\frac{D \supset E \qquad E \wedge G \supset H}{D \wedge G \supset E \wedge H}(\supset -)$$

- Double Implication Elimination

$$\frac{(D \supset E) \wedge F \supset G \qquad D \wedge H \supset E}{H \wedge F \supset G}(\supset --)$$

- Weakening

$$\frac{D \supset E \wedge F}{G \wedge D \supset E}(\wedge + -)$$

Figure 2.3. SSP admissible inference rules.

2.2.3 Extended Structural Synthesis of Programs

Later on we introduce disjunctions on the right side of axioms as well. Formally speaking, this requires rules for disjunction. In fact, these disjunctions can be eliminated by substituting them with subtasks [14], [16] therefore we do not show the rules for disjunctions here.

2.2.4 Proof Search Algorithm

The proof search algorithm used in SSP was first described in [3]. The proof search strategy in SSP 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 reach the goal, 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.

The proof search algorithm is extended, when disjunctions are allowed in the logical language. The extended proof search algorithm rewrites logical specifications containing the disjunction connective dynamically. 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.

Example. A search tree of the Minimax problem (see Chapter 5) is shown in Figure 2.4. Each solid arrow represents a step of the goal-driven backward search, i.e. subtask selection. Nodes represent assumption-driven search, i.e. computability statement selection. The dotted arrow indicates backtracking.

The axioms are:

$$(row \supset max) \supset min$$

$$(column \supset element) \supset max$$

$$row \wedge column \supset element$$

and the goal is:

$$min$$

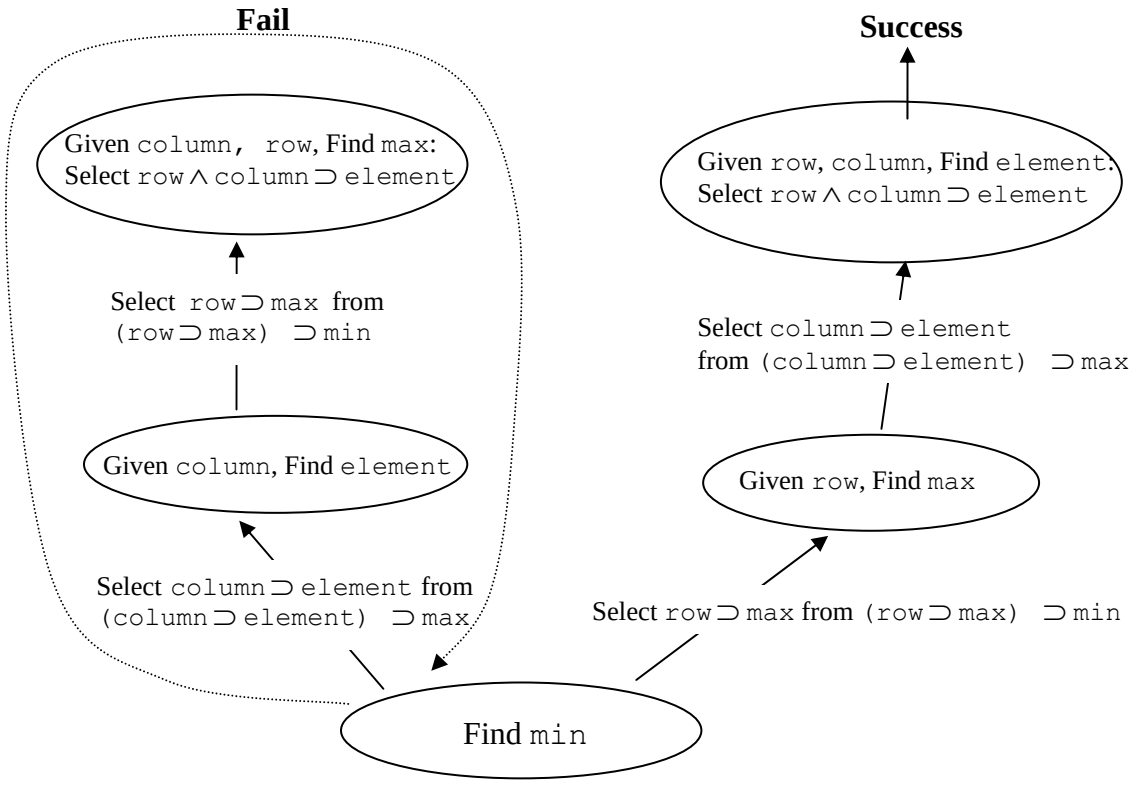


Figure 2.4. Search Tree of Minimax

The proof search algorithm of SSP can be implemented with a data structure that represents a set of computability statements describing problem conditions [5].

In [21] Tyugu explains this data structure as higher-order dataflow schema (HODS). The nodes of HODS are either *control* or *functional nodes*. Functional nodes receive subtasks as input. Control nodes exercise control over the execution order of their subtasks and perform computation as well. Every propositional variable is an ordinary node and computability statement can be either a functional or control node. Each subtask is also a node, which connects as input to the respective computability statement and behaves like ordinary propositional variable.

Figure 2.5 shows the data-flow relation of the Minimax specification (see section 5.2). This is an abstract visualization of the internal representation of the specification in the form of a graph. Each computability statement is denoted as a relation (name starts from an uppercase letter), subtask relations are S_1 and S_2. Propositional variables are represented by their names starting from lowercase letters.

$S_1 \supset$ max	Max
$S_2 \supset$ min	Min
row $\wedge$ column $\supset$ element	Matrix
column $\supset$ element	S_1
row $\supset$ max	S_2

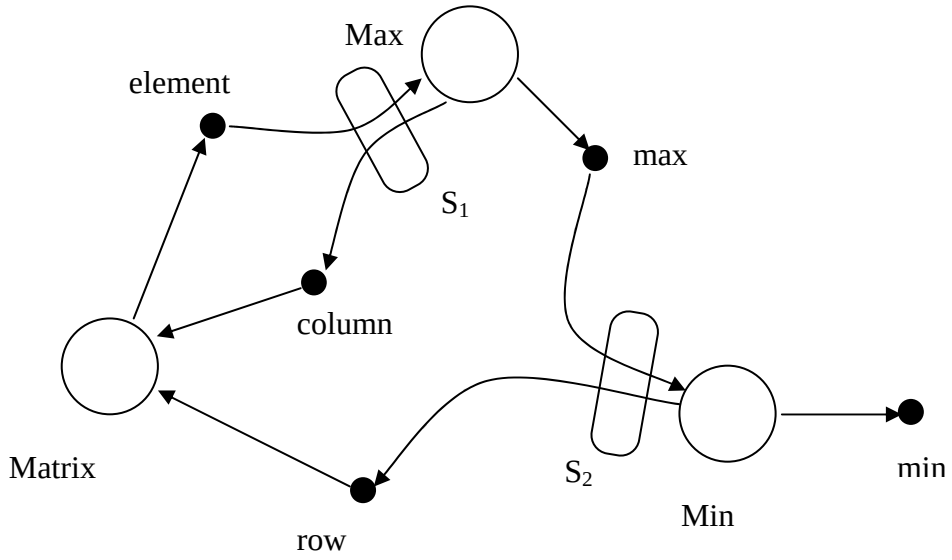


Figure 2.5. Higher-order dataflow schema for Minimax

3 System description

3.1 Architecture

The synthesis system is a framework for automated composition of Java programs using visual specification-language and automatic deductive program synthesis [1]. The idea was first implemented in the NUT system [4]. The framework includes several independent modules. The three top-level modules are *Visual Scheme Editor*, *Equation solver* and *Synthesizer*.

To design a problem, user can draw a scheme which will be translated into specification language and later will be converted to the logical specification. Synthesizer will use that specification to plan an algorithm and generate Java source code which can be compiled and launched at run-time.

3.2 Specification Language

The specification language was introduced by Saabas, Tyugu [2]. It allows declaring variables, axioms, bindings and equations in so called *metainterfaces.* Metainterface is a logical specification of a Java class. In addition, term Metaclass can be used to describe a complete Java class that contains metainterface and some method realizations.

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

The *SpecificationName* is the name of the specification being declared.

3.2.1 Variables

Variable declaration follows the Java syntax.

VariableDeclaration ::= Type IdentList';'
IdentList ::= Identifier [',' IdentList]

Identifier is the name of the declared variable in the interface. Type can be:

- a primitive type of the Java programming language or 'void'
- a class in the Java programming language
- a specification already described in the package

Example.

```
specification Wheel {
     int radius;
     String description;
}
```

3.2.2 Constants

The declaration of a constant has the form:

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

Where *value* refers to a constant value in the Java programming language.

Example.

```
String[] str;
double pi;
str := {"str1", "str2"};
pi := Math.PI;
```

3.2.3 Bindings

Binding is a logical equality used to show that the realization of the left component is the same as the realization of the right component. The specification of the binding has the form

Binding ::= Variable '=' Variable

Where Variable can be in the form of

Variable ::= Identifier | Variable'.'Identifier

This definition introduces hierarchical structure of variables and compound names. The specification $a.x = b.y$ means that the realization of the variable $a.x$ is the same as the realization of the variable $b.y$. In our logical language relations $x = y$ and $y = x$ are semantically equivalent.

3.2.4 Axioms

Axiom specifies a possible computation of a component. It is written in the form of implication. Preconditions are listed in the left side of the implication, and postconditions of an axiom are in the right side of the implication. The realization of the axiom (Java method) is specified in the curly brackets. Postcondition shows the variable

whose value is returned from the realization. The axiom states that the realization can be applied if all of its preconditions are derivable.

```
double radius;
double area;
radius -> area {calcArea};

...

double calcArea(double r) {
      return Math.PI*Math.pow(r, 2);
}
```

The specification above states that the *area* can be computed in the specified class using *radius* as the input parameter of the method *calcArea*, which represents the realization of the given axiom.

The precondition of an axiom can be a variable or a subtask (see 3.2.5).
The postcondition of an axiom includes a variable. In addition, postcondition may include one or more control variables (see 3.2.6). The logical meaning is that when an axiom is applied, all variables in preconditions have their values and variables in postconditions derivable. Disjunction '|' symbol is used for separating possible alternative outputs of the applied method, for instance, exceptions.

Axiom ::= ImplementedAxiom | UnimplementedAxiom
ImplementedAxiom ::= [VariableList [',' SubTaskList]] '->'
VariableList ['|' VariableList] '{'Realization'}'
UnimplementedAxiom ::= [VariableList] '->' VariableList
VariableList ::= Variable [',' VariableList]

Unimplemented axiom has no realization and denotes a goal on the right-hand side of the implication. In other words, variables that appear in the postcondition of the axiom must be computed to solve the given problem.

3.2.5 Subtasks

Precondition in the form of an implication denotes a subgoal in an (implemented) axiom. It is called a **subtask.** Interface variable on the left-hand side from the implication shows the given value and on the right-hand side the value to be computed by a piece of synthesized program that is a realization of the subtask. More precisely, realization of a subtask is a synthesized class implementing interface *Subtask* with a synthesized method *run*().

```
SubtaskList ::= Subtask [',' SubtaskList]
Subtask ::= '['IdentifierList '->' Identifer ']'
```

Subtasks are discussed in greater detail in chapters 4 and 5.

3.2.6 Control variables

Control variables do not have a computational meaning, though if a control variable is used as one of the preconditions in the axiom, it should be derivable. In specification, declared control variable should be of *void* type.

Example.

```
void ready;
String path;
path -> ready {Init};        (1)
ready -> {showStatus};       (2)
```

The purpose of the given example was to show how the control variable allows to control methods' execution priority. Method *showStatus* should be executed after the method *Init*, which returns void. If the axiom (1) had no postcondition, this would lead to a problem where we would be unable to guide the synthesis process. Control variables help solving this problem. When the axiom (1) is applied, *ready* becomes

derivable. Only then it is possible to use *ready* as a precondition of (2) to apply the axiom.

3.3 Extensions

The specification language is extended with several syntax structures. This “syntactic sugar” simplifies the development of the specifications and makes specifications easier to read. Each extension can be extracted into the core-language syntax. In fact that is what takes place while parsing a specification.

3.3.1 Aliases

```
Alias ::= "alias" Identifier "=" "(" VariableList")"
```

Example.

```
specification Point {
int x, y;
alias coordinates = (x, y);
}
```

Assume that we have two instances of Point, A and B. If the following binding is applied:

```
A.coordinates = B.coordinates,
```

it is unfolded to

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

3.3.2 Equations

Equations are used to get rid of excess axioms in the specifications. Equations can be declared using the following grammar:

```
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"
```

Example.

```
r -> c {calcLength}
c -> r {calcRadius}
```

The axioms above, used to calculate the radius and the circumference, can be folded to a single equation

```
c = 2 * pi * r;
```

If the goal is to calculate the radius, “equation solver” will automatically extract the required equation

```
r = c / (2 * pi);
```

4 Application to Java

The following section describes only one module of the whole framework – the Synthesizer, which deals with program synthesis and automatic composition of Java programs.

4.1 Synthesizer

The synthesizer does the following steps:

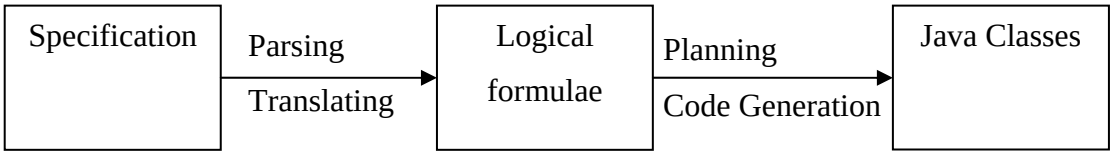


Figure 4.1. Structure of Synthesizer

The first step is to parse the specification. We assume that synthesizer is called when specification already exists. It does not matter how synthesizer gets the specification, from the visual editor or command line, etc. The specification is parsed recursively, unfolding all required items. Recursion is needed in case the main specification contains the instances of classes also containing specifications.

The next step is translating the unfolded specification into logical language. Internal representation of the specification in the logical language has the form of a graph where each variable and formula is a node. Thereafter the planner takes control over the synthesizing process. The class *Planner* will be described in the section 4.3.

The final step is generating Java code based on the algorithm returned by the Planner. The Code Generator will be discussed in section 4.4.

The implementation of the Synthesizer in our framework project is the Java package *synthesize.* It includes all required classes used in the synthesis process. The main classes are shown in Figure 4.2.

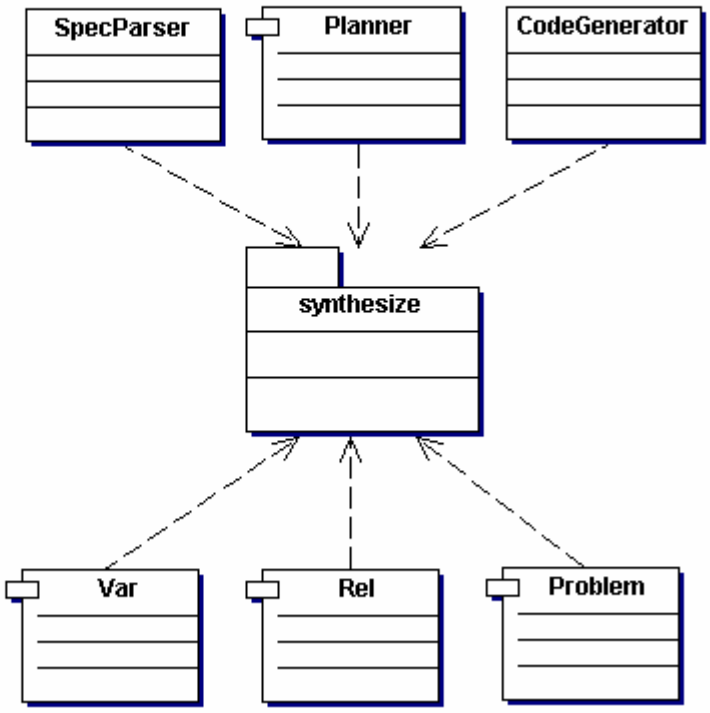
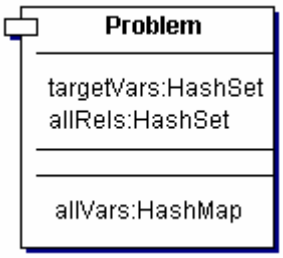


Figure 4.2. Main classes of the package *synthesize*

4.2 Specification parser and translator

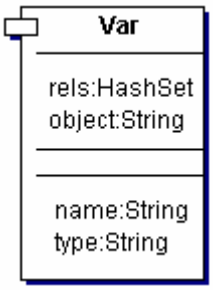
The specification can be given using Visual Editor or typed manually.

Class *SpecParser* consists of several specific methods for parsing the specification and translating it into logical representation. This representation is presented as sets of object of classes *Var* (variables) and *Rel* (axioms). It is implemented in the class *Problem*.



Problem is a data structure representing a graph on which planning is applied. It contains sets of objects used for planning. These objects are:

- Class *Var* encapsulates any variable declared in the specification. It can be a control variable, an object instance, primitive type variable etc.



Main members of the class *Var* are:
HashSet *rels* contains all axioms encapsulated in the classes *Rel* which includes the *Var* instance.
String *object* contains object name to which the variable belongs.
String *name* shows the name of the variable.
String *type* contains the type of the variable.

Example.

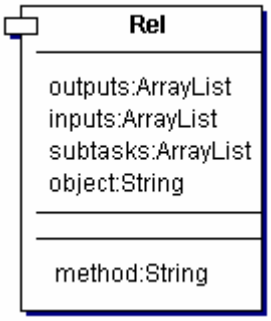
```
specification Wheel {
     double radius;
}
```

After parsing the given specification and during translating process new *Var* instance is created where class members get the following values:

```
name = "radius";
type = "double";
object = "this";
```

If the variable is declared in the body of the specification and it does not belong to any other object, value "this" is assigned to member variable *object* of the class *Var*.

- Class *Rel* is the structure for encapsulating axioms from the specification.



Members of the class *Rel* represent values:
ArrayList *inputs* contains a list of *Var* instances which belong to the precondition of an axiom, except subtasks.
ArrayList *outputs* contains a list of *Var* instances which belong on the postcondition of an axiom.
ArrayList *subtasks* is a set of *Rels* representing subtasks which appear in the axiom.
String *method* is a method name that appears in the realization.
String *object* shows an object instance name to which an axiom belongs. If an axiom is declared in the main specification, *object* value is "this".

Example.

```
specification Wheel {
      double radius;
      double area;
```

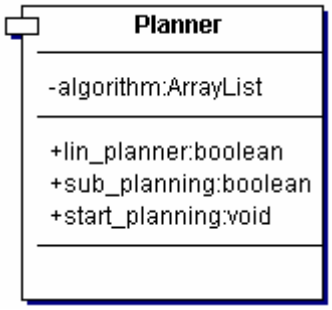
```
      radius -> area {calcArea};
}
```

New *Rel* instance will have the following values:

```
inputs = { Var(radius) };
outputs = { Var(area) };
method = "calcArea";
object = "this";
```

4.3 Planner

After the translating stage is completed and object *Problem* is created, this graph structure is passed to the class *Planner*. The purpose of *Planner* is to find a proof for the specified problem and generate a correct algorithm. The planner has been developed together with Ando Saabas [1]. The planning algorithm itself was first described in [3]. When an instance of *Planner* is created, members of an object *Problem* are unfolded to different sets: target variable list, usable relation list, and known variable list in its constructor.



When it comes to planning, there are three methods, which correspond to this process. We call "linear planning" a part of planning without subtasks. Much more complicated part of planning is "planning with subtasks". Two methods in the class Planner perform those types of planning, *lin_planner*() and *sub_planner*() respectively.

The third method *start_planning*() manages the process. Its algorithm is shown in the following figure:

```
start linear planning
if linear planning unsuccessful then
        start planning with subtasks
        if subtask planning succeeded then
                algorithm ready
        else if something found then
                start planning
        else problem not solved
else algorithm ready
```

Figure 4.3. Algorithm for managing synthesis process

Linear planning is implemented in method *lin_planner*(). Its algorithm tries to solve the given problem, i.e. find all target (goal) variables' realizations using axioms without subtasks.

Planning is being applied on the sets of *Rel* and *Var* objects which form a graph. Class *Rel* contains important counter variable used in linear planning algorithm. This variable, we call it "precondition counter", shows how many specification variables are left unknown on the left-hand side of an axiom. If "precondition counter" equals zero, then:

- postcondition variables become known and are added to the corresponding set;
- a new step (*Rel*) is added to the algorithm;
- new computed variables are excluded from the target variable list;
- if **target variable list** is empty then the synthesized algorithm is ready;

Linear planner algorithm:

```
for each known variable {
        if known variable is in target variable list
                remove known variable from target variable list
}
new variables derived ← true
while not target variables known and new variables derived {
        new variables derived ← false
        for each known variable {
                for each of its relation {
                        decrement the precondition counter by 1
                        if precondition counter is zero {
                                remove relation from relation list
                                if not all of relation output variables already found {
                                        add output variables of relations to known
                                variables list
                                        add relation to algorithm
                                        remove variables from target variable list
                                        new variables derived ← true
                                }
                        }
                }
                remove variable from list of known variables
        }
}
if target variable list is empty
        return true, problem solved
else
        return false, problem not solved
```

Figure 4.4. Linear planning algorithm.

After the linear planning is applied, an algorithm has to be optimized to calculate the variables that are targets. The package *synthesize* includes class *Optimizer* that has the method *optimize*() which is used in optimizing algorithms returned by the linear planner.

If linear planning does not solve a given problem, planning with subtasks starts. In this case linear planning is used as one step of planning. Current implementation of subtask planner is iterative and involves no recursion. The Planner tries to solve subtasks one by one, handling each subtask as a new problem. If no subtask is solvable independently, a new iteration of the process of solving each subtask is performed using subtasks inside solving subtask one level deeper, etc. This is a process of breadth-first search in the set of sequences of subtasks. Bringing in the recursion with backtracking is one of the steps in our future work.

```
new variables derived = true
while new variables derived {
    new variables derived ← false
    for each relation with subtasks {
        for each subtask in current relation {
            add subtask inputs to known variable list
            remove relations with outputs of subtask inputs from relation list
            for each known variable {
                for each sub_relation of known variable {
                    if sub_relation is in relation list {
                        decrease "precondition counter" by 1
                        remove known variable from known variable list
                        if "precondition flag" of sub_relation is zero {
                            if beginning of subtask
                                add <subtask> to algorithm                    (*)
                            add sub_relation to algorithm
                            add outputs of sub_relation to known variable list
                            new variables derived ← true
                            start linear planning for new derived variables
                        }
                    }
                }
                if outputs of relation with subtasks in known list {
                    add </subtask> to algorithm
                    add relation with subtasks to algorithm
                    remove relation with subtasks from relation list
                }
            }
        }
```

```
    }
}

if target variable list is empty then
        return true, problem solved
else
        return false, problem not solved
```

Figure 4.5. Algorithm for planning with subtasks.

We decided to use XML style (tag <subtask> (*)) to distinguish subtasks in a sequence of relations in the algorithm. When subtasks are involved, synthesized algorithm structure changes from a sequence of relations to a tree.

Assume that R_i is a relation and S_j is a sub-relation, i.e. relation with a subtask. Consider the following example:

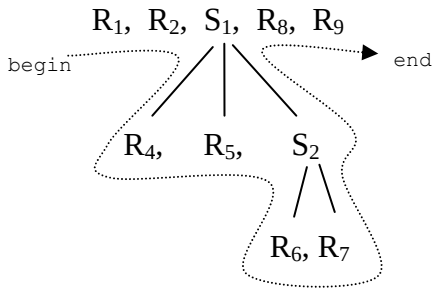


Figure 4.6. Algorithm with subtasks

Then the tree-structured algorithm can be represented as a sequence of relations and sub-relations:

Algorithm := R_1, R_2 <S> R_4, R_5 <S> R_6, R_7 </S> </S> R_8, R_9

To understand how subtasks can be used in practice, see Chapter 5.

4.4 Code Generator

Java program source code is extracted from a synthesized algorithm returned by the planner. As mentioned before, algorithm is a set of sequences of *Rel* objects that prescribe the structure of the final Java program. Code generation is a final computational step in a synthesis process.

4.4.1 Realization of classes and specification variables

Variables, declared in a specification, are not proper components of a class. If they are needed in the synthesized Java program, these variables have to be transformed to Java syntax. This transformation is the first step of the Code Generator. Classes which contain specifications are rewritten to new classes with specifications converted to Java syntax. Assume we have the following metaclass (i.e. class with specification):

```
class Matrix
{
	/*@
	specification Matrix {
		int i, j, element;
		i, j -> element{getElement};
	}
	@*/

	public int getElement(int i, int j) {
		...
	}
}
```

Metaclass Matrix is converted to Java compilable source code. To distinguish new Java class from the metaclass, its name starts and ends with an underscore '_'.

```
public class _Matrix_
{
	public int i;
	public int j;
	public int element;

	public int getElement(int i, int j) {
		...
	}
}
```

The synthesis starts from the main specification of a synthesis problem. It can be composed using Visual Editor or typed manually. By default, this specification is placed into the class *GeneratedClass*. For example:

```
public class GeneratedClass {
    /*@ specification  GeneratedClass {
       int a, b;
       a = 2;
       b = a*2;
    }@*/
}
```

The Code Generator transforms *GeneratedClass* into *_GeneratedClass_* that implements an interface *IGeneratedClass*.

```
public interface IGeneratedClass {

        public void compute();

}
```

This interface declares a method *compute*() which is necessary for the final synthesized Java program. The *compute*() method will contain synthesized code in its body, including classes that implement subtasks.

```
public class _GeneratedClass_ implements IGeneratedClass {

       public int a;
       public int b;
       public void compute() {
              a = 2;
              b = (a * 2);
       }
}
```

The *compute*() method plays almost the same role as the method *main*() in a Java primary class or *run*() in Thread class. When synthesized algorithm is transformed into the source code, it is compiled at runtime. Thereafter it is possible to run a compiled program and the system will use *compute*() for computing goal variables.

4.4.2 Exception handling

The specification language allows specifying alternative outputs in axioms using disjunction on the right-hand side of the main implication [16]. In current realization they are represented by Java exceptions that can be thrown by a method that implements an axiom. Exceptions are used to guarantee the correct continuation or application shutdown in case an error occurs. To declare an exception we use interface variables that specify classes extending `java.lang.Exception`.

```
specification Matrix {
        int i, j, element;
        java.lang.Exception excp1;
        i, j -> element | excp1 {getElement};
}
```

If a *Rel* object that appears in synthesized algorithm corresponds to an axiom where its implementation may throw an exception, it will be surrounded with try-catch statement during code generation:

```
public void compute() {
        ...
        try {
                element = getElement(i, j);
        } catch(java.lang.Exception excp1) {
                excp1. printStackTrace();
                return;
        }
        ...
}
```

The reason of using the return statement instead of `System.exit(1)` is that we do not actually shut down the program, the goal is to finish processing of the method *compute*(). Due to the fact that synthesized program is compiled and launched within the bounds of the framework (with the aid of Java's built-in support for reflection) the usage of `System.exit(1)` would cause the shutdown of the whole application.

4.4.3 Generating subtasks

For each subtask the Code Generator creates a Java class that implements the *Subtask* interface.

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

Subtask is a synthesized class which is declared as a inner class of the method *compute*(). The advantage of inner classes is that they have a quick access to the member variables of the main class. The code generation for a subtask consists of the following steps:

- Define a subtask class as inner class of *compute*();
- Create an instance of a subtask class;
- Generate a method call with a subtask instance as parameter;

Example.

From the given specification

```
public class GeneratedClass {
	/*@ specification  GeneratedClass {
		int a, b, c;
		...
		[a -> b] -> c {calc};
		...
	}@*/

	public int calc(Subtask sub) {
		...
	}

}
```

Planner returns the algorithm:

```
...
<subtask>
...
```

```
</subtask>
c  = calc(subtask)
...
```

and the source code below will be generated. The comments “Step 1”, “Step 2” and “Step 3” represent the code generation steps described above.

```
public class _GeneratedClass_ implements IGeneratedClass {
	public int a;
	public int b;
	public int c;
	...
	public void compute() {
		...
		//Step 1.
		class Subtask_1 implements Subtask {

			public Object[] run(Object[] in)
						throws Exception {
				x = ((Integer)in[0]).intValue();	(*)
				...
				Object[] out = new Object[1];
				...
				return out;
			}
		}
		//Step 2.
		Subtask_1 subtask_1 = new Subtask_1();

		//Step 3.
		c = calc(subtask_1);

	}
	public int calc(Subtask subtask) {
		try {
			Object[] in = new Object[1];
			in[0] = new Integer(a);
			Object[] out = subtask.run(in);
			b = ((Integer)out[0]).intValue();		(*)
			...
		} catch(Exception ex) {
			... //developer's code
		}
	}
}
```

The method *run*() in a class that implements Subtask uses Object array as input and output. The direct casting of a primitive type variable to Object type is forbidden in Java. This is why primitive type variables have to be encapsulated with corresponding Objects. To generate the correct code, we use specific methods to convert *int* to *Integer*, *double* to *Double*, etc. See (*) in the example above.

Sometimes (e.g. in the Minimax problem), subtasks have their own inner subtasks. In these cases, the Code Generator uses recursive algorithm to generate source code for the inner subtasks.

5 Experiments

The whole process of problem solving from the specification to the code generation, Java compilation and computations will be illustrated here.

5.1 The Minimax problem

Minimax example is a problem of finding the element in a matrix which has the minimal value among all maximal elements of rows [11], [12]. Minimax enables to test almost all aspects of the program synthesis, e.g. linear planning and planning with subtasks. Program components for solving the problem are described within three classes: *Matrix*, *Max and Min*. The program for computation of *Minimax* is synthesized automatically.

```
specification Matrix {
      int row, col, element;
      row, col -> element {getElement};
}
```

This is a simple representation of the class *Matrix*. Elements of the matrix are organized in rows and columns. In the current description the representation of the matrix is not considered. It can be two-dimensional array, file or any other. Assume that the method *getElement()* takes care about proper representation and returns a correct value based on the given parameters: a row and a column number.

```
specification Max {
            int arg, val, maxval;
            [arg -> val] -> maxval {getMaxVal};
}
```

The class *Max* has the method *getMaxVal()* for computing the maximum value of a function computed by an implementation of the subtask *arg -> val*.

```
specification Min {
            int arg, val, minval;
            [arg -> val] -> minval {getMinVal};
}
```

The class *Min* has the method *getMinVal()* for computing the minimum value of a function computed by an implementation of the subtask by *arg -> val*.

5.2 The realization of Minimax

In our realization of the Minimax problem, the matrix is given as a two-dimensional array.

Specifications of the classes involved in Minimax problem were extended for smooth realization purposes. The initial (i.e. logically required) specification is shown in bold font.

```
class Matrix {
      /*@
      specification Matrix {
            int row, col, element;
            void m_ready;
            String[] tm;
            java.lang.Exception excpt1;
            tm -> m_ready | excpt1 {parseMatrix};
            row, col, m_ready -> element | excpt1 {getElement};
      }
      @*/
      public int[][] matrix;

      public int getElement(int i, int j)
            throws ArrayIndexOutOfBoundsException    {
            return matrix[i][j];
      }
      ...
}
```

Figure 5.1. Specification of class Matrix

Control variable *m_ready* denotes that the method *getElement()* can be invoked only after the matrix is initiated using the method *parseMatrix()*.

The method *getElement()* may return a matrix element or throw an exception. The exception occurs when given parameters, row or column exceed bounds of an array. The purpose is that we do not have to show explicitly the exact dimensions of a matrix; as a result fewer variables are used, what makes the development easier.

Figure 5.2 represents the extended specification of Max and realization of method *getMaxVal*().

```
class Max {
	/*@
	specification Max {
		int arg, val, maxval;
		[arg -> val] -> maxval {getMaxVal};

	}
	@*/
	public int getMaxVal(Subtask sbt) throws Exception {
		boolean ready = true;
		arg = 0;
		try {
			Object[] in = new Object[1];
			in[0] = new Integer(arg);
			Object[] out = sbt.run(in);

			maxval = ((Integer)out[0]).intValue();
			for(arg = 1; true; arg++) {
				ready = false;
				in[0] = new Integer(arg);
				out = sbt.run(in);
				val = ((Integer)out[0]).intValue();
				if(maxval < val) {
					maxval = val;
				}
		}
		} catch (Exception ex) {
			if(ready)
				throw new Exception();
			return maxval;
		}
	}
}
```

Figure 5.2. Specification of class Max

Figure 5.3 shows specification of the class Min. Realization of *getMinVal*() is similar to *getMaxVal*() from Max.

```
class Min {
	/*@
	specification Min {
		int arg, val, minval;
		java.lang.Exception excpt1;
		[arg -> val] -> minval | excpt1 {getMinVal};

	}
	@*/
	...
}
```

Figure 5.3. Specification of class Min

Methods *getMaxVal*() and *getMinVal*() are totally hand-coded because they are realizations of the main axioms of metaclasses Max and Min. They both use subtasks as input parameters and can throw *Exception*.

The metaclass MinMax specified by means of a scheme defines the problem in the following form:

```
specification MinMax {
      Matrix m;
      Min min;
      Max max;
      max.arg = m.col;
      min.arg = m.row;
      max.val = m.element;
      min.val = max.maxval;
      -> min.minval;
}
```

Unimplemented axiom

```
-> min.minval;
```

denotes the goal, i.e. the minimal element to be computed.

Synthesized Java program that finds the element in a matrix which has the minimal value among all maximal elements of rows is introduced in Figure 5.4. We do not show here unfolded metainterfaces, but concentrate on the generated source code with a focus on subtasks. The goal `min.minval` is computed by applying the method *getMinVal*() to which the `Subtask_1` is passed as a parameter. It holds the row number of the matrix which is used in inner `Subtask_2`. `Subtask_2` is passed as a parameter to the method *getMaxVal*(). It holds the column number of the matrix and uses row number from `Subtask_1` to get the matrix element from the method *getElement*().

```
public class MinMax implements IGeneratedClass
{
    public _Matrix_ m = new _Matrix_();
    public _Min_ min = new _Min_();
    public _Max_ max = new _Max_();

    public void compute()
    {
        ...
        class Subtask_1
            implements Subtask
        {
            public Object[] run(Object[] in) throws Exception
            {
                m.row = ((Integer)in[0]).intValue();

                class Subtask_2
                    implements Subtask
                {
                    public Object[] run(Object[] in) throws Exception
                    {
                        m.col = ((Integer)in[0]).intValue();
                        m.element = m.getElement(m.i, m.j);
                        Object[] out = new Object[1];
                        out[0] = new Integer(m.element);
                        return out;
                    }
                }

                Subtask_2 subtask_2 = new Subtask_2();

                max.maxval = max.getMaxVal(subtask_2);
                Object[] out = new Object[1];
                out[0] = new Integer(max.maxval);
                return out;
            }
        }

        Subtask_1 subtask_1 = new Subtask_1();

        try
        {
            min.minval = min.getMinVal(subtask_1);
        }
        catch (Exception excp1)
        {
            excp1.printStackTrace();
            return;
        }
    }
}
```

Figure 5.4. Synthesized program Minimax

6 Conclusion

In this thesis we have presented a work on developing a framework for automatic composition of Java programs using deductive program synthesis. To be more precise, this paper is related only to one module of the framework that was described in detail - the *synthesizer*. It is a top-level module of the framework which uses Structural Synthesis of Program (SSP).

SSP uses admissible rules for deriving new formulas instead of using the conventional rules of the Intuitionistic Propositional Calculus. Any admissible elimination rule represents a fragment of derivation, and corresponds precisely to application of one program method. This guarantees efficient proof search. The theoretical part of the work was discussed in Chapter 2.

In Chapter 3 a brief overview of the framework's architecture was given. We have introduced *metainterfaces* as logical specifications of Java classes. Metainterfaces contain variables and axioms of the class which denote the logical relations between the components of the specification. To describe relations of the components, *specification language* is used. It was illustrated in the given chapter as well.

Chapter 4 is totally dedicated to one single module of the framework - the *Synthesizer*. It contains several sub-modules which were discussed in different sections. The first module is used for parsing the problem's specification and translating it into logical representation. After this stage is completed, the second module, the Planner, tries to find a proof for the specified problem and generate a correct algorithm. A problem might contain subtasks, in that case the Planner uses algorithm for planning with subtasks. Implementing an algorithm for planning with subtasks was one of the goals of

the work. Code generation is a final computational step in a synthesis process. Code Generator extracts Java source code from a synthesized algorithm returned by the Planner. Code Generator's prototype for generating subtasks was implemented as well.

The whole process of problem solving from the specification to the Java code generation is illustrated in Chapter 5 using the example Minimax. Minimax is a problem of finding the element in a matrix which has the minimal value among all maximal elements of rows. It allowed us to test most important aspects of the program synthesis, i.e. linear planning and planning with subtasks.

The results of the work are the following:

- Implemented Planner with subtasks as a part of the synthesizer;
- Implemented Code Generator prototype for subtasks;
- Successful testing of the synthesizer using the Minimax example.

Bibliography

[1] Ando Saabas. *A Framework for Design and Implementation of Visual Languages*. M.Sc thesis, IOC, Tallinn, 2004.

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

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

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

[5] Enn Tyugu and Tarmo Uustalu. *Higher-Order Functional Constraint Networks*. Constraing Programming, pp. 116-140, 1994.

[6] Enn Tyugu. *On the border between functional programming and program synthesis.* KTH, Stockholm, 1998

[7] Enn Tyugu. *Using Classes as Specifications for Automatic Construction of Programs in the NUT System*. Automated Software Engineering, 1: 315-334, 1994.

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

[9] Jean-Yves Girard, Paul Taylor, Yves Lafont. *Proofs and Types*. Cambridge University Press, Cambridge, 1989.

[10] Mait Harf, Kristiina Kindel, Vahur Kotkas, Peep Kungas, and Enn Tyugu. *Automated Program Synthesis for Java Programming Language*. Perspectives of System Informatics. Preliminary Proc. of Andrei Ershow Fourth International Conference, July 2-6, 2001, Akademgorodok, Novosibirsk, Russia, pp. 85-87, 2001.

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

[12] Mihhail Matskin and Enn Tyugu. *Strategies of structural synthesis of programs and its extensions.* Research Report TRITA-IT/R 99:03, Dept of Teleinformatics, KTH, Stockholm, 1999.

[13] Mihhail Matskin, Jan Komorowski, John Krogstie. *Partial Deduction in the Framework of Structural Synthesis of Programs.* In Logic of Program Synthesis and Transformation (Gallagher, J., ed). LNCS 1207. Springer Verlag, Berlin, pp. 239 – 254, 1997.

[14] Sven Lammermann and Enn Tyugu. *Implementing Extended Structural Synthesis of Programs.* Spring Symposium Series on Logic-Based Program Synthesis: State of the Art and Future Trends. AAAI Press, pp. 63 – 71, April 2002.

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

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

[17] Tarmo Uustalu. *Extensions of Structural Synthesis of Programs*. In U. H. Engberg et al., eds., Proc. of 6th Nordic Wksh. on Programming Theory, NWPT'94 (Århus, Denmark, 17-19 Oct 1994), BRICS Notes Series NS-94-6, Dept. of Comp. Sci., Univ. of Århus, pp. 416-428, 1994.

[18] Zohar Manna and Richard Waldinger. *Fundamentals of Deductive Program Synthesis*. TSE 18(8): 674 – 704, 1992.

[19] R. M. Burstall, J. Darlington. *A transformation system for developing recursive programs.* J. ACM, 24(1), pp. 44-67, 1977.

[20] A.W. Biermann, R. Kirshnaswany. Constructing programs from example computations. IEEE Trans. On Software Engineering, vol. 2, pp. 141 -153, 1976.

[21] Enn Tyugu. *Higher-Order Dataflow Schemas*. Theoretical Computer Science, v.90, pp. 185 – 1998, 1991.

Resümee

Antud töös on välja töötatud programmide automaatse sünteesi moodul Java keele jaoks. See töö on osa projektist, mille eesmärgiks on luua visuaalse spetsifitseerimiskeelega programmeerimiskeskkond Java baasil.
See keskkond on ette nähtud erinevate valdkondade projekteerimisprobleemide lahendamiseks. Programmide automaatse sünteesi moodul kasutab programmide struktuurse synteesi meetodit (SSP). Tuuakse sisse metainterfeisi mõiste, mis kujutab endast Java klassi loogilist spetsifikatsiooni. See laiendab Java klasside kasutamise võimalusi.
Metainterfeisid teisendatakse graafi kujule. Programmi süntees põhineb nende graafide kasutamisel planeerija poolt. Planeerija on otsimisprogramm, mis püüab leida konstruktiivset tõestust probleemile mis on kirjeldatud metainterfeisi spetsifikatsioonis. Probleem võib sisaldata alamülesandeid, mille puhul sünteesi moodul kasutab teist programmi alameesmärke planeerimiseks. Kui tõestus on leitud, teisendatakse see sama struturiga algoritmiks, mis antakse koodigeneraatorile Java lähteteksti genereerimiseks. Genereeritud tekst komileeritakse ja käivitatakse Java refleksiivsuse vahendite abil.

Appendixes

Appendix A

class Planner.

```
package ee.ioc.cs.vsle.synthesize;

import java.util.*;
import ee.ioc.cs.vsle.util.db;

/**
 This class is responsible for  planning.
 @author Ando Saabas, Pavel Grigorenko
 */

public class Planner {
    private ArrayList algorithm;

    //"m_" - class members
    private HashSet m_knownVars;
    private HashSet m_targetVars;
    private HashSet m_axioms;
    private HashSet m_allRels;
    private HashSet m_allVars;
    private HashSet m_foundVars;
    private HashSet m_subtaskRels;
    private boolean m_computeAll = false;
    private Planner planner;

    /**
         @param problem the specification unfolded as a graph.
         @param computeAll set to true, if we try to find everything that can be
computed on the problem graph.
         @param alg - algorithm
     */
    public Planner(Problem problem, boolean computeAll, ArrayList alg) {
        if (alg == null) {
            algorithm = new ArrayList();
        }
        else {
            algorithm = alg;
        }
        this.m_computeAll = computeAll;
        this.m_knownVars = problem.knownVars;
        this.m_targetVars = problem.targetVars;
        this.m_axioms = problem.axioms;
        this.m_allRels = problem.allRels;
        this.m_allVars = new HashSet(problem.allVars.values());
        Iterator allVarsIter = m_allVars.iterator();

        start_planning();
    }
```

```
    /**
     * Manages synthesis process
     * start_planning
     */
    public void start_planning() {
        if (!lin_planner()) {
            if (!sub_planning()) {
                lin_planner();
            }
        }
    }


    /** Does the linear planning.
     @return true if problem is solved, otherwise false.
     */
    boolean lin_planner() {

        /* while iterating through hashset, items cant be removed from/added to that
set.
         Theyre collected into these sets and added/removedall together after iteration
         is finished*/
        HashSet newVars = new HashSet();
        HashSet removableVars = new HashSet();
        HashSet removableTargets = new HashSet();
        HashSet removableAxioms = new HashSet();
        HashSet allTargetVars = new HashSet(m_targetVars);
        // the set of variables we know in the graph.
        m_foundVars = new HashSet(m_knownVars);

        Iterator knownVarsIter;
        Iterator relIter;
        Iterator targetIter;
        Iterator axiomIter;

        Var var, targetVar, relVar;
        Rel rel;
        // Iterate through all components/variables
        boolean changed = true;
        db.p("------Starting linear planning--------");
        m_subtaskRels = new HashSet();
        axiomIter = m_axioms.iterator(); //r - problem.
        while (axiomIter.hasNext()) {
            rel = (Rel) axiomIter.next();
            if (rel.subtaskFlag == 0) {
                m_knownVars.addAll(rel.outputs); //r - problem.addKnown(rel.outputs);
                algorithm.add(rel);
                removableAxioms.add(rel);
                m_foundVars.addAll(rel.outputs);
            }
            else {
                m_subtaskRels.add(rel);
                removableAxioms.add(rel);
            }
        }
        relIter = m_allRels.iterator();
        while (relIter.hasNext()) {
            rel = (Rel) relIter.next();
            if (rel.subtaskFlag > 0 && rel.inputs.size() > 0) {
                m_subtaskRels.add(rel);
            }
        }
        m_axioms.removeAll(removableAxioms);
        int counter = 1;

        while ( (!m_computeAll && changed && !m_targetVars.isEmpty()) ||
               (changed && m_computeAll)) {
            db.p("----Iteration " + counter + "----");
            counter++;
            changed = false;
            targetIter = m_targetVars.iterator();
            while (targetIter.hasNext()) {
                targetVar = (Var) targetIter.next();
                if (m_knownVars.contains(targetVar)) {
                    removableTargets.add(targetVar);
```

```
            }
        }
        m_targetVars.removeAll(removableTargets);
        knownVarsIter = m_knownVars.iterator();
        while (knownVarsIter.hasNext()) {
            var = (Var) knownVarsIter.next();
            // Check the relations of all components
            relIter = var.rels.iterator();
            while (relIter.hasNext()) {
                rel = (Rel) relIter.next();
                if (m_allRels.contains(rel)) {
                    rel.flag--;
                    removableVars.add(var);
                    if (rel.flag == 0) {
                        boolean relIsNeeded = false;

                        for (int i = 0; i < rel.outputs.size(); i++) {
                            relVar = (Var) rel.outputs.get(i);
                            if (!m_foundVars.contains(relVar)) {
                                relIsNeeded = true;
                            }
                        }
                        if (rel.outputs.isEmpty()) {
                            relIsNeeded = true;
                        }
                        if (relIsNeeded && rel.subtaskFlag < 1) {
                            if (!rel.outputs.isEmpty()) {
                                newVars.addAll(rel.outputs);
                                m_foundVars.addAll(rel.outputs);
                            }
                            algorithm.add(rel);
                        }

                        m_allRels.remove(rel);
                        changed = true;
                    }
                }
            }
        }
        m_knownVars.addAll(newVars);
        m_knownVars.removeAll(removableVars);
        newVars.clear();
    }
    boolean solved = m_targetVars.isEmpty();

    if (!m_computeAll) {
        Optimizer optimizer = new Optimizer();
        algorithm = optimizer.optimize(algorithm, allTargetVars);
    }
    return solved;

}

/** Does the planning with subtasks.
 @return true if problem is solved, otherwise false.
 */

public boolean sub_planning() {
    HashSet newVars = new HashSet();
    HashSet removableVars = new HashSet();
    HashSet removableRels = new HashSet();
    HashSet removableSubRels = new HashSet();
    HashSet subTaskInputs = new HashSet();
    db.p("-------Sub Planning-------");

    boolean changed = true;
    int counter = 0;
    while (changed) {
        changed = false;
        db.p("-------Iteration " + ++counter + "-------");
        Iterator subIter = m_subtaskRels.iterator();
        while (subIter.hasNext()) {
            Rel rel = (Rel) subIter.next();
            for (int i = 0; i < rel.subtasks.size(); i++) {
                Rel subRel = (Rel) rel.subtasks.get(i);
                m_knownVars.addAll(subRel.inputs);
                subRel.flag -= subRel.inputs.size();
```

```
//////////////////////////////////////////
//remove rels with outputs of subTask inputs
subTaskInputs.addAll(subRel.inputs);
allRelIter = m_allRels.iterator();
while (allRelIter.hasNext()) {
    Rel trel = (Rel) allRelIter.next();
    for (int k = 0; k < trel.outputs.size(); k++) {
        Var outv = (Var) trel.outputs.get(k);
        if (subTaskInputs.contains(outv)) {
            trel.flag = 0;
            m_foundVars.add(outv);
            removableRels.add(trel);
        }
    }
}
m_allRels.removeAll(removableRels);
removableRels.clear();
//////////////////////////////////////
Iterator knownVarIter = m_knownVars.iterator();
while (knownVarIter.hasNext()) {
    Var var = (Var) knownVarIter.next();
    Iterator relIter = var.rels.iterator();
    while (relIter.hasNext()) {
        Rel subVarRel = (Rel) relIter.next();
        if (m_allRels.contains(subVarRel)) {
            subVarRel.flag--;
            removableVars.add(var);
            if (subVarRel.flag == 0) {

                boolean relIsNeeded = false;
                for (int j = 0; j < subVarRel.outputs.size();
                     j++) {
                    Var subVarRelOutVar = (Var) subVarRel.
                        outputs.get(j);
                    if (!m_foundVars.contains(
                        subVarRelOutVar)) {
                        relIsNeeded = true;
                    }
                }
                if (relIsNeeded) {
                    if (!rel.inAlgorithm) {
                        rel.inAlgorithm = true;
                        algorithm.add("<subtask>");
                        algorithm.add(subRel);
                    }
                    m_foundVars.addAll(subVarRel.outputs);
                    newVars.addAll(subVarRel.outputs);
                    algorithm.add(subVarRel);
                }
                m_allRels.remove(subVarRel);
                changed = true;
            }
        }
    }
}
if (m_foundVars.contains(subRel.outputs.get(0))) {
    if (rel.inAlgorithm) {
        algorithm.add("</subtask>");
    }
    algorithm.add(rel);
    newVars.addAll(rel.outputs);
    m_foundVars.addAll(rel.outputs);
    removableSubRels.add(rel);
    allRelIter = m_allRels.iterator();
    while (allRelIter.hasNext()) {
        Rel trel = (Rel) allRelIter.next();
        if (trel.outputs.contains(rel.outputs.get(0))) {
            removableRels.add(trel);
        }
    }
    m_allRels.removeAll(removableRels);

}
m_knownVars.addAll(newVars);
m_knownVars.removeAll(removableVars);
newVars.clear();
```

```
                }
            }
            m_subtaskRels.removeAll(removableSubRels);
        }
        removableVars.clear();
        Iterator targetIter = m_targetVars.iterator();
        while (targetIter.hasNext()) {
            Var targetVar = (Var) targetIter.next();
            if (m_foundVars.contains(targetVar)) {
                removableVars.add(targetVar);
            }
        }
        m_targetVars.removeAll(removableVars);
        return m_targetVars.isEmpty();

    }


    /**
     * returns synthesized algorithm as ArrayList object
     *
     * @return ArrayList
     */
    public ArrayList getAlgorithm() {
        return algorithm;
    }

}
```

Appendix B

class CodeGenerator.

```
package ee.ioc.cs.vsle.synthesize;

import java.util.*;
import java.util.regex.*;
import ee.ioc.cs.vsle.util.db;
import ee.ioc.cs.vsle.vclass.ClassField;

/**
 * The CodeGenerator prototype
 * @author Ando Saabas, Pavel Grigorenko
 */
public class CodeGenerator {
    final String tab = "        ";
    String offset = "";
    int subCount = 0;
    ArrayList subtask_outputs = new ArrayList();
    public CodeGenerator() {}

    /**
     * Generates the source code for a given algorithm
     *
     * @param algRelList ArrayList
     * @return String synthesized program's source code
     */
    public String generate(ArrayList algRelList) {
        String algorithm = "";
        StringBuffer alg = new StringBuffer();
        addOffset(1, 1);
        for (int i = 0; i < algRelList.size(); i++) {
            Object temp = algRelList.get(i);
            Rel rel = null;
            if (isRel(temp) && subCount == 0) { // && subCount == 0
                rel = (Rel) temp;
                alg.append(addOffset(0, 0) + getStringFromRel(rel));

                continue;
            }
            if (isString(temp) && ( (String) temp).equals("<subtask>")) {
                subCount++;
                alg.append(addOffset(0, 0) + "class Subtask_" + subCount +
                           " implements Subtask {\n");
                alg.append(addOffset(1, 1) +
                           "public Object[] run(Object[] in) throws Exception {\n");
                if (isRel(algRelList.get(i + 1)) &&
                    ( (Rel) algRelList.get(i + 1)).type == RelType.subtask) {
                    Rel subtask = (Rel) algRelList.get(i + 1);
                    subtask_outputs.add(subtask.outputs.get(0));
                    int sub_inputsCount = subtask.inputs.size();
                    offset += tab;
                    for (int j = 1; j < sub_inputsCount + 1; j++) {
                        Var var = (Var) ( (Rel) algRelList.get(i + 1 + j)).
                            outputs.get(0);
                        if (var.type.equals("int")) {
                            alg.append(addOffset(0, 0) + var + " = " +
                                       intToObj("in[" + (j - 1) + "]") + ";\n");
                        }
                    }
                    i += sub_inputsCount + 1;
                }
                continue;
            }
            if (isRel(temp)) {
                rel = (Rel) temp;
                if (subtask_outputs.contains(rel.outputs.get(0))) {
                    alg.append(addOffset(0, 0) +
                               "Object[] out = new Object[1];\n" +
```

```
                            addOffset(0, 0) + "out[0] = " +
                            objFromInt(rel.inputs.get(0).toString()) + ";\n" +
                            addOffset(0, 0) + "return out;\n" +
                            addOffset(2, 1) + "}\n" +
                            addOffset(2, 1) + "}\n" +
                            addOffset(0, 0) + "Subtask_" + subCount +
                            " subtask_" + subCount +
                            " = new Subtask_" + subCount + "();\n"
                            );
                }
            }
            if (isString(temp) && ( (String) temp).equals("</subtask>")) {
                if (isRel(algRelList.get(i + 1)) &&
                    ( (Rel) algRelList.get(i + 1)).type ==
                    RelType.method_with_subtask) {
                    Rel method_subtask = (Rel) algRelList.get(i + 1);
                    if (method_subtask.subtasks.size() > 0) {
                        alg.append(addOffset(0, 0) +
                                   (Var) method_subtask.outputs.get(0) +
                                   " = " +
                                   method_subtask.getObject(method_subtask.
                            object) + method_subtask.method +
                                   "(subtask_" + subCount + ");\n"
                                   );
                        subCount--;
                        i++;
                    }
                }
            }
        }
        return alg.toString();
    }

    /**
     * Support method of generate()
     *
     * @param rel Rel
     * @return String converts rel to a string representation
     */
    String getStringFromRel(Rel rel) {
        Pattern pattern;
        Matcher matcher;

        if (rel.type == RelType.alias) {
            return "";
        }
        if (rel.type == RelType.javamethod) {
            Var op = (Var) rel.outputs.get(0);

            if (op.type.equals("void")) {
                return (rel.getObject(rel.object) + rel.method +
                        rel.getParameters() + ";\n");
            }
            else {
                return ( (Var) rel.outputs.get(0) + " = " +
                        rel.getObject(rel.object) +
                        rel.method +
                        rel.getParameters() + ";\n");
            }
        }
        else if (rel.type == RelType.equation) {
            // if its an array assingment
            if (rel.inputs.size() == 0 && rel.outputs.size() == 1) {
                String assign;
                Var op = (Var) rel.outputs.get(0);

                if (op.field.isPrimOrStringArray()) {
                    String[] split = rel.method.split("=");
                    assign = op.field.type + " " + " TEMP" +
                        Integer.toString(rel.auxVarCounter) + "=" + split[1] +
                        ";\n";
                    assign += addOffset(0, 0) + rel.getObject(op.object) +
                        op.name + " = TEMP" +
                        Integer.toString(rel.auxVarCounter) + ";\n";
                    rel.auxVarCounter++;
                    return assign;
```

```
    }
}

if (rel.inputs.size() == 1 && rel.outputs.size() == 1) {
    String s1, assigns = "";
    Var ip = (Var) rel.inputs.get(0);
    Var op = (Var) rel.outputs.get(0);

    if (ip.field.isArray() && op.field.isAlias()) {

        for (int i = 0;
             i < ( (Var) rel.outputs.get(0)).field.vars.size();
             i++) {
            s1 = ( (ClassField) op.field.vars.get(i)).toString();
            assigns += "        " + rel.getObject(op.object) + s1 +
                " = " +
                ip + "[" + Integer.toString(i) + "];\n";
        }
        return assigns;
    }
    if (op.field.isArray() && ip.field.isAlias()) {

        assigns += op.field.type + " TEMP" +
            Integer.toString(rel.auxVarCounter) + " = new " +
            op.field.arrayType() + "[" + ip.field.vars.size() +
            "];\n";
        for (int i = 0; i < ip.field.vars.size(); i++) {
            s1 = ( (ClassField) ip.field.vars.get(i)).toString();
            assigns += "        " + " TEMP" +
                Integer.toString(rel.auxVarCounter) + "[" +
                Integer.toString(i) + "] = " +
                rel.getObject(ip.object) +
                s1 +
                ";\n";
        }
        assigns += "        " + op + " = " + " TEMP" +
            Integer.toString(rel.auxVarCounter);
        rel.auxVarCounter++;
        return assigns;
    }
}

Var var;
String m = new String(rel.method + " ");
String left = "";
String left2 = "";
String right = "";

for (int i = 0; i < rel.inputs.size(); i++) {
    var = (Var) rel.inputs.get(i);
    pattern = Pattern.compile("([^a-zA-Z_])(([a-zA-Z_0-9]+\\.)*)" +
                              var.name + "([^a-zA-Z0-9_])");
    matcher = pattern.matcher(m);
    if (matcher.find()) {
        left = matcher.group(1);
        left2 = matcher.group(2);
        right = matcher.group(4);
    }
    m = m.replaceFirst("([^a-zA-Z_]" + left2 + var.name +
                       "[^a-zA-Z0-9_])",
                       left + rel.getObject(var.object) + var.name +
                       right);
}
left2 = "";
var = (Var) rel.outputs.get(0);
pattern = Pattern.compile("([^a-zA-Z_]?)(([a-zA-Z_0-9]+\\.)*)" +
                          var.name + "([^a-zA-Z0-9_])");
matcher = pattern.matcher(m);
if (matcher.find()) {
    left = matcher.group(1);
    left2 = matcher.group(2);
    right = matcher.group(4);
}

m = m.replaceFirst("([^a-zA-Z_]?" + left2 + var.name +
                   "[^a-zA-Z0-9_])",
                   left + rel.getObject(var.object) + var.name +
```

```
                        right);

        if ( ( (Var) rel.outputs.get(0)).type.equals("int") &&
            (!rel.getMaxType(rel.inputs).equals("int") ||
             rel.method.indexOf(".") >= 0)) {
            m = m.replaceFirst("=", "= (int)(") + ")";

        }
        return m;
    }
    else {
        String s1, s2, assigns = "";
        Var ip = (Var) rel.inputs.get(0);
        Var op = (Var) rel.outputs.get(0);

        if (ip.field.isArray() && op.field.isAlias()) {

            for (int i = 0; i < ( (Var) rel.outputs.get(0)).field.vars.size();
                 i++) {
                s1 = (String) ( (Var) rel.outputs.get(0)).field.vars.get(i);
                assigns += "        " + rel.getObject(op.object) + s1 +
                    " = " +
                    ip +
                    "[" + Integer.toString(i) + "];\n";
            }
            return assigns;
        }
        if (op.field.isArray() && ip.field.isAlias()) {
            for (int i = 0; i < ip.field.vars.size(); i++) {
                s1 = (String) ip.field.vars.get(i);
                assigns += "        " + op + "[" + Integer.toString(i) +
                    "] = " + rel.getObject(ip.object) + s1 + ";\n";
            }
            return assigns;
        }
        if (op.field.isAlias() && ip.field.isAlias()) {
            for (int i = 0; i < ip.field.vars.size(); i++) {
                s1 = (String) ip.field.vars.get(i);
                s2 = (String) op.field.vars.get(i);

                assigns += "        " + rel.getObject(op.object) + s2 +
                    " = " +
                    rel.getObject(ip.object) + s1 + ";\n";
            }
            return assigns;
        }

        return op + " = " + ip;
    }

}

/**
 * used in formatting the source code
 * @param incr int
 * @param times int
 * @return String
 */
String addOffset(int incr, int times) {
    //0 - no change, 1 - increase, 2 - decrease
    if (incr == 1) {
        for (int i = 0; i < times; i++) {
            offset += tab;
        }
        return offset;
    }
    else if (incr == 2) {
        for (int i = 0; i < times; i++) {
            offset = offset.substring(tab.length());
        }
        return offset;
    }
    else {
        return offset;
    }

}
```

```
    /**
     * returns true if a given object is of Rel type
     *
     * @param ob Object
     * @return boolean
     */
    boolean isRel(Object ob) {
        return ob instanceof Rel;
    }

    /**
     * returns true if a given object is of String type
     *
     * @param ob Object
     * @return boolean
     */
    boolean isString(Object ob) {
        return ob instanceof String;
    }

    /**
     * int -> Integer
     *
     * @param var String
     * @return String
     */
    String objFromInt(String var) {
        return "new Integer(" + var + ")";
    }

    /**
     * cast int to Object
     *
     * @param var String
     * @return String
     */
    String intToObj(String var) {
        return "((Integer)" + var + ").intValue()";
    }

}
```