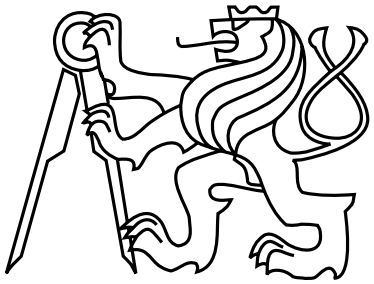
Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studiijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

Framework Squander Usage

Bc. Martin Kožený

Supervisor: Ing. Jiří Daněček

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Science and Engineering

October 21, 2011

Aknowledgements

At this point I would like to thank supervisor of my thesis Mr Ing. Jiří Daněček and author of Squander framework Mr Aleksandar Milicevic for valuable comments and advices to this work.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague, 30. 9. 2011 .....

Abstract

The aim of this work is to describe and study framework Squander developed on Massachusetts Institute of Technology by Mr. Aleksandar Milicevic. This framework brings into language Java another way of programming, which can improve effectivness of implementing and computation performance of the program.

Work shows how was this framework used for implementing set of algorithms, especially NP graph algorithms, and compares that implementation with common imperative way of programming.

Abstrakt

Abstrakt práce by měl velmi stručně vystihovat její podstatu. Tedy čím se práce zabývá a co je jejím výsledkem/přínosem.
Očekávají se cca 1 – 2 odstavce, maximálně půl stránky.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As was said in abstract, the main purpose of this thesis is to describe and use framework Squander developed at Massachusetts Institute of Technology (MIT) by Mr Aleksandar Milicevic. This framework brings to Java declarative constructs, which are useful for implementing programs that involve computations that are relatively easy to specify but hard to solve algorithmically. In such cases is better to use declarative constraints to naturally express the core of the computation, whereas imperative code is natural choice to read input parameters and setting up data structures for the computation. This is big advantage of this framework, that programmer can smoothly switch between declarative logical and imperative programming.

By ability of mixing imperative and declarative code can programmer easily express constraints of problem in terms of existing data structures and objects on the heap. Despite having overhead of encoding and decoding, it is surprising how is competitive Squander's SAT-based solution with specialized heuristic developed for concrete problem.

In first chapter I am going to describe basic principles of logical programming using language Prolog and in following one meaning of annotations, which are used to express logical constructions in framework Squander. Last chapters are devoted to framework itself and its comparison to common imperativ way of programming.

Chapter 2

Logical programming principles

The most known logical programming in the world is definitely Prolog and thereby we can describe properties of logical programming using this language. His name is derived from term PROgramming in LOGic and was developed for programming of symbolic computation. His success led to formation new discipline in mathematical information technology - **logical programming**.

Logical programming focus on description of relation's properties without need to know how to do that.

2.1 Logical programming paradigm

Logical programming differ from imperative languages in following points:

1. no assignment statement
2. no cycles, no branching
3. no flow control
4. object is marked as *variable*, which satisfies some set of conditions, that are being during computation more specified

Logical programming is based on following concepts:

1. declaring facts about objects and relations between objects
2. declaring valid rules about objects and relations between thmeselves and computing queries

In logical programming are *facts* unconditional commands and *rules* are conditional commands. Facts and rules are stored in one shared database. Language does not differ between program and data.

2.1.1 Facts

For expressing *facts* and *rules* are used *clauses*. Facts are used for expressing unconditionally true assertions and are clauses with defined headers, but with no body. Usual way to ilustrate how to composed facts are family relationships.

```
parent(david, john).
parent(john, jane).
parent(ann, jane).
parent(john, richard).
parent(ann, richard).
man(john).
man(richard).
womam(ann).
woman(jane).
```

Every clause declares concrete fact about relation. We can see that relation `parent`, e.g. `parent(john, jane).` is concrete *instance* of this relation for *objects* `john` and `jane`. After declaring those facts is possible to form queries concerning relation `parent`.

```
?-parent(john, jane).
```

Example above shows, how it is possible to ask, if John is parent to Jane. Because language has this fact recorded in its environment and answer is:

```
yes
```

Similarly can be query constructed on non-existing fact:

```
?-parent(john, emily).

no
```

Answer to this query is `no` of course. Query can be also composed in a way, that we want to get some object, which is with other object in required relation.

```
?-parent(X, jane).

X = john
```

Here it is also possible to get other possible solution, so we get one more positive answer:

```
X = ann
```

After that are all possible answers exhausted, so for the next command we get:

```
no
```

Now we try to express little bit complicated query: who is mother of Jane. This query is necessary construct from two suqueries. First we limit set of solution to Jane's parents. For that purpose we use query already shown above:

```
?-parent(X, jane).
```

In variable `X` is now stored every object, who has relation `parent` to object `jane`. In our case `john` and `ann`. Now we have to limit this set of results to object, which is declared in clause `women`, so we add:

```
?-parent(X, jane), women(X).
```

As a result of these clauses we get:

```
X = ann
```

and nothing more. In framework Squander are as facts used objects, that are declared in framework's rules (see subsection 2.1.2). There are no facts declared explicitly so we do not devote to them any more.

2.1.2 Rules

Expressing knowledge by facts cannot be always effective. Complexity of relationships in family expressed by unconditional commands would lead to big expansion of database. Despite having unlimited memory available, searching for relevant information would has been time consuming. For this reason Prolog provides conditional expressions - *rules*.

By investigating facts is possible to derived new rule based on logical or factual context. That knowledge allow us to express facts, which are not explicitly stored in database. Let us show it in following example:

```
mother(X, Y) :- parent(X, Y),woman(X).
```

Left side of rule expressed so called **head of rule** and right side **body of rule**. This rule express, that `X` is mother of `Y`. Rule is only labeled generalization of last example in previous subsection 2.1.1. In next example will be shown more complicated construct:

```
brother(X, Y) :- parent(O, Y),parent(O, X),man(X).
```

Expression above means, that `X` is brother of `Y` if exists at least one object `O`, which is common in relation `parent` for both `X` and `Y` assuming `X` is a man. Mathematical interpretation of teh rule is:

„For all arbitrary persons X and Y,
if some of the parents of X is O
and some of the parents of Y is O
and person X is a man,
then X is brother of Y.“

When calling this rule with following parameters:

```
brother(richard, Y).

Y = jane
```

Rules are main construct of framework Squander, because they defined state of object before and after computation, declares which object or object's properties can be modified etc. On the other hand these rules are not called as it is in Prolog, but they are declared as metadata for handling with objects. More about Squander's rules in chapter 4.

2.2 Terminology

When simplyfying in Prolog, we can say, that in every task appear *objects* and *relations*. Name of objects are called *terms* and name of relations are called *predicates*. Terms are analogous to arithmetic expressions, which point to the computed value, and predicates are analogous to name of procedures, which defines relationship between input parameters and output values, in imperative programming language.

There are two types of terms: *simple terms* consisting of constants (e.g. `ann`, `richard`,...) and *compound terms*. Compound terms are also called *structures* is every term containing simple or compound term.

In previous subsection 2.1.2, there are predicates `parent`, `man` and `woman` as names of three relations defined by program. Predicate with name `parent` is defined as a set: `{(john,jane),(ann,jane),(john,richard),(ann,richard)}`. Next predicate with name `man` is defined as a set: `{john,richard}` and finally predicate with name `woman` is defined as a set: `{ann,jane}`.

Finally there are three types of *formulas* in Prolog:

- *atomic* - basic formulas (e.g. `parent(john, jane).`, `parent(ann, jane).`)
- *conditional command* - implication constructs $A :- P_1, P_2, \ldots, P_n$ where $P_1, P_2, \ldots, P_n$ are atomic formulas (e.g. `brother(X, Y) :- parent(O, Y),parent(O, X),man(X).`)
- *target clauses* - query type ($?- C_1, C_2, \ldots, C_n$ where $C_1, C_2, \ldots, C_n$ are targets) (e.g. `?-parent(X, jane), women(X).`)

2.3 Evaluation in Prolog

Main difference between **procedural semantics** and **declarative semantics** is shown on clause below:

```
P :- Q, R.
```

where `P` and `Q` are arbitrary forms of terms. We can read this clause from declarative point of view:

- `P` is true, if `Q` and `R` are true
- from validity of `Q` and `R` follow `P`

From procedural point of view has clause different meaning:

- to solve problem `P`, it is necessary to solve **first** problem `Q` and **then** problem `R`
- to fulfill target `P`, it is necessary **first** to fulfil target `Q` and then fulfil target `R`

`Evaluation` of sequence of targets $G_1, G_2, ..., G_m$ in Prolog consists of following steps:

- If is sequence of targets empty, evaluation **succeded**.
- For non-empty sequence of targets, operation *SEARCHING* is invoked.
- *SEARCHING*: Clauses of the program from top to bottom are searched till first clause C occurence, whose head is successfuly unified with target G_1. If such clause is not found, evaluation ended unsuccessfuly.

  If is found clause C in form

  $$H :- B_1, B_2, \ldots, B_n$$

  then all its variables are renamed such, that new form C' of clause C, which has no varibles in common with targets $G_1, G_2, ..., G_m$. C' has form:

  $$H' :- B'_1, B'_2, \ldots, B'_n$$

  G and H are unified by substition S. In sequence of targets is target G_1 replaced by body of clause C' such, that new seqence has form:

  $$B'_1, B'_2, \ldots, B'_n, G_2, \ldots, G_m$$

  If is C fact, then $n = 0$ and sequence of targets is shortened to $m - 1$ targets.

  Then substituion S is donein order to make new list of targets, so new form is:

  $$B''_1, B''_2, \ldots, B''_n, G'_2, \ldots, G'_m$$

By recursive invoication procedure `Evaluate` is evaulated this sequence of targets. If this evaluation ends successfuly, previous evaluation is treated as also successful. If this evaluation does not end successfuly, last sequence of targets is left and it is made return to operation *SEARCHING*, where is continued immediately behind clause C in order to find some next usable clause.

Let us show evaulation in following rules, which used facts declared in subsection 2.1.1:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), parent(Z, Y).
```

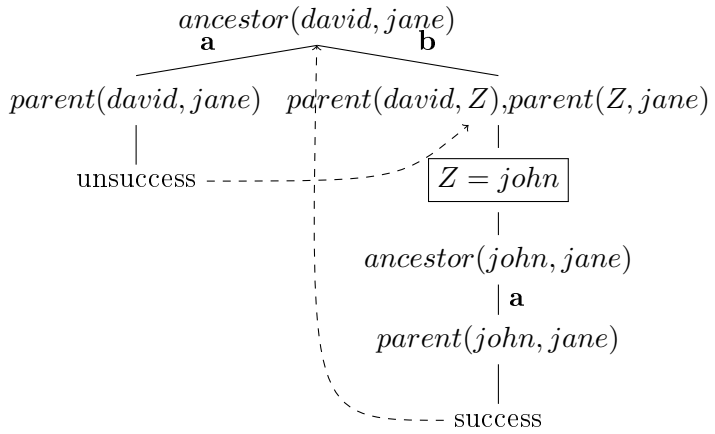


Figure 2.1: Expression evaluation in Prolog

First of the queries is answered unsuccessfuly when applied rule **a**, rule **b** is then successful.

Chapter 3

Meaning of annotations

3.1 Introduction

Annotations are tags that programmer insert into source code so that they can be processed by tools. The Java compiler understands a couple of annotations, but to go any further, you need to build your own processing tool or obtain a tool from a third party. Most common use of annotations are:

- Automatic generation of auxiliary files, such as deployment descriptors or bean information classes.
- Automatic generation of code for testing, logging, transaction semantics, and so on.

In EJB 3.0 are annotations used in such sense, that a lot of repetitive code is automated by annotations.

3.2 Annotations as metadata

Metadata are data about data. In context of computer program, metadata are data about the code. Since Java 5.0 release, programmer can insert arbitrary data into his source code. Annotations in Java is used like an *modifier*, placed before annotated item (it is a keyword similar to `public` or `static`). Annotations are used to annotate classes, fields or local variables and can be processed by tools that read them.

Each annotation is declared by annotation interface correspond to the element of the annotation.

Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program. Annotations can be read from source files, class files, or reflectively at runtime.

3.3 Examples of use

3.3.1 Field validation

Annotation type declarations are similar to normal interface declarations. An at-sign (`@`) precedes the `interface` keyword. Each method declaration defines an *element* of the annotation type. Method declarations must not have any parameters or a `throws` clause. Return types are restricted to primitives, `String`, `Class`, `enums`, annotations, and arrays of the preceding types. Methods can have default values. Here is an example annotation type declaration:

```
1 @Target(ElementType.FIELD)
2 public @interface Length {
3     int max();
4     int min();
5 }
```

Listing 3.1: Length annotation

Annotations can be annotatied itself. Such annotations are called *meta-annotations*. E.g. using (`@Target(ElementType.FIELD)`) indicates, that annotation type should be used to annotate only field declarations. This simple annotation is typically used for entity field e.g. of type `String` for permitted length of this field.

Once an annotation type is defined, you can use it to annotate declarations. An annotation is a special kind of modifier, and can be used anywhere that other modifiers (such as `public`, `static`, or `final`) can be used. By convention, annotations precede other modifiers. Annotations consist of an at-sign (`@`) followed by an annotation type and a parenthesized list of element-value pairs. The values must be compile-time constants. Here is a filed declaration with an annotation corresponding to the annotation type declared above:

```
1 public class Role {
2   @Length(max = 50)
3   String name;
4
5   @Length(max = 150)
6   String description;
7 }
```

Listing 3.2: Class Role

Annotation type without any elements is called *marker* annotation type, e.g.:

```
1 public @interface Entity{ }
```

Listing 3.3: Entity annotation

We can join these two types of annotation together in following example:

```
1 @Entity
2 public class Role {
3   @Length(max = 50)
4   String name;
5
6   @Length(max = 150)
7   String description;
8 }
```

Listing 3.4: Entity Role

Another type of meta-annotation (`@Target(ElementType.METHOD)`) is presented below and indicates using annotation only for method declarations:

```
1 @Target(ElementType.METHOD)
2 public @interface Test { }
```

Listing 3.5: Test annotation

In following example, there is updated entity shown above with tool for testing the annotations:

```
@Entity
public class Role {
  @Length(max = 50)
  String name;

  @Length(max = 150)
  String description;

  @Test
  public void showMeTheFunny() {
    System.out.println("Here you have funny");
  }

  public void foo() {
    System.out.println("Foo");
  }

  public void bar() {
    System.out.println("Bar");
  }
}
```

Listing 3.6: Extended class Role

```
public class RunTests {
   public static void main(String[] args) throws Exception {
      int passed = 0, failed = 0;
      for (Method m : Class.forName(args[0]).getMethods()) {
         if (m.isAnnotationPresent(Test.class)) {
            try {
               m.invoke(null);
               passed++;
            } catch (Throwable ex) {
               System.out.printf("Test %s failed: %s %n", m, ex.getCause());
               failed++;
            }
         }


      }
      System.out.printf("Passed: %d, Failed %d%n", passed, failed);

   }
}
```

Listing 3.7: Test of method annotation

```
1  public class TestRole {
2    public void test(Role role) throws Exception {
3      for (Field f : Role.class.getFields()) {
4        Annotation[] annotations = f.getDeclaredAnnotations();
5        for (Annotation a : annotations) {
6          if (a.annotationType().equals(Length.class)) {
7            if (f.get(role) instanceof String && ((String)f.get(role)).length()
                 > ((Length) a).max())
8              throw new Exception(
9                  "Unacceptable length of field "+f.getName()+" of class Role");
10         }
11         ...
12       }
13     }
14   }
15 }
```

Listing 3.8: Test of entity Role annotation

3.3.2 Annotating Event Handlers

In this subsection is shown example presented in [2]. When programmer have to use in his code action listeners, it leads to write a lot of repetitive code. Listeners are usually declared:

```
1 myButton.addActionListener(new
2   ActionListener()
3   {
4     public void actionPerformed(ActionEvent event)
5     {
6       doSomething();
7     }
8 });
```

Listing 3.9: Action listener

Writing this boring code can be omitted by using annotations. Annotation will have form:

```
1 @ActionListenerFor(source="myButton") void doSomething() { . . . }
```

Listing 3.10: Annotation for performing action

Instead of calling action listener is every method tagged with annotation.

```
1  public class ButtonTest {
2    public static void main(String[] args) {
3      ButtonFrame frame = new ButtonFrame();
4      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5      frame.setVisible(true);
6    }
7  }
8  /**
9   * A frame with a button panel
10  */
11 class ButtonFrame extends JFrame {
12   public ButtonFrame() {
13     setTitle("ButtonTest");
14     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
15
16     panel = new JPanel();
17     add(panel);
18
19     // create buttons
20     yellowButton = new JButton("Yellow");
21     blueButton = new JButton("Blue");
22     redButton = new JButton("Red");
23
24     // add buttons to panel
25     panel.add(yellowButton);
26     panel.add(blueButton);
27     panel.add(redButton);
28
29     ActionListenerInstaller.processAnnotations(this);
30   }
31
32   @ActionListenerFor(source = "yellowButton")
33   public void yellowBackground() {
34     panel.setBackground(Color.YELLOW);
35   }
36
37   @ActionListenerFor(source = "blueButton")
38   public void blueBackground() {
39     panel.setBackground(Color.BLUE);
40   }
41
42   @ActionListenerFor(source = "redButton")
43   public void redBackground() {
44     panel.setBackground(Color.RED);
45   }
46
47   public static final int DEFAULT_WIDTH = 300;
48   public static final int DEFAULT_HEIGHT = 200;
49
50   private JPanel panel;
51   private JButton yellowButton;
52   private JButton blueButton;
53   private JButton redButton;
54 }
```

Listing 3.11: Use annotation for performing action

```
1 @Target(ElementType.METHOD)
2 @Retention(RetentionPolicy.RUNTIME)
3 public @interface ActionListenerFor
4 {
5   String source();
6 }
```

Listing 3.12: Declaration of annotation for performing action

We now need a mechanism to analyze them and install action listeners. That is the job of the `ActionListenerInstaller` class. The `ButtonFrame` constructor calls

```
1 ActionListenerInstaller.processAnnotations(this);
```

Listing 3.13: Action Installer invocation

The static `processAnnotations` method enumerates all methods of the object that it received. For each method, it gets the `ActionListenerFor` annotation object and processes it.

Here, we use the getAnnotation method that is defined in the `AnnotatedElement` interface. The classes `Method`, `Constructor`, `Field`, `Class`, and `Package` implement this interface. The name of the source field is stored in the annotation object. We retrieve it by calling the `source` method, and then look up the matching field.

```
1 String fieldName = a.source();
2 Field f = cl.getDeclaredField(fieldName);
```

Listing 3.14: Looking up for field

For each annotated method, we construct a proxy object that implements the `ActionListener` interface and whose `actionPerformed` method calls the annotated method. The details are not important. The key observation is that the functionality of the annotations was established by the `processAnnotations` method. In Example 3.15, the annotations were processed at run time. It would also have been possible to process them at the source level. A source code generator might have produced the code for adding the listeners. Alternatively, the annotations might have been processed at the bytecode level. A bytecode editor might have injected the calls to `addActionListener` into the frame constructor.

```
1  public class ActionListenerInstaller {
2    /**
3     * Processes all ActionListenerFor annotations in the given object.
4     *
5     * @param obj
6     *            an object whose methods may have ActionListenerFor annotations
7     */
8    public static void processAnnotations(Object obj) {
9      try {
10       Class cl = obj.getClass();
11       for (Method m : cl.getDeclaredMethods()) {
12         ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
13         if (a != null) {
14           Field f = cl.getDeclaredField(a.source());
15           f.setAccessible(true);
16           addListener(f.get(obj), obj, m);
17         }
18       }
19     } catch (Exception e) {
20       e.printStackTrace();
21     }
22   }
23
24   /**
25    * Adds an action listener that calls a given method.
26    *
27    * @param source
28    *            the event source to which an action listener is added
29    * @param param
30    *            the implicit parameter of the method that the listener calls
31    * @param m
32    *            the method that the listener calls
33    */
34   public static void addListener(Object source, final Object param,
35       final Method m) throws NoSuchMethodException,
36       IllegalAccessException, InvocationTargetException {
37     InvocationHandler handler = new InvocationHandler() {
38       public Object invoke(Object proxy, Method mm, Object[] args)
39           throws Throwable {
40         return m.invoke(param);
41       }
42     };
43     Object listener = Proxy.newProxyInstance(null,
44         new Class[] { java.awt.event.ActionListener.class }, handler);
45     Method adder = source.getClass().getMethod("addActionListener",
46         ActionListener.class);
47     adder.invoke(source, listener);
48   }
49 }
```

Listing 3.15: Action Listener Installer class

Chapter 4

Framework description

4.1 Introduction

4.1.1 Overview

Squander is a framework providing unified environment for both declarative constraints and imperative statements in single program. This is very practical when implementing problems, which are easy to define but difficult to solve them algorithmically. In such cases, declarative constraints can be natural way to express problem, whereas imprative code is used for setting up the problem and data manipulation.

Thanks to ability of mixing imperative code with executable declarations, it is possible to express problem in terms of existing data structures and then run framework solver, which according to given constraints update the heap to reflect the solution.

Without having technology like this one, programmer has to translate his program to external solver, then run the solver and then again manually translate the solution back to the native programming language.

4.1.2 Architecture

We can divide running of Squander in following steps:

1. serialize heap into relations
2. translate specs and heap relations into Kodkod
3. translate relational into boolean logic
4. (if a solution is found) restore relations from boolean assignments (if a solution is found) restore field values from relations
5. (if a solution is found) restore the heap to reflect the solution

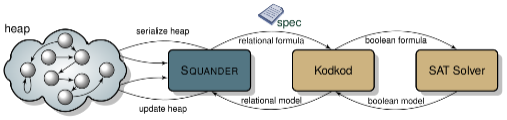


Figure 4.1: Architecture diagram

Further will be explained in deatil all of these steps and terms like Kodkod etc.

4.1.3 Applications

In this section will be briefly described applications of framework. Most typical applications are these:

- **solving hard constraint problems** - puzzles (solitaire, sudoku, n-queens,...), graph problems (traveling salesman problem, Hamiltonian path, general bisection breadth,...), schedulers, dependency managers,...
- **test input generation** - e.g. generate data structure instances that satisfy complex constraints
- **specification validation** - specifications can also contain errors, and the most intuitive way to test a specification would be to execute it on some concrete input and see if the result makes sense or not

- **runtime assertion checking** - check whether a given rich property holds at an arbitrary point during the execution of a program

On the other hand, framework has some limitations:

- **boundedness** - everything has to be bounded → framework cannot generate an arbitrary set of new objects (which may be needed to satisfy a specification); instead, the exact number of new objects of each class must be specified by the user
- **small integers** - integers must also be bounded to a small bitwidth (to make the solving tractable), which can occasionally cause subtle integer overflow bugs, which are typically hard to find
- **equality issues** - referential equality is used by default for all classes except for `String`, so it is impossible to write a spec that asserts that two objects are equal in the sense of Java `equals`
- **lack of support for higher-order expressions** - it is not possible to write a specification that says „find a path in the graph such that there is no other path in the graph longer than it“ and solve it with Squander; it is possible, however, to express and solve „find a path in the graph with at least k nodes“, which is computationally as hard as the previous problem, because a binary search can be used to efficiently find the maximum k for which a solution exists

4.2 Execution of Squander

Information about framework presented in this section are taken from [1], where is thre background of Squander described in detail.

4.2.1 Kodkod

Kodkod is a solver for relational logic. Kodkod requires bounded universe , a set of untyped relations, bounds for every relation and relation formula. Then translates given problem into boolean satisfiability problem (SAT) and applies of-the- shelf SAT solverto search for satysfiyng solution, which is reflected back if found.

When are the relations in Kodkod created, they are untyped, meaning that every relation can potentionally contain any tuple drawn from the finite universe. Actual set of tuples, which relation actually may contains is defined through Kodkod *bounds*:

- *lower bound* to define tuples, that relation **must** contain.
- *upper bounds* to define tuples, which relation **may** contain.

The size of these bounds has a big influence on search time - the fewer tuples are in the difference of lower and upper bound, the smaller search space is, the faster solving is.

4.2.2 JFSL

JSFL is formal lightweight specification for Java supporting relational and set algebra, as well as common Java operators. Using expressive power of relation algebra, JFSL makes it easy to succinctly and formally specify complex properties about Java programs such as method pre and postcondition, class invariants and so called *frame conditions*, which means portion of the heap, that is methos allow to modify. It also supports *specification fields* which can be useful for specifying abstract data types.

4.2.2.1 JFSL expressions

JFSL expressions are evaluated into relations. JFSL provides common algebra operators, together with interger and boolean operators. Some of them are shown in Table 4.1 and quantified operators, which are also provided by the framework, are shown in Table 4.2.

Operator	Description
`#`	set cardinality
`no`	„no“ multiplicity (empty)
`lone`	„lone“ mulitiplicity (zero or one)
`one`	„one“ multiplicity (exactly one)
`some`	„some“ multiplicity (one or more)
`!`	boolean negation
`-`	integer negation
`sum`	integer summation
`=>`	boolean implication
`<=>`	boolean equivalence („if and only if“)
`?`	if-then-else ternary operator (as in Java)
`@+`	relational union
`@-`	relational difference
`@&`	relational intersection

Table 4.1: Unary expressions supported by JFSL

Operator	Description	Usage
`all`	universal quantifier	`all x: T | P(x)`
`some`	existential quantifier	`some x: T | P(x)`
`sum`	integer summation	`sum i: int | a[i]`
`union`	set comprehension	`{x: T | P(x)}`

Table 4.2: Quantified expressions supported by JFSL

4.2.2.2 JFSL annotations

JFSL annotations are written as Java annotations and are for interaction between framework and programmer. Main components (JFSL annotations) are:

- `@Invariant` - this annotaion is attached to the classes and used to define, that condition must be fulfilled for the given class. That means, that this condition has to be fulfilled before and after execution.
- `@Requires` - this annotation specifies constraints before method invocation. The method is expected to execute correctly if only the precondition is satisfied immediately before method invocation. Class invariants are added immediately to the method preconditions.

- `@Ensures` - attached to the methods and used to specify constraints on the state, which have to be satisfied after method invocation. It means, that it captures all effect that method is expected to produce. Class invariants are implicitly added to method postconditions.
- `@Modifies` - this annotation is attached to the methods to specify frame conditions. Frame condition can hold up 4 different pieces of specification (syntax: `@Modifies ("f [s][l][u]")`). First, and the only one mandatory piece, is the name of modifiable field, `f`. It is optionally followed by *instance selector* `s`, followed by *lower bound* and the *upper bound*. Instance selector specifies instances, for which the field may be modified - if not specified, it is assumed „all“. The lower bound contains concrete field values for some objects in the post-state. The upper bound holds the possible field values in the post-state.
- `@SpecField` - attached to the classes and used to define specification fields. Definition of type specification consists of type declaration, and optionally an abstract function. the abstraction function defines, how the field value is computed in terms of other fields. For example `@SpecField("x: one int| x = this.y - this.z")` defines a singleton integer field `x`, the value of which must be equal to the difference of `y` and `z`. Specifications fields are inherited from super-types and sub-types can override the abstraction function (by simply redefining it), a feature that is particularly useful for specifying abstract datatypes, such as Java collections.

In context of these specification, the goal is to execute a method, that makes modification of the portion of the heap (specified in frame condition), so that final state satisfies postcondition.

4.2.3 From object heap to relational logic

Execution of Squander begins in client's code when invoking method `Squander.exe()` involving following steps:

1. Assembling all revelant constraints from annotations, as well as class annotations corresponding to all invariant classes.
2. Construction relations representing objects and their fields in pre-state and adding additional relations for holding their values in the post-state, along with their Kodkod bounds
3. Parsing constraints and converting them to a single relational formula
4. If solution is found, translation of the Kodkod result objects into updates of Java heap state, by modification of the object fields

4.2.3.1 Traversing the heap

For discovering reachable portion of the heap, bread-first search algorithm is used, which started from the set of root objects and repeatedly visiting all children until all reachable objects have been visited. The most important part is how to enumerate childre, i.e. how to serialize a given object into a set of field values.

4.2.4 Relations and bounds

After traversing the heap, founding all reachable objects and discovered all classes/-fields reffered in the specification, we have enough information to construct the relation, that represent state of the heap.

The translation does not used all fields, but only those, whose are considered as *relevant fields* - those, who are explicitly mentioned in the specification (Squander's annotation). Similarly, not all reachable objects are needed; only objects reachable by following the relevant fields are included in the translation. These objects will be referred to as *literals*.

First we define a finite universe consisting of all literals, plus integers within the bound. For every literal, a unary relation is created.

For each Java type, one could either create a new relation (with appropriate bounds so that it contains the known literals), or one could construct a relational expression denoting the union of relations corresponding to all instance literals of that type.

For every field, a relation of type `fld`.*declType* $\rightarrow$ `fld`.*type* is created to hold assignments of field values to objects. If the field is modifiable (inferred from its mention in a `@Modifies` clause), an additional relation is created, with the suffix „pre“ appended to denote the pre-state value. Relations for unmodifiable fields are given an exact bound that reflects the current state of the heap. For the modifiable relations, the „pre“ relation is given the same exact bound, and the “post” relation is bounded so that it may contain any tuple permitted by the field's type. Local variables, such as `this`, `return`, and method arguments are treated similarly to literals.

Table 4.3 below summarizes how relations and bounds are created. Function `rel` takes a Java element and, depending whether the element is modifiable, returns either one or two relations. Function `bound` takes a Java element and its corresponding relation, and returns a bound for the relation. The `Bound` data type contains both lower and upper bounds. If only one expression is passed to its constructor (**B**), both bounds are set to that value. Helper functions `is_mod`, `is_post` and `fldval` are used to check whether a field is modifiable, to check whether a relation refers to the post-state, and to return a literal that corresponds to the value of a given field of a given literal, respectively.

rel :: Element → [Relation]		
rel (Literal lit)	=	[**R**(lit.*name*, lit.*type*)]
rel (Type t)	=	$\bigcup_{lit<:t}$ **rel** lit
rel (Field fld)	=	
if is_ mod(fld)		
[**R**(fld.*name*, fld.*declType* →fld.*type*)] ++		
[**R**(fld.*name* + _pre, fld.*declType* →fld.*type*)]		
else		
[**R**(f.*name*, f.*declType* →f.*type*)]		
rel (Local var)	=	[**R**(var.*name*, var.*type*)]
rel :: Element, Relation → [Bound]		
bound (Literal lit) (Relation r)	=	**B**(lit)
bound (Field f) (Relation r)	=	
if is_ mod(fld) ∧ is_post(r)		
[**B**({}, ext(fld.*declType*×fld.*type*))]		
else		
[**B**($\bigcup_{lit<:Object}$ **rel** lit × fldval(lit,fld))]		
bound (Return ret) (Relation r)	=	[**B**({}, ext(ret.*type*))]
bound (Local var) (Relation r)	=	[**B**(var)]
[Type] → Expression		(hepler)
ext []	=	{}
ext (t:[])	=	$\bigcup_{lit<:t}$ lit
ext (t:xs)	=	**ext** t × **ext** xs

Table 4.3: Translation of different Java constructs into relations (function `rel`) and bounds (function `bound`)

4.2.5 Example: translation of `BST.insert`

We will use for illustration of translation Binary Search Tree example. This BST has single root node. Every node contains integer key value and pointer to the left and right node. To verify validity of BST, some Squander's specifications are used (see Listing 4.1). `@SpecField` constraint declares, that set of BST nodes consist of root node and all nodes in left and right subtree of this root node except for `null` values. Another constraints represented by `@Invariant` annotation delares: (1) there are no loops in BST - using transitive closure to define, that actual node is not accessible transitively via field `parent` - and (2) and for every node n in the BST, the *key* of n is strictly greater, than all keys of all nodes in the left subtree and strictly lower than all keys of all nodes in its right subtree - there is used reflexive transitive closure operator (`.*`) to conveniently specify all reachable nodes starting from the root node.

```
1  @SpecField("this.nodes = this.root.*(left + right) - null")
2  public class BST {
3    @Invariant({
4        /*format tree*/"this !in this.^parent",
5        /*left sorted*/"all x: this.left.*(left + right) - null | " +
6            "x.key < this.key",
7        /*right sorted*/"all x: this.right.*(left + right) - null | " +
8            "x.key > this.key" })
9    public static class Node {
10     Node left, right, parent;
11     int key;
12   }
13
14   private Node root;
15 }
```

Listing 4.1: Binary Search Tree declaration

Methods for modifying tree, `insert()` and `remove()` shown in Listing 4.2 are obvious. In order to insert node into tree, node with same key may not exist in the tree, and after the insertion, tree must contain the given node. All left and right pointers of nodes and pointer to root node are allowed to modify. Deletion is defined in similar way, BST must contain node with the same key as node to remove and after deletion this node may not exist in BST.

```
1  @Requires("z.key !in this.nodes.key")
2  @Ensures("this.nodes = @old (this.nodes + z)")
3  @Modifies("Node.left , Node.right , this.root")
4  public void insert(Node z) {
5    Squander.exe(this, z);
6  }
7
8  @Requires("z in this.nodes")
9  @Ensures("this.nodes = @old(this.nodes) - z")
10 @Modifies("Node.left , Node.right , this.root")
11 public void remove(Node z) {
12   Squander.exe(this, z);
13 }
```

Listing 4.2: Operations on BST

The resulting set of relations, shown in Table 4.4, are divided into three sections. Relations in the upper section are unary, unmodifiable relations, and represent objects on the heap. The middle section contains relations that are also unmodifiable, because they are used to either represent unmodifiable fields or values in the pre-state of modifiable fields. Finally, the relations in the bottom section represent the post-state of modifiable fields; these are the relations for which the solver will attempt to find appropriate values. By default, the lower bound is simply set to an empty set and the upper bound is the upper bound is set the extent of the field's type.

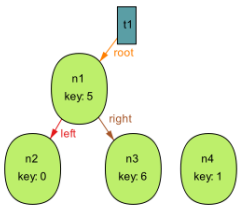


Figure 4.2: A snapshot of `t1.insert(t4)`

BST:	$\{t_1\}$
N1:	$\{n_1\}$
N2:	$\{n_1\}$
N3:	$\{n_1\}$
N4:	$\{n_1\}$
null:	$\{null\}$
BST_this:	$\{t_1\}$
z:	$\{n_4\}$
ints:	$\{0, 1, 5, 6\}$
key:	$\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
root_pre:	$\{(t_1 \to n_1)\}$
nodes_pre:	$\{(t_1 \to n_1), (t_1 \to n_2), (t_1 \to n_3), (t_1 \to n_4)\}$
right_pre:	$\{(n_1 \to n_2), (n_2 \to 0), (n_3 \to null), (n_4 \to null)\}$
left_pre:	$\{(n_1 \to n_3), (n_2 \to 0), (n_3 \to null), (n_4 \to null)\}$
root:	$\{\}, \quad \{t_1\} \times \{n_1, n_2, n_3, n_4\}$
nodes:	$\{\}, \quad \{t_1\} \times \{n_1, n_2, n_3, n_4\}$
left:	$\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$
right:	$\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$

Table 4.4: Translation of the heap from Figure 4.2

4.2.6 Tightening the bounds

By declaring as modifiable fields `left` and `right`, as in the specification in 4.1, we allow arbitrary modifications to the tree, as long as all constraints are satisfied. In effect, after the execution of the specification for the `insert()` method, the tree will contain all old nodes plus the new node, but the shape of the tree may randomly change.

If we want to change method specification so, that tree topology is preserved and new nodes can be append only to leaf nodes, we can do that by adding new clauses to the postcondition specification in sense, that left and right pointers of certain nodes must remain the same in the post state. This idea is completely right, but more efficient would to change the frame condition also for improving performance. Reducing search space by allowing modification only `left` and `right` pointers pointing to `null` value improve performance significantly.

Consider modified frame condition shown in Listng 4.3 specifying additional constraints on the modification of `left` and `right` fields. This frame condition shows idea specified in paragraph before and ensures, that all new nodes will be inserted at the leaf positions.

```
1 @Requires("z.key !in this.nodes.key")
2 @Ensures("this.nodes = @old ( this.nodes + z)")
3 @Modifies({"Node.left [{ n: this.nodes | n.left == null}]",
4     "Node.right [{ n: this.nodes | n.right == null}]",
5     "this.root"})
6 public void insert(Node z) {
7   Squander.exe(this, z);
8 }
```

Listing 4.3: Modified frame condition on `insert()` method

With the list of modifiable objects, Squander modifies the bounds for the corresponding field relation so that the current field values of the objects not to be modified are included in the lower bound, thus forcing the value to stay the same in the post state. For the heap shown in Figure 4.2, the modifiable objects for the `left` field are `n2` and `n3`, because their left pointers are currently set to null. Similarly, for the `right` field, the modifiable objects are also `n2` and `n3`. The updated bounds for these two fields are shown in Table 4.5.

left:	$\{(n_1 \to n_2), (n_4 \to null)\}$,	$(n_1 \to n_2) \cup (n_4 \to null) \cup \{n_2, n_3\} \times \{n_1, n_2, n_3, n_4\}$
right:	$\{(n_1 \to n_3), (n_4 \to null)\}$,	$(n_1 \to n_3) \cup (n_4 \to null) \cup \{n_2, n_3\} \times \{n_1, n_2, n_3, n_4\}$

Table 4.5: The updated bounds for the `left` and `right` relations

4.2.7 Minimizing the universe size

For representing relations in Squander are used single sequential arrays indexed by Java integer. Maximal count of members in this array is `Integer.MAX_VALUE` which is 2147483647. In practice this can be a problem, because framework for relation of arity k, which has n atoms in the universe (later it will be described, what is exactly meant under this term), allocates matrix of size n^k (when ternary relation contains more than 1290 atoms). Heap with more than 1290 objects are not uncommon, so a simple translation described in 4.2.5 would not work. In the next subsection will be described a different translation technique called *KodkodPart*, which was developed to minimize number of atoms representing object heap.

4.2.7.1 *KodkodPart* translation

KodkodPart translation achieves a universe with fewer atoms by mapping Java objects to Kodkod atoms (not injective mapping). That means that multiple literals are mapped to a

single atom, so that there will be fewer atoms than literals. The key requirements is, that there exists some inverse function, which maps back from the atoms literals.

Considering previous example with BST (see subsection 4.2.5), we can define domains $\mathcal{D}$, literals $\mathcal{L}$ and assignment literals to domains $\gamma : \mathcal{D} \to \mathcal{P}(\mathcal{L})$ for this example, which is summarized in Table 4.6.

$$\begin{aligned}
\mathcal{D} &= \{\texttt{BST},\ \texttt{Node},\ \texttt{Null},\ \texttt{Integer}\} \\
\mathcal{L} &= \{bst_1,\ n_1,\ n_2,\ n_3,\ n_4,\ null,\ 0,\ 1,\ 5,\ 6\} \\
\gamma(\texttt{BST}) &= \{bst_1\} \\
\gamma(\texttt{Node}) &= \{n_1,\ n_2,\ n_3,\ n_4\} \\
\gamma(\texttt{Null}) &= \{null\} \\
\gamma(\texttt{Integer}) &= \{0,\ 1,\ 5,\ 6\}
\end{aligned}$$

Table 4.6: Summary of domains and instances for the `BST.insert` example

Recall that field types are represented as unions of base types (in this section also called *partitions*). For instance, the type of the field `BST.root` is $\texttt{BST} \to \texttt{Node} \cup \texttt{Null}$, because values of this field can be either instances of `Node` or the `null` constant. That means that all objects of class `Node` plus the constant `null` must be mapped to different atoms, so that it is possible to unambiguously restore the value of the field `root`. This is the basic idea behind the KodkodPart translation: *all literals within any given partition must be mapped to different atoms, whereas literals not belonging to a common partition may share atoms.* The inversion function can then work as follows: for a given atom, first select the correct partition based on the type of the field being restored, then unambiguously select the corresponding literal from that partition. To complete the example, the set of all unary types used in the specification for this example is:

$$\mathcal{T} = \{\texttt{BST},\ \texttt{BST} \cup \texttt{Null},\ \texttt{Node} \cup \texttt{Null},\ \texttt{Null},\ \texttt{Integer}\}$$

Set presented above is set of our partitions. A valid assignment of atoms to literals that uses only 5 atoms, as opposed to 10 which is how many the original translation would use, could be:

$bst_1 \to a_0$	$n_1 \to a_0$	$n_2 \to a_1$	$n_3 \to a_2$	$n_4 \to a_3$
$null \to a_4$	$0 \to a_0$	$1 \to a_1$	$5 \to a_2$	$6 \to a_3$

Limitation of this technique is when the class `Object` is used as a field type or anywhere in the specification. This lead toone big partition containing all literals (because every class is a subclass of `Object`), making the algorithm equivalent to the original translation.

4.2.7.2 Partitioning algorithm

For a given set of base domains $\mathcal{D}$, literals $\mathcal{L}$, and partitions $\mathcal{T}\,(\mathcal{T} = \mathcal{P}\,(\mathcal{D}))$, and a given function $\gamma : \mathcal{D} \to \mathcal{P}\,(\mathcal{L})$ that maps domains to their instance literals, this algorithm produces a set of atoms $\mathcal{A}$ and a function $\alpha : \mathcal{L} \to \mathcal{A}$, such that for every partition p, function α returns different values for all instance literals of p. Formally:

$$(\forall p \in \mathcal{T})\,(\forall l_1, l_2 \in \psi\,(p))\, l_1 \neq l_2 \Longrightarrow \alpha\,(l_1) \neq \alpha\,(l_2)$$

where ψ is a function that for a given partition returns a comprehension of all instance literals of all of its domains:

$$\psi : \mathcal{T} \to \mathcal{P}\,(\mathcal{L})\,;\;\; \psi\,(p) = \{\gamma\,(d) \mid d \in p\}$$

Obviously, a simple bijection would satisfy this specification, but such a solution wouldn't achieve its main goal, which is to minimize the number of atoms, because the number of atoms in this case would be exactly the same as the number of literals. In order to specify solutions that are actually useful, we are going to require the algorithm to produce a result such that the cardinality of $\mathcal{A}$ (i.e. the total number of atoms) is minimal.

It is not immediately clear what the minimal number of atoms ought to be. One might think that no more atoms are required than the number of instances in the largest partition. However, this is not always true. Consider the case shown in Figure 4.3. The largest partitions are `P1` and `P4`, both having 5 literals. On the other hand, any pair of the domains `B`, `C`, and `D` have a partition in common, even though there is no single partition containing them all. They thus form a strongly connected component, and their literals must differ.

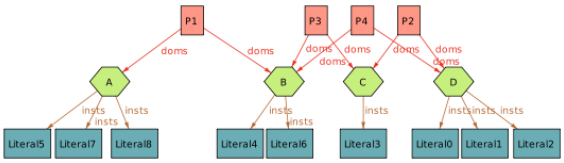


Figure 4.3: KodkodPart: an example where more than the number of literals of the largest partition is needed.

There are 6 literals in total in these three domains, so 5 atoms cannot be enough. As a conclusion, the minimal number of atoms is indeed the number of literals in the largest partitions, but only after all strongly connected domains have been merged into a single partition.

Luckily, cases like the one in Figure 4.3 never happen in Squander, so our implementation of the algorithm doesn't have to search for cliques and merge partitions. The reason this never happens is that domains are always Java classes, and partitions are types used to represent fields. A type of a field is a union type which includes the entire subclass hierarchy of the field's base type. For instance, if `C` and `D` are Java classes, `C` extends `D` (`C <: D`), and some field has declared type `D`, then the type of the field (in the relational world) will be `D` ∪ `C` ∪ `Null`, meaning that `D` ∪ `Null` is never going to be used as a partition for anything.

In summary, the actual implementation inside Squander works as follows:

1. Dependencies between domains are computed. A domain depends on all domains with which it shares a partition. Let the function $\delta : \mathcal{D} \to \mathcal{P}(\mathcal{D})$ express this:

$$\delta(d) = \{d_1 \mid d_1 \neq d \ \wedge ((\exists p \in \mathcal{T})\, d_1 \in p)\}$$

2. The largest partition p_{max} is found such that

$$(\nexists p \in \mathcal{T})\, |\psi(p)| > |\psi(p_{max})|$$

3. For every literal l in $\psi(p_{max})$ an atom a is created, it is added to the universe $\mathcal{A}$ and assigned to l, such that $\alpha(l) = a$. From this point onwards, $\mathcal{A}$ is fixed.

4. For every other partition p iteratively, for all literals $l_p \in p$ that do not already have an atom assigned, a set of possible atoms $\mathcal{A}_{lp}$ is computed and the first value from this set is assigned to l_p. $\mathcal{A}_{lp}$ is computed when atoms corresponding to all literals of all dependent domains is subtracted from $\mathcal{A}$, i.e.:

$$\mathcal{A}_{lp} = \mathcal{A} \setminus \{\alpha(l) \mid l \in \mathcal{L}_d\}, \text{ where}$$

$$\mathcal{L}_d = \{\gamma(d) \mid d \in \mathcal{D}_d\}, \text{ where}$$

$$\mathcal{D}_d = \delta(d_l), \text{ where } d_l \in \mathcal{D} \wedge l_p \in \gamma(d_l)$$

4.3 Differences between Prolog and framework

Thing, which have both classical logical programming in Prolog and programming using framework Squander in common, is defining what to compute instead of how. Both programming paradigms use set of rules for its computation, but here the analogy ends.

If we study Prolog and Squander more in depth, we can see, that there is significant difference in declaring input data. Whereas Squander is strongly connected to its Java data structures, Prolog use set of facts, as was mentioned in chapter 2.

Both Prolog and Squander use set of rules for its computation. If we want to compute some variable in Prolog, language go recursively through all rules, matching the input using rewriting clauses, until all of them are replaced by valid facts and we do not care about order of evaluation (no flow control). Prolog uses three types of formula. *Atomic* formulas declares basic facts, *conditional commands* are used for implication constructs and finally, *target clauses* form query types.

To incorporate Squander into Java annotations are used. For solving some algorithm, annotation `@Requires`,`@Ensures` and `@Modifies` are mostly used. First one, `@Requires`, is used for declaring requirements for input data before method `Squander.exe()` is invoked. Annotation `@Ensures` defines how the data structures should look like after method execution. Last one, `@Modifies`, tells framework class objects or object's fields can be modified during execution. This is very important declaration for the performace of the computation in particular. If we define this annotation optimally, we minimize search space and speed up solving. Calculation itself is done by translating object heap into relations. Then are these relations transformed into SAT problem, which is then solved and the solution, if exists, is reflected back to the heap.

Chapter 5

Implementation of algorithms

5.1 Introduction

As was mentioned before, framework Squander is suitable for implementation of algorithms, which are easy to specify, but difficult to implement imperatively. For this reason I implemented algorithms belonging to NP problems. Problem is said that belongs to the set of NP problems if it is solveable in polynomial time on nondeterministic Turing machine. There is one more expression to explain for description of NP problems - *nondeterministic Turing machine*. First of all we have to define deterministic Turing machine, which for given set of rules prescribes at most one action to be performed for any given situation. Whereas nondetermistic Turing machine is machine, that for given set of rules prescribes more than one action. We can imagine, that in every step of computation is Turing machine cloned in nondeterministic Turing machine.

Most of the algorithm I tried to implement are NPO (especially graph algorithms), but there is a little problem with Squander. It does not support highr-order expressions (subsection 4.1.3). Unfortunately it is not possible to write a specification that says „find a path in the graph such that there is no other path in the graph longer than it“ and solve it with Squander; it is possible, however, to express and solve „find a path in the graph with at least k nodes“, which is computationally as hard as the previous problem, because a binary search can be used to efficiently find the maximum k for which a solution exists. Due that restriction are all of the optimizations algorithms implemented in such way, that there is some treshold level, which solution has to satisfy.

All in all, Squander should be able to solve all NP problems, because it transforms everything into SAT problem, which is NP-complete (Cook's theorem) problem and every NP problem is transferable into NP-complete problem.

5.2 Knapsack Problem

At the beginning when I started „playing“ with Squander, I tried to implement well known algorithm Knapsack. By using Squander's method specification it was pretty easy to declare annotations for computation. Firstly it was necessary to prepare data structures. In this case I implement class `KnapSacck.java`, which has following properties:

- *integer array c* - costs of particular things in knapsack
- *integer array v* - weights of particular things in knapsack
- *integer n* - count of things in knapsack
- *integer capacity* - capacity of knapsack (measured in weight units)

Chapter 6

Comparison

- Způsob, průběh a výsledky testování.
- Srovnání s existujícími řešeními, pokud jsou známy.

Chapter 7

Conclusion

- Zhodnocení splnění cílů DP/BP a vlastního přínosu práce (při formulaci je třeba vzít v potaz zadání práce).
- Diskuse dalšího možného pokračování práce.

Bibliography

[1] Aleksandar Milicevic; MASSACHUSETTS INSTITUTE OF TECHNOLOGY, Department of Electrical Engineering and Computer Science; *Executable Specifications for Java Programs*, September 2010.

[2] Cay Horstmann and Gary Cornell; *Core Java*, 7th edition.

Appendix A

Testování zaplnění stránky a odsazení odstavců

Tato příloha nebude součástí vaší práce. Slouží pouze jako příklad formátování textu.

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta,

která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která

se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta,

která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili? Určitě existuje nějaká pěkná latinská věta, která se k tomuhle testování používá, ale co mají dělat ti, kteří se nikdy latinsky neučili?

Appendix B

Pokyny a návody k formátování textu práce

Tato příloha samozřejmě nebude součástí vaší práce. Slouží pouze jako příklad formátování textu.

Používat se dají všechny příkazy systému LaTeX. Existuje velké množství volně přístupné dokumentace, tutoriálů, příruček a dalších materiálů v elektronické podobě. Výchozím bodem, kromě Googlu, může být stránka CSTUG (Czech Tech Users Group) [**?**]. Tam najdete odkazy na další materiály. Vetšinou dostačující a přehledně organizovanou elektronikou dokumentaci najdete například na [**?**] nebo [**?**].

Existují i různé nadstavby nad systémy TeX a LaTeX, které výrazně usnadní psaní textu zejména začátečníkům. Velmi rozšířený v Linuxovém prostředí je systém Kile.

B.1 Vkládání obrázků

Obrázky se umísťují do plovoucího prostředí `figure`. Každý obrázek by měl obsahovat **název** (`\caption`) a **návěští** (`\label`). Použití příkazu pro vložení obrázku `\includegraphics` je podmíněno aktivací (načtením) balíku graphicx příkazem `\usepackage{graphicx}`.

Budete-li zdrojový text zpracovávat pomocí programu `pdflatex`, očekávají se obrázky s příponou `*.pdf`[1], použijete-li k formátování `latex`, očekávají se obrázky s příponou `*.eps`.[2]

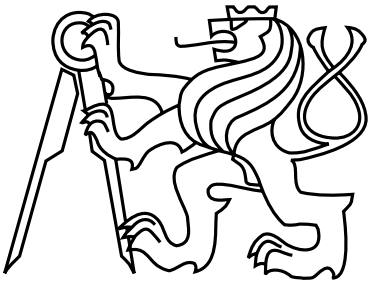


Figure B.1: Popiska obrázku

Příklad vložení obrázku:

```
\begin{figure}[h]
\begin{center}
\includegraphics[width=5cm]{figures/LogoCVUT}
\caption{Popiska obrazku}
\label{fig:logo}
\end{center}
\end{figure}
```

B.2 Kreslení obrázků

Zřejmě každý z vás má nějaký oblíbený nástroj pro tvorbu obrázků. Jde jen o to, abyste dokázali obrázek uložit v požadovaném formátu nebo jej do něj konvertovat (viz předchozí kapitola). Je zřejmě vhodné kreslit obrázky vektorově. Celkem oblíbený, na ovládání celkem jednoduchý a přitom dostatečně mocný je například program Inkscape.

Zde stojí za to upozornit na kreslící programe Ipe [**?**], který dokáže do obrázku vkládat komentáře přímo v latexovském formátu (vzroce, stejné fonty atd.). Podobné věci umí na Linuxové platformě nástroj Xfig.

Za pozornost ještě stojí schopnost editoru Ipe importovat obrázek (jpg nebo bitmap) a krelit do něj latexovské popisky a komentáře. Výsledek pak umí exportovat přímo do pdf.

[1] pdflatex umí také formáty PNG a JPG.

[2] Vzájemnou konverzi mezi snad všemi typy obrazku včetně změn vekostí a dalších vymožeností vám může zajistit balík ImageMagic (http://www.imagemagick.org/script/index.php). Je dostupný pod Linuxem, Mac OS i MS Windows. Důležité jsou zejména příkazy convert a identify.

DTD	construction	elimination
|	`in1|A|B a:sum A B` `in1|A|B b:sum A B`	`case([_:A]a)([_:B]a)ab:A` `case([_:A]b)([_:B]b)ba:B`
+	`do_reg:A -> reg A`	`undo_reg:reg A -> A`
∗, ?	the same like | and + with `emtpy_el:empty`	the same like | and + with `emtpy_el:empty`
R(a,b)	`make_R:A->B->R`	`a: R -> A` `b: R -> B`

Table B.1: Ukázka tabulky

B.3 Tabulky

Existuje více způsobů, jak sázet tabulky. Například je možno použít prostředí `table`, které je velmi podobné prostředí `figure`.

Zdrojový text tabulky B.1 vypadá takto:

```
\begin{table}
\begin{center}
\begin{tabular}{|c|l|l|}
\hline
\textbf{DTD} & \textbf{construction} & \textbf{elimination} \\
\hline
$\mid$ & \verb+in1|A|B a:sum A B+ & \verb+case([_:A]a)([_:B]a)ab:A+\\
&\verb+in1|A|B b:sum A B+ & \verb+case([_:A]b)([_:B]b)ba:B+\\
\hline
$+$&\verb+do_reg:A -> reg A+&\verb+undo_reg:reg A -> A+\\
\hline
$*,?$& the same like $\mid$ and $+$ & the same like $\mid$ and $+$\\
& with \verb+emtpy_el:empty+ & with \verb+emtpy_el:empty+\\
\hline
R(a,b) & \verb+make_R:A->B->R+ & \verb+a: R -> A+\\
 & & \verb+b: R -> B+\\
\hline
\end{tabular}
\end{center}
\caption{Ukázka tabulky}
\label{tab:tab1}
\end{table}
\begin{table}
```

B.4 Odkazy v textu

B.4.1 Odkazy na literaturu

Jsou realizovány příkazem `\cite{odkaz}`.

Seznam literatury je dobré zapsat do samostatného souboru a ten pak zpracovat programem bibtex (viz soubor `reference.bib`). Zdrojový soubor pro `bibtex` vypadá například takto:

```
@Article{Chen01,
  author  = "Yong-Sheng Chen and Yi-Ping Hung and Chiou-Shann Fuh",
  title   = "Fast Block Matching Algorithm Based on
             the Winner-Update Strategy",
  journal = "IEEE Transactions On Image Processing",
  pages   = "1212--1222",
  volume  =  10,
  number  =   8,
  year    = 2001,
}

@Misc{latexdocweb,
  author  = "",
  title   = "{\LaTeX} --- online manuál",
  note    = "\verb|http://www.cstug.cz/latex/lm/frames.html|",
  year    = "",
}
...
```

Pozor: Sazba názvů odkazů je dána BibTEX stylem (`\bibliographystyle{abbrv}`). BibTEX tedy obvykle vysází velké pouze počáteční písmeno z názvu zdroje, ostatní písmena zůstanou malá bez ohledu na to, jak je napíšete. Přesněji řečeno, styl může zvolit pro každý typ publikace jiné konverze. Pro časopisecké články třeba výše uvedené, jiné pro monografie (u nich často bývá naopak velikost písmen zachována).

Pokud chcete BibTEXu napovědět, která písmena nechat bez konverzí (viz `title = "{\LaTeX} --- online manuál"` v předchozím příkladu), je nutné příslušné písmeno (zde

celé makro) uzavřít do složených závorek. Pro přehlednost je proto vhodné celé parametry uzavírat do uvozovek (`author = "..."`), nikoliv do složených závorek.

Odkazy na literaturu ve zdrojovém textu se pak zapisují:

```
Podívejte se na \cite{Chen01},
další detaily najdete na \cite{latexdocweb}
```

Vazbu mezi soubory `*.tex` a `*.bib` zajistíte příkazem `\bibliography{}` v souboru `*.tex`. V našem případě tedy zdrojový dokument `thesis.tex` obsahuje příkaz `\bibliography{reference}`.

Zpracování zdrojového textu s odkazy se provede postupným voláním programů `pdflatex <soubor>` (případně `latex <soubor>`), `bibtex <soubor>` a opět `pdflatex <soubor>`.[3]

Níže uvedený příklad je převzat z dříve existujících pokynů studentům, kteří dělají svou diplomovou nebo bakalářskou práci v Grafické skupině.[4] Zde se praví:

```
...
j) Seznam literatury a dalších použitých pramenů, odkazy na WWW stránky, ...
 Pozor na to, že na veškeré uvedené prameny se musíte v textu práce
 odkazovat -- [1].
Pramen, na který neodkazujete, vypadá, že jste ho vlastně nepotřebovali
a je uveden jen do počtu. Příklad citace knihy [1], článku v časopise [2],
stati ve sborníku [3] a html odkazu [4]:
[1] J. Žára, B. Beneš;, and P. Felkel.
     Moderní počítačová grafika. Computer Press s.r.o, Brno, 1 edition, 1998.
     (in Czech).
[2] P. Slavík. Grammars and Rewriting Systems as Models for Graphical User
     Interfaces. Cognitive Systems, 4(4--3):381--399, 1997.
[3] M. Haindl, Š. Kment, and P. Slavík. Virtual Information Systems.
     In WSCG'2000 -- Short communication papers, pages 22--27, Pilsen, 2000.
     University of West Bohemia.
[4] Knihovna grafické skupiny katedry počítačů:
     http://www.cgg.cvut.cz/Bib/library/
```

[3] První volání `pdflatex` vytvoří soubor s koncovkou `*.aux`, který je vstupem pro program `bibtex`, pak je potřeba znovu zavolat program `pdflatex` (`latex`), který tentokrát zpracuje soubory s příponami `.aux` a `.tex`. Informaci o případných nevyřešených odkazech (cross-reference) vidíte přímo při zpracovávání zdrojového souboru příkazem `pdflatex`. Program `pdflatex` (`latex`) lze volat vícekrát, pokud stále vidíte nevyřešené závislosti.

[4] Několikrát jsem byl upozorněn, že web s těmito pokyny byl zrušen, proto jej zde přímo necituji. Nicméně příklad sám o sobě dokumentuje obecně přijímaný konsensus ohledně citací v bakalářských a diplomových pracích na KP.

... abychom výše citované odkazy skutečně našli v (automaticky generovaném) seznamu literatury tohoto textu, musíme je nyní alespoň jednou citovat: Kniha [?], článek v časopisu [?], příspěvek na konferenci [?], www odkaz [?].

Ještě přidáme další ukázku citací online zdrojů podle české normy. Odkaz na wiki o frameworcich [?] a ORM [?]. Použití viz soubor `reference.bib`. V seznamu literatury by nyní měly být živé odkazy na zdroje. V `reference.bib` je zcela nový typ publikace. Detaily dohledal a dodal Petr Dlouhý v dubnu 2010. Podrobnosti najdete ve zdrojovém souboru tohoto textu v komentáři u příkazu `\thebibliography`.

B.4.2 Odkazy na obrázky, tabulky a kapitoly

- Označení místa v textu, na které chcete později čtenáře práce odkázat, se provede příkazem `\label{navesti}`. Lze použít v prostředích `figure` a `table`, ale též za názvem kapitoly nebo podkapitoly.
- Na návěští se odkážeme příkazem `\ref{navesti}` nebo `\pageref{navesti}`.

B.5 Rovnice, centrovaná, číslovaná matematika

Jednoduchý matematický výraz zapsaný přímo do textu se vysází pomocí prostředí `math`, resp. zkrácený zápis pomocí uzavření textu rovnice mezi znaky `$`.

Kód `$ S = \pi * r^2 $` bude vysázen takto: $S = \pi * r^2$.

Pokud chcete nečíslované rovnice, ale umístěné centrovaně na samostatné řádky, pak lze použít prostředí `displaymath`, resp. zkrácený zápis pomocí uzavření textu rovnice mezi znaky `$$`. Zdrojový kód: `|$$ S = \pi * r^2 $$|` bude pak vysázen takto:

$$S = \pi * r^2$$

Chcete-li mít rovnice číslované, je třeba použít prostředí `eqation`. Kód:

```
\begin{equation}
  S = \pi * r^2
\end{equation}

\begin{equation}
  V = \pi * r^3
\end{equation}
```

je potom vysázen takto:

$$S = \pi * r^2 \tag{B.1}$$

$$V = \pi * r^3 \tag{B.2}$$

B.6 Kódy programu

Chceme-li vysázet například část zdrojového kódu programu (bez formátování), hodí se prostředí `verbatim`:

```
         (* nickname2 *)
Lego> Refine in1
             (do_reg (nickname1 h));
Refine by  in1 (do_reg (nickname1 h))
   ?4 : pcdata
   ?5 : pcdata
          (* surname2 *)
Lego> Refine surname1 h;
Refine by  surname1 h
   ?5 : pcdata
          (* email2 *)
Lego> Refine undo_reg (email1 h);
Refine by  undo_reg (email1 h)
*** QED ***
```

B.7 Další poznámky

B.7.1 České uvozovky

V souboru `k336_thesis_macros.tex` je příkaz `\uv{}` pro sázení českých uvozovek. „Text uzavřený do českých uvozovek.“

Appendix C

Seznam použitých zkratek

2D Two-Dimensional

ABN Abstract Boolean Networks

ASIC Application-Specific Integrated Circuit

⋮

Appendix D

UML diagramy

Tato příloha není povinná a zřejmě se neobjeví v každé práci. Máte-li ale větší množství podobných diagramů popisujících systém, není nutné všechny umísťovat do hlavního textu, zvláště pokud by to snižovalo jeho čitelnost.

Appendix E

Instalační a uživatelská příručka

Tato příloha velmi žádoucí zejména u softwarových implementačních prací.

Appendix F

Obsah přiloženého CD

Tato příloha je povinná pro každou práci. Každá práce musí totiž obsahovat přiložené CD. Viz dále.

Může vypadat například takto. Váš seznam samozřejmě bude odpovídat typu vaší práce. (viz [**?**]):

Na GNU/Linuxu si strukturu přiloženého CD můžete snadno vyrobit příkazem:
`$ tree . >tree.txt`
Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případne index.html apod.) musí být rovněž zřejmé, jak programy instalovat, spouštět a jaké požadavky mají tyto programy na hardware.

Adresář **text** musí obsahovat soubor s vlastním textem práce v PDF nebo PS formátu, který bude později použit pro prezentaci diplomové práce na WWW.

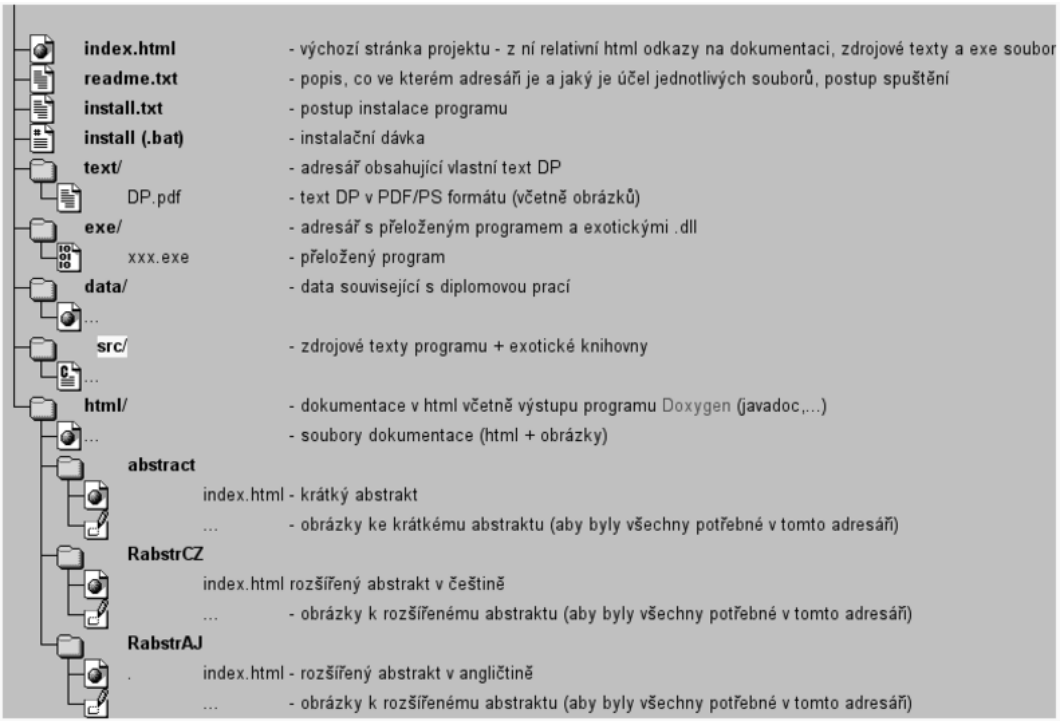


Figure F.1: Seznam přiloženého CD — příklad