

# Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním 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 13, 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 effectiveness of implementing and computation performance of the program.

Work shows how was this framework used for implementing set of algorithms, especially NP-complete 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Logical programming principles</b>	<b>3</b>
2.1	Logical programming paradigm . . . . .	3
2.1.1	Facts . . . . .	4
2.1.2	Rules . . . . .	5
2.2	Terminology . . . . .	6
2.3	Evaluation in Prolog . . . . .	7
<b>3</b>	<b>Meaning of annotations</b>	<b>9</b>
<b>4</b>	<b>Framework description</b>	<b>13</b>
4.1	Introduction . . . . .	13
4.1.1	Overview . . . . .	13
4.1.2	Architecture . . . . .	14
4.1.3	Applications . . . . .	14
4.2	Execution of Squander . . . . .	15
4.2.1	Kodkod . . . . .	15
4.2.2	JFSL . . . . .	16
4.2.2.1	JFSL expressions . . . . .	16
4.2.2.2	JFSL annotations . . . . .	17
<b>5</b>	<b>Implementation of algorithms</b>	<b>19</b>
<b>6</b>	<b>Comparison</b>	<b>21</b>
<b>7</b>	<b>Conclusion</b>	<b>23</b>
<b>A</b>	<b>Testování zaplnění stránky a odsazení odstavců</b>	<b>27</b>
<b>B</b>	<b>Pokyny a návody k formátování textu práce</b>	<b>33</b>
B.1	Vkládání obrázků . . . . .	33
B.2	Kreslení obrázků . . . . .	34
B.3	Tabulky . . . . .	35
B.4	Odkazy v textu . . . . .	36
B.4.1	Odkazy na literaturu . . . . .	36

B.4.2	Odkazy na obrázky, tabulky a kapitoly . . . . .	38
B.5	Rovnice, centrovaná, číslovaná matematika . . . . .	38
B.6	Kódy programu . . . . .	39
B.7	Další poznámky . . . . .	40
B.7.1	České uvozovky . . . . .	40
<b>C</b>	<b>Seznam použitých zkratk</b>	<b>41</b>
<b>D</b>	<b>UML diagramy</b>	<b>43</b>
<b>E</b>	<b>Instalační a uživatelská příručka</b>	<b>45</b>
<b>F</b>	<b>Obsah přiloženého CD</b>	<b>47</b>

# List of Figures

2.1	Expression evaluation in Prolog . . . . .	8
4.1	Architecture diagram . . . . .	14
B.1	Popiska obrázku . . . . .	34
F.1	Seznam přiloženého CD — příklad . . . . .	48



# List of Tables

4.1	Unary expressions supported by JFSL . . . . .	17
4.2	Quantified expressions supported by JFSL . . . . .	17
B.1	Ukázka tabulky . . . . .	35





# 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 imperative 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 themselves 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 illustrate 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 queries. 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 have been time consuming. For this reason Prolog provides conditional expressions - *rules*.

By investigating facts is possible to derive new rule based on logical or factual context. That knowledge allows 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 expresses, 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 the 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  $A : - P_1, P_2, \dots, P_n$  where  $P_1, P_2, \dots, P_n$  are atomic formulas (e.g. `brother(X, Y) :- parent(O, Y), parent(O, X), man(X).`)
- *target clauses* - query type  $(? - C_1, C_2, \dots, C_n$  where  $C_1, C_2, \dots, C_n$  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  $G_1, G_2, \dots, G_m$  in Prolog consists of following steps:

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

If is found clause  $C$  in form

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

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

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

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

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

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

Then substitution  $S$  is done in order to make new list of targets, so new form is:

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

By recursive invocation procedure **Evaluate** is evaluated this sequence of targets. If this evaluation ends successfully, previous evaluation is treated as also successful. If this evaluation does not end successfully, 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 evaluation 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).

```

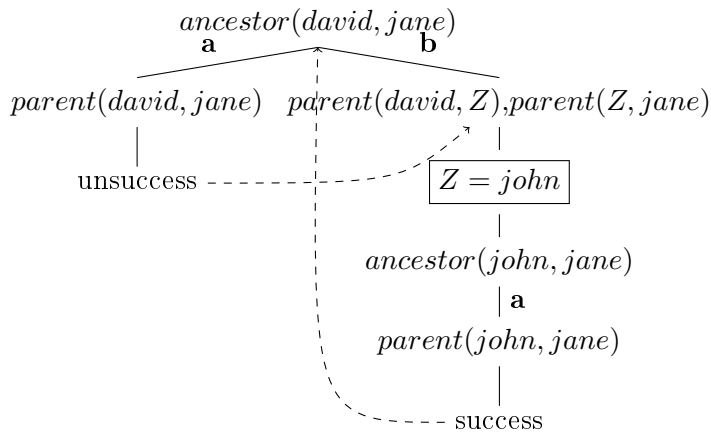


Figure 2.1: Expression evaluation in Prolog

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



## Chapter 3

# Meaning of annotations

Since Java 5.0 release, general purpose of annotation was ability to define and use own annotation types. The facility consists of a syntax for declaring annotation types, a syntax for annotating declarations, APIs for reading annotations, a class file representation for annotations.

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.

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 public @interface Length {  
2     int max();  
3     int min();  
4 }
```

Listing 3.1: Length annotation

This is simple annotation 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 with no 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

Annotations can be annotated itself. Such annotations are called *meta-annotations*. E.g. using `@Target(ElementType.METHOD)` indicates, that annotation type should be used to annotate only 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:

```

1 @Entity
2 public class Role {
3     @Length(max = 50)
4     String name;
5
6     @Length(max = 150)
7     String description;
8
9     @Test
10    public void showMeTheFunny() {
11        System.out.println("Here you have funny");
12    }
13
14    public void foo() {
15        System.out.println("Foo");
16    }
17
18    public void bar() {
19        System.out.println("Bar");
20    }
21 }

```

Listing 3.6: Extended class Role

```

1 public class RunTests {
2     public static void main(String[] args) throws Exception {
3         int passed = 0, failed = 0;
4         for (Method m : Class.forName(args[0]).getMethods()) {
5             if (m.isAnnotationPresent(Test.class)) {
6                 try {
7                     m.invoke(null);
8                     passed++;
9                 } catch (Throwable ex) {
10                    System.out.printf("Test %s failed: %s %n", m, ex.getCause());
11                    failed++;
12                }
13            }
14        }
15
16        System.out.printf("Passed: %d, Failed %d%n", passed, failed);
17    }
18 }

```

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()  
8                         > ((Length) a).max())  
9                         throw new Exception(  
10                             "Unacceptable length of field "+f.getName()+" of class Role");  
11                 }  
12             }  
13         }  
14     }  
}
```

Listing 3.8: Test of entity Role annotation

## 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

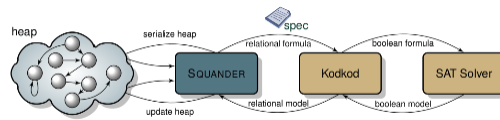


Figure 4.1: Architecture diagram

Further will be explained in detail 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  $\rightarrow$  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 the 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 off-the-shelf SAT solver to search for satisfying solution, which is reflected back if found.

When are the relations in Kodkod created, they are untyped, meaning that every relation can potentially 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 methods 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.



Table 4.1: Unary expressions supported by JFSL

Operator	Description
#	set cardinality
no	„no“ multiplicity (empty)
lone	„lone“ multiplicity (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 4.2: Quantified expressions supported by JFSL

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

#### 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 annotation 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.

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

1. Assembling all relevant 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

## Chapter 5

# Implementation of algorithms

Popis implementace/realizace se zaměřením na nestandardní části řešení.



## 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.



## 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,













## 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  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ . 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. Většinou dostačující a přehledně organizovanou elektronikou dokumentaci najdete například na [? ] nebo [? ].

Existují i různé nadstavby nad systémy  $\text{T}_{\text{E}}\text{X}$  a  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ , 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`<sup>1</sup>, použijete-li k formátování `latex`, očekávají se obrázky s příponou `*.eps`.<sup>2</sup>

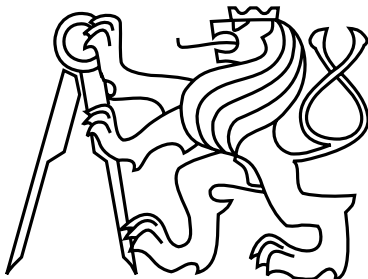


Figure B.1: Popiska obrázku

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 kreslicí 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.

<sup>1</sup>pdflatex umí také formáty PNG a JPG.

<sup>2</sup>Vzájemnou konverzi mezi snad všemi typy obrazku včetně změn velikostí 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
	$\text{in1} A B \text{ a:sum } A \ B$ $\text{in1} A B \text{ b:sum } A \ B$	$\text{case}([\_ :A]a)([\_ :B]a)ab:A$ $\text{case}([\_ :A]b)([\_ :B]b)ba:B$
+	$\text{do\_reg}:A \rightarrow \text{reg } A$	$\text{undo\_reg}:reg \ A \rightarrow A$
*, ?	the same like   and + with <code>empty_el:empty</code>	the same like   and + with <code>empty_el:empty</code>
$R(a,b)$	$\text{make\_R}:A \rightarrow B \rightarrow R$	$a: R \rightarrow A$ $b: R \rightarrow B$

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
| & \texttt{in1}|A|B \texttt{ a:sum } A \ B \\
& \texttt{in1}|A|B \texttt{ b:sum } A \ B & \texttt{case}([\_ :A]a)([\_ :B]a)ab:A \\
& & \texttt{case}([\_ :A]b)([\_ :B]b)ba:B \\
\hline
+ & \texttt{do\_reg}:A -> reg A & \texttt{undo\_reg}:reg A -> A \\
\hline
*, ? & the same like | and + \\
& with \texttt{empty\_el:empty} & the same like | and + \\
& & with \texttt{empty\_el:empty} \\
\hline
R(a,b) & \texttt{make\_R}:A->B->R & a: R -> A \\
& & 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 Bib<sub>TEX</sub> stylem (`\bibliographystyle{abbrv}`). Bib<sub>TEX</sub> 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 Bib<sub>TEX</sub>u 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>`.<sup>3</sup>

Níže uvedený příklad je převzat z dříve existujících pokynů studentům, kteří dělají svou diplomovou nebo bakalářskou práci v Grafické skupině.<sup>4</sup> 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],
statí 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/
```

<sup>3</sup>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.

<sup>4</sup>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 ukázkou citací online zdrojů podle české normy. Odkaz na wiki o frameworkch [? ] 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 centrovane 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 `equation`. Kód:

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

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

je potom vysázen takto:

$$S = \pi * r^2 \quad (\text{B.1})$$

$$V = \pi * r^3 \quad (\text{B.2})$$

## 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 : pcddata
  ?5 : pcddata
      (* surname2 *)
Lego> Refine surname1 h;
Refine by surname1 h
  ?5 : pcddata
      (* 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 příloženého CD

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

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

Na GNU/Linuxu si strukturu příloženého CD můžete snadno vyrobit příkazem:

```
$ tree . >tree.txt
```

Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případně 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.

	<b>index.html</b>	- výchozí stránka projektu - z ní relativní html odkazy na dokumentaci, zdrojové texty a exe soubor
	<b>readme.txt</b>	- popis, co ve kterém adresáři je a jaký je účel jednotlivých souborů, postup spuštění
	<b>install.txt</b>	- postup instalace programu
	<b>install (.bat)</b>	- instalační dávka
	<b>text/</b>	- adresář obsahující vlastní text DP
	DP.pdf	- text DP v PDF/PS formátu (včetně obrázků)
	<b>exe/</b>	- adresář s přeloženým programem a exotickými .dll
	xxx.exe	- přeložený program
	<b>data/</b>	- data související s diplomovou prací
	...	
	<b>src/</b>	- zdrojové texty programu + exotické knihovny
	...	
	<b>html/</b>	- dokumentace v html včetně výstupu programu Doxygen (javadoc,...)
	...	- soubory dokumentace (html + obrázky)
	<b>abstract</b>	
	index.html	- krátký abstrakt
	...	- obrázky ke krátkému abstraktu (aby byly všechny potřebné v tomto adresáři)
	<b>RabstrCZ</b>	
	index.html	- rozšířený abstrakt v češtině
	...	- obrázky k rozšířenému abstraktu (aby byly všechny potřebné v tomto adresáři)
	<b>RabstrAJ</b>	
	index.html	- rozšířený abstrakt v angličtině
	...	- obrázky k rozšířenému abstraktu (aby byly všechny potřebné v tomto adresáři)

Figure F.1: Seznam přiloženého CD — příklad