RWTH Aachen

Department of Computer Science
Technical Report

An Open Framework for Data–Flow Analysis in Java

Markus Mohnen

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2002-08

RWTH Aachen · Department of Computer Science · April 2002

An Open Framework for Data–Flow Analysis in Java

Markus Mohnen

Lehrstuhl für Informatik II, RWTH Aachen, Germany
mohnen@informatik.rwth-aachen.de

Abstract. We describe work in progress on a framework for data–flow based program analysis. By using this framework, researchers and developers can easily implement analyses, test their correctness, and evaluate their performance. In addition, the framework allows the definition of intraprocedural analyses for Java Virtual Machine (JVM) code on a high level of abstraction.

The framework is provided as a set of APIs for Java. Through the extensive use of Java interface concept, we established an open framework: For instance, specific implementations of abstract domains can easily be used in our framework.

1 Introduction

Data–flow analysis (DFA) [17] is the basic technique used for the static analysis of (imperative) programs. Research project in this field are typically concerned with the development of specific analyses or the improvement of the basic technique.

In this paper, we describe the current state of a framework for DFA–based program analysis. By using this framework, researchers and developers can easily implement specific analyses, test their correctness, and evaluate their performance. In addition, the framework has special support for Java Virtual Machine (JVM) code [11]. It allows the definition of intraprocedural analyses for JVM code on a higher level of abstraction.

Our framework is provided as a collection of APIs for Java [8]. Besides the portability of Java programs, the major reason for this choice was the interface concept of Java, which makes it possible to base programs on *properties of classes* instead of classes. By using this concept, we established an *open* framework: For instance, specific implementations of abstract domains can easily be used in our framework.

We have successfully used the framework in several projects: `JOpt` [9] and `JoGa`, Java class file optimisers, and [12], a tool for reverse engineering of communication protocols of Erlang [2] applications. Currently, we use the framework for an empirical study of basic–block–graph and flow–graph performance.

Structure of this article. In the next section, we give an overview over the framework. Section 3 describes the API for representing graphs. The next section shows how we model mathematical structures, e.g. lattices. The basic data–flow API is described in Section 5, followed by a description of the additional level of abstraction for the implementation of abstract interpretation based intraprocedural analyses for Java Virtual Machine (JVM) code in Section 6. Finally, Section 7 concludes the paper.

2 Overview

In this section, we give an overview over the structure of the framework and briefly describe each component. Each component is a separate API implemented by a Java package. The packages structure the framework in layers.

`de.rwth.graph`: This package is used for representing graphs and graph colourings. Besides the obvious use for DFA, we also use this package for the visualisation of abstract domains through Hasse diagrams. Graphs represented with this package can be easily visualised using `graphviz` [7].

`de.rwth.domains`: To model mathematical structures like sets, partially ordered sets, domains, and functions, this package provides an interface hierarchy. Finite and infinite structures can be modelled using this hierarchy. Since the rest of our framework solely depends on this interface hierarchy, it is possible to use all means for implementing an abstract domain, e.g. bit vectors or BDDs [5, 4].

Of course, each of the mathematical structures has a set of mathematical properties that must be fulfilled, e.g. the commutativity of the join operation. A class implementing a structure cannot be forced to conform with the properties by the means of a programming language. Hence, a developer of a mathematical structure must take care of this. Therefore, the package contains two means for debugging: (a) as already mentioned, a method for constructing the Hasse diagram of a partially ordered set and (b) a method which checks if an implementation conforms with the assumed mathematical properties. It turned out that both (a) and (b) are especially useful for debugging the implementation of domains.

The interfaces in this package are not *orthogonal*, i.e. some methods can be simulated by the combination of others. For instance, the interface for partially ordered sets has methods for “less–than”, “less–or–equal–than”, and “equals”. To reduce the effort required for implementation, some of the interfaces are equipped with a set of *default implementations* of methods. They are especially interesting for implementing methods of an interface in a canonical way (like above). In these cases, an implementation may be obtained by implementing the base methods and use default implementations of the other methods. Inspired by this work, we have proposed an extension of Java for providing default implementations in interfaces [13, 15].

`de.rwth.domains.templates`: This package contains a set of classes implementing some of the interfaces from `de.rwth.domains`, like simple sets, bit vector lattices, and kill–gen bit vector functions. In addition, the package contains classes for constructing implementations of structures, like dual partially ordered sets, flat complete lattices, and composed functions.

`de.rwth.dfa`: The core of this package is an implementation of the classical iterative algorithm [17, 1, 16] for DFA. Since it uses `de.rwth.graph` for the graph representation, it is not important where the graphs come from: They can be flow–graph, basic–block graphs (see [10] for comments on the adequateness of this choice), or something completely different. Since we use the interfaces from `de.rwth.domains` to model initial values, transfer functions, and solutions for nodes, any implementation of domains and functions may be used. The package supports computation of greatest and least fixed point, as well as backward and forward flow direction.

`de.rwth.dfa.jvm`: This package adds an additional level of abstraction above the low–level `de.rwth.dfa` package. It provides an easy way to implement abstract interpretation based intraprocedural analyses for Java Virtual Machine (JVM) code [11]. An implementation can describe an abstraction by implementing the corresponding interface. Therfore, it must provide methods for computing the domain, the initial value associated with an instruction, and the transfer function associated with an instruction. It addition, it must determine if the analysis is existential or universal and if it is a backward or a forward analysis.

For solving a data flow problem consisting of an abstraction and a JVM class, the package provides three choices: Either by using `de.rwth.dfa` on basic block graphs or flow graphs, or by using our new graph–free solver [14]. To access JVM class files, this package uses the excellent Byte Code Engineering Library [3]. The solvers can handle full JVM code, including exceptions. It is also possible to use factorised graphs [6], i.e. graphs without edges for possible exceptions. Having more than one solver at hand is very useful for debugging abstractions, since some errors can be found by comparing the results of different solvers.

In addition, the package gives access to timing data of the solver and to (an approximation of) the memory usage. This is useful for evaluating the performance of a solver or an abstraction.

The JVM specification [11] defines that a valid method must allow to determine this size of the computation stack for each instruction. If abstractions model the computation stack of the JVM, it is often necessary to know this size. For instance, constant folding propagation is an analysis of this kind. Therfore, the package contains a class which provides the stack size.

Finally, the package contains two analyses complete analyses for JVM code: “constant folding propagation” and “live variables”. They can be applied directly to JVM class files and have been used in `JOpt` [9].

Fig. 1 Structure of the Framework

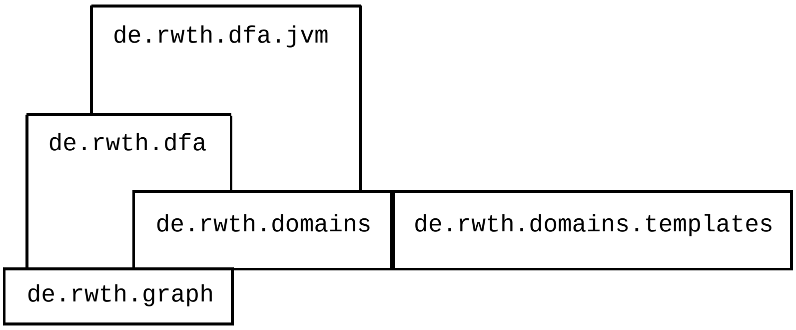
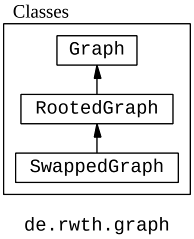


Fig. 1 summarises the layered structure of the framework. By using interfaces to a large extend, our framework is open for extensions. In the following sections, we describe each layer in more detail.

3 The Graph Layer

The package `de.rwth.graph` implements an API for representing directed graphs. Its very simple top–level class hierarchy consists of the main `Graph` class, the class `RootedGraph` for graphs with distinguished root and leaf nodes, and an auxiliary class `SwappedGraph` which transforms a graph to a new graph where the role of nodes and edges are swapped.

Fig. 2 Inheritance Diagram of `de.rwth.graph`



The main class `Graph` allows the creation of graphs through the use of *inner classes*: After creating an object `g` of class `Graph`, new nodes can be added by creating inner objects of the graph through `g.new Node()`. In the same way, edges are created as inner objects of nodes, i.e. for objects `n1` and `n2` of class `Graph.Node`, an edge originating in `n1` and ending in `n2` may be created by `n1.new Edge(n2)`. Of course, nodes and edges can also be labelled, where a label can be any `Object`.

By using this approach, we have avoided top–level classes for nodes and edges, and we can assure that at any time of program execution, nodes are always associated with a graph and edges are always associated with two nodes.

In addition to labels, the API is capable of representing colourings of graphs node. Therefore, we also use an inner class: Given an object `g` of class `Graph`, a node colouring may be created by `g.new NodeColouring()`. Different colourings are independent and the number of colourings is not limited. With a colouring represented as object of class `Graph.NodeColouring`, we can associate any object as colour for a node with the method `setColour`.

Both graphs and node colourings represented with this package can be easily visualised using `graphviz` [7]: The method `toString()` creates a text which can be used directly as input for `dot`. For instance, the program fragment in Fig. 3(a) creates a graph and prints in representation (Fig. 3(b)) which can be transformed to the picture in Fig. 3(c) using the `dot` program which is part of `graphviz`.

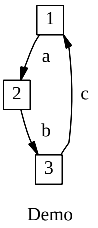
Fig. 3 Example for Graph Visualisation

```
...
Graph g=new Graph("Demo");
Graph.Node n1=g.new Node("1");
Graph.Node n2=g.new Node("2");
Graph.Node n3=g.new Node("3");
n1.new Edge(n2,"a");
n2.new Edge(n3,"b");
n3.new Edge(n1,"c");
System.out.println(g);
...
```

(a) Program Fragment

```
digraph "Demo" {
  label="\nDemo"
  n0 [shape=box,label="1"];
  n0 -> n1 [label="a"];
  n1 [shape=box,label="2"];
  n1 -> n2 [label="b"];
  n2 [shape=box,label="3"];
  n2 -> n0 [label="c"];
}
```

(b) Output

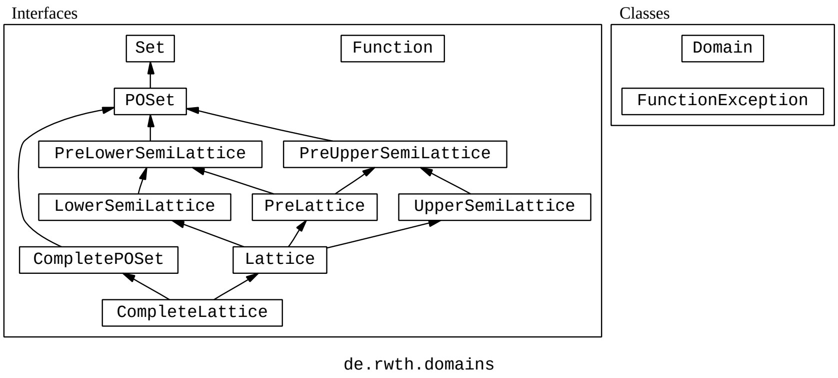


(c) Visualisation

4 The Domain Layer

For modelling the mathematical structures needed for DFA, our framework provides two APIs: the package `de.rwth.domains`, which defines the interface hierarchy shown in Fig. 4, and the package `de.rwth.domains.templates`, which contains implementations for standard domains and domain constructors like bit vectors, Cartesian product, or lifted partially ordered sets. A list of the current content of the package `de.rwth.domains.templates` can be found in Appendix A.

Fig. 4 Inheritance Diagram of `de.rwth.domains`



By this separation and since the rest of our framework solely depends on the interface hierarchy in the package `de.rwth.domains`, it is possible to use all means for implementing an abstract domain, e.g. bit vectors or BDDs [5, 4]. Therfore, an implementation must only extend the appropriate interface.

4.1 Mathematical Structures

Since interfaces do not contain implementations, the package mainly defines the existence of certain operations and predicates of a mathematical structure as methods of a corresponding implementing class. The API allows the elements of all structures to be any object of Java's root class `Object`.

`Set`: This interface defines the membership predicate and equality check for the elements as the methods `boolean isElement(Object)` and `equals(Object, Object)`.

`POSet`: The interface `POSet` for representing partially ordered sets has additional methods `le(Object, Object)` and `lt(Object, Object)`, both returning `boolean`, for representing the "less–than" and "less–or–equal–than" predicates.

`LowerSemiLattice`, `UpperSemiLattice`: In addition to `POSet`, these interfaces have the following operations: `meet(Object,Object)` and `join(Object, Object)`, both returning `Object`.

`CompletePOSet`, `CompleteLattice`: The interface `CompletePOSet` adds a constant `Object bottom()`, and the interface `CompleteLattice` adds the constants `Object bottom()` and `Object top()`.

In addition to these structures, the package contains the interface `Function` for representing functions. Instances of this interface must be associated with a domain and range, which are both instances of `Set`. A instance of the interface `Function` can be applied to an argument with the method `Object apply(Object x)`. If an invalid argument is passed as argument, the method may throw an exception of class `de.rwth.domains.FunctionsException`.

In addition to these representations of mathematical operations and predicates, the root interface `Set` of the package defines methods for accessing the set as a whole: The method `size()` returning a `long` gives the number of elements for finite sets and `-1` for infinite sets. For finite sets, the method `iterator()` returns an object of class `java.util.Iterator`, which can be used to iterate through all elements of the set. These methods are mainly intended for debugging implementations (see below). Since they are useless for finite sets, the API introduces the notion of a *set skeleton*: The underlying idea is that often infinite structures have some kind of regularity which can be represented by a finite structure. For instance, the infinite lattice for constant folding propagation[1] in Fig. 5(a) can be represented using the skeleton in Fig. 5(b). Here, we assume that the element 42 represents the equivalence class of all numbers and that all operations are compatible with this representation.

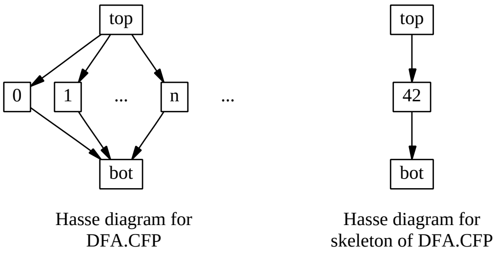
Each of the mathematical structures has a set of mathematical properties that must be fulfilled, e.g. the totality of the operations or the commutativity of the join operation. All these constraints are defined by the API specification, but a class implementing a

---

[1] This lattice can be constructed using the classes from `de.rwth.domains.templates` in the following way:

```
package DFA;
import de.rwth.domains.templates.*;
public class CFP extends FlatCompleteLattice {
  public CFP () { super(new NumberSet()); }
}
```

Fig. 5 Example for Set Skeletons



(a) Infinite CFP Lattice (b) Finite CFP Skeleton

structure cannot be forced to conform with the properties by the means of a programming language. Hence, a developer of a mathematical structure must take care of this. However, the API provides debugging of the compliance with the properties.

The difference between those interfaces with `Pre` as prefix and those without lies in these additional constraints: While the methods of classes implementing interfaces without prefix must be defined for all elements, the methods of classes implementing interfaces with prefix must not be total. This is useful since some domains constructors like `LiftedCompleteLattice` need this.

4.2 Default Implementations

The interfaces in this package are not *orthogonal*, i.e. some methods can be simulated by the combination of others. For instance, an invocation `s.le(o1,o2)` can always be simulated by `s.lt(o1,o2)||s.equals(o1,o2)` if the implementing class of `s` conforms with the constraints defined by the API.

To reduce the effort required for implementation, some of the interfaces of the package are equipped with a set of *default implementations* of methods. They are especially interesting for implementing methods of an interface in a canonical way (like above). In these cases, an implementation may be obtained by implementing the base methods and use default implementations of the other methods. Inspired by this work, we have proposed an extension of Java for providing default implementations in interfaces [13, 15]. Currently, the default implementations in the framework are still provided in the style of [13].

4.3 Debugging Implementations

Implementing an interface from this package can be subtle, especially in view of the mathematical constraints. However, the rest of the framework depends on the compliance with the constraints: For instance, the data flow algorithm might not terminate if the constraints are violated by an implementing class.

Therefore, the package contains the class `Domains`, containing two means for debugging, which can be used for finite structures and infinite structures with a finite skeleton:

1. Given an object `po` of an implementation of (at least) the interface `POSet`, the (static) method invocation `Domains.hasseDiagram(po)` constructs an object of class `de.rwth.graph.Graph` with the Hasse diagram of the partially ordered set or its underlying skeleton.
2. Given an object `s` of an implementation of any interface I, the (static) method invocation `checkProperties(s)` checks if the objects behaviour conforms with the assumed mathematical properties for I.

Although both means are crude in the sense that they only perform an exhaustive check of the implementation, we found that they are very helpful for the detection of errors.

5 The Data–Flow Layer

The core of this package is the class `DataFlowSolver`, which contains an implementation of the classical iterative algorithm [17, 1, 16]. For creating an instance of this class, the following parameters must be passed to the constructor:

`de.rwth.graph.RootedGraph g`: The graph on which the algorithm should run.

`de.rwth.graph.Graph.NodeColouring inits`: The initial values at each node, represented as a colouring of the nodes of the graph `g`. All colours must be elements of the same instance of `de.rwth.domains.Lattice`.

`de.rwth.graph.Graph.NodeColouring fns`: The transfer functions associated with each node, also represented as a colouring of the nodes of the graph `g`. The colour of each node must be an instance of `de.rwth.domains.Function` such that domain and range are the same `de.rwth.domains.Lattice` as for the initial values.

`boolean isAll`: This parameter determines whether the universal (greatest fixed point) solution or the existential (least fixed point) solution is computed.

`boolean forward`: This parameter determines the direction of flow to be considered: Forward flow means that the values computed are associated with entry point of a node, and that the transfer functions determine the value at the exit point of a node. For backward flow, this is vice versa. Here, entry and exit points can be imagined as the points where all incoming/outgoing edges join.

Altogether, these parameters determine a data–flow problem. Given an instance of this class, the solution of the associated data–flow problem can be computed using the method `solve()`, which returns the solution as node colouring of the graph `g`. Of course, the colouring of each node is from the same `de.rwth.domains.Lattice` as for the initial values.

Since this class only depends on `de.rwth.graph` for the graph representation, it is not important where the graphs come from: They can be flow–graph, basic–block graphs, or something completely different. Since we only use the interfaces from the package `de.rwth.domains` to model initial values, transfer functions, and solutions for nodes, any implementation of domains and functions may be used.

6 The JVM Layer

This package (`de.rwth.dfa.jvm`) adds an additional level of abstraction above the `de.rwth.dfa` package. It allows to implement abstract interpretation based intraprocedural analyses for Java Virtual Machine (JVM) code [11]. Therfore, it provides the interface `Abstraction` and the class `Solver`.

Fig. 6 Interface `de.rwth.dfa.jvm.Abstraction`

```
package de.rwth.dfa.jvm;

import de.fub.bytecode.generic.*;
import de.rwth.domains.*;

public interface Abstraction {
  public Lattice getLattice();

  public static final int DIRECTION_FORWARD = 0;
  public static final int DIRECTION_BACKWARD = 1;
  public int getDirection();

  public static final int QUANTIFIER_ALL = 0;
  public static final int QUANTIFIER_EXISTS = 1;
  public int getQuantifier();

  public Object getInitialValue(InstructionHandle ih, boolean isEntry);
  public Object getInitialValue(InstructionHandleVector ihs, boolean isEntry);

  public Function getAbstract(InstructionHandle ih);
  public Function getAbstract(InstructionHandleVector ihv);
}
```

By implementing the interface `Abstraction` in Fig. 6, a class describes an abstract interpretation of (a single method of) JVM code:

- The abstract domain is the instance of `de.rwth.domains.Lattice` returned by the method `getLattice()`.
- Direction (forward/backward) and quantification (universal $\simeq$ greatest fixed point / existential $\simeq$ least fixed point) are determined by the methods `getDirection()` and `getQuantifier()`.
- The initial value at an instruction is computed by an implementing class through the method `getInitialValue(InstructionHandle,boolean)`. Of course, it must be an element of the abstract domain. The instruction for which the initial value is computed is passed to `getInitialValue` as first argument of the class `de.fub.bytecode.generic.InstructionHandle` from the Byte Code

Engineering Library [3]. In addition, the second argument determines if the instruction is an *entry point*:

  - For forward flow, these are the first instruction of the method and the first instructions of the exception handlers.
  - For backward flow, the entry points are all instructions which leave the method, i.e. `ATHROW` or one of the `RETURN` instructions.

  An implementation can ignore the arguments and always return the same value, e.g. top of bottom element in case of a complete lattice.

- The abstractions of JVM instructions are modelled as instances of the interface `de.rwth.domains.Function`. They must all have the abstract domain returned by `getLattice()` as domain and range and are computed by the method `getAbstract(InstructionHandle)`.

In addition, the interface contains methods for computing initial values and functions that take as first argument an object of class `InstructionHandleVector`. These objects are used for representing basic blocks. The package contains a default implementation, which computes these values from the corresponding methods for objects of class `InstructionHandle`, by using function composition in the case of `getAbstract`. However, an implementation might do it differently: For instance, if the functions are kill–gen functions, a single kill–gen function can be computed instead of the composition.

Given an implementation of `Abstraction` and a JVM method, an object of class `Solver` can be created. Its method `getSolution()` can be used to compute the solution of the associated data–flow problem as an array of elements from the abstract domain, one for each instruction of the method. Here, the package provides three choices:

Flow graphs: The flow graph (or single instruction graph) for the given method is determined and the solution is computed using `de.rwth.dfa`.

Basic block graphs: The basic block graph for the given method is determined and the solution is computed using `de.rwth.dfa`.

Abstract Execution: The solution is computed without an additional graph representation and without use of `de.rwth.dfa` by abstract execution of the program (see [14] for a more detailed discussion of this approach).

In all cases, the class `Solver` can handle full JVM code, including exceptions. For the graph–based solvers, it is also possible to use factorised graphs [6], i.e. graphs without edges for possible exceptions.

Having more than one solver at hand is very useful for debugging abstractions, since some errors, especially in the implementation of the underlying abstract domain or the abstraction, can be found by comparing the results of different solvers.

In addition, the package gives access to timing data of the solver, to the memory usage[2], and the number of iterations needed. This data are available after executing the method `getSolution()`.

---

[2] Since Java does not give access to precise memory information, an approximation of the memory usage is provided.

The JVM specification [11] defines that a valid method must allow to determine this size of the computation stack for each instruction. If abstractions model the computation stack of the JVM, it is often necessary to know this size. For instance, constant folding propagation is an analysis of this kind. Therfore, the package contains an abstract class `AbstractSSDependingAbstraction` which can be used as super class for implementations of analyses of this kind.

Finally, the package contains two analyses complete analyses for JVM code: "constant folding propagation" and "live variables". They can be applied directly to JVM class files and have been used in `JOpt` [9].

7 Conclusions

We have described a framework for data–flow based program analysis. It is provided as a set of five APIs for Java sharing the domain prefix `de.rwth`: `graph` can be used for representing graphs, `domains` defines an interface hierarchy for mathematical structures, `domains.templates` contains implementations for standard domains and domains constructors, `dfa` contains an implementation of the classical iterative algorithm for DFA, and finally `dfa.jvm` allows to implement abstract interpretation based intraprocedural analyses for Java Virtual Machine (JVM) code on a high level of abstraction.

By using this framework, researchers and developers can easily implement specific analyses, test their correctness, and evaluate their performance. The APIs provide the ability to debug implementations of mathematical structures and gives access to timing and memory usage of the solvers.

Through the extensive use of Java interface concept, we established an open framework: For instance, specific implementations of abstract domains can easily be used in our framework.

Since this paper describes work in progress, there are many directions for further research:

- We plan to extend the API `de.rwth.domains.templates` with more domain constructors.
- Currently, the default implementations in the framework are provided in the style of [13]. We plan to change this to the more efficient style described in [15].
- By using interface in even more places, we could achieve an even more open and reusable framework: For instance, graphs could be modelled as interfaces (with the current implementation as one possibility) which would allow to study on–the–fly techniques where the graph is created on demand.
- Although the `de.rwth.domains.templates` API simplifies the task of defining domains, it is still necessary to explicitely write Java classes for implementing domains. It would be interesting to develop a graphical user interface for this task.
- Very much in the sense of [18] it would be interesting to design additional APIs for model checking: We could definitely reuse the `de.rwth.graph` API (modified as described above) and maybe the `de.rwth.domains` API.

The framework is available on request from the author.

References

[1] A.V. Ahos, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[3] B. Bokowski and M. Dahm. Byte Code Engineering. In C. H. Cap, editor, *Java-Informations-Tage (JIT)*, Informatik Aktuell. Springer–Verlag, 1998. See also at http://bcel.sourceforge.net/.

[4] K. Brace, R. Bryant, and L. Rudell. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, June 1990.

[5] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35, August 1986.

[6] J.D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, volume 24.5 of *Software Engeneering Notes (SEN)*, pages 21–31. ACM Press, 1999.

[7] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233, September 2000.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series. Addison Wesley, 2nd edition, 2000.

[9] M. Jansen. Optimierung von Java Class Dateien. Diploma Thesis, RWTH Aachen, 2000. in German; URL: http://www-i2.informatik.rwth-aachen.de/~markusj/jopt/.

[10] J. Knoop, D. Koschützki, and B. Steffen. Basic-Block Graphs: Living Dinosaurs? In K. Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction (CC)*, number 1383 in Lecture Notes in Computer Science, pages 65–79. Springer–Verlag, 1998.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley, 2nd edition, 1999.

[12] M. Joußen. Reverse Engineering von Kommunikationsprotokollen aus verteilten Erlang Anwendungen. Diploma Thesis, RWTH Aachen, 2001. in German.

[13] M. Mohnen. Interfaces with Skeletal Implementations in Java. In *Object-Oriented Technology – ECOOP 2000 Workshop Reader*, number 1964 in Lecture Notes in Computer Science, pages 295–296. Springer–Verlag, 2000.

[14] M. Mohnen. A Graph–Free Approach to Data–Flow Analysis. In R. N. Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction (CC)*, number 12304 in Lecture Notes in Computer Science, pages 46–61. Springer–Verlag, 2002.

[15] M. Mohnen. Interfaces with Default Implementations in Java. Technical Report AIB-2002-09, RWTH Aachen, April 2002. http://aib.informatik.rwth-aachen.de/2002/2002-09.ps.gz.

[16] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[17] S. S. Muchnick and N. D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice–Hall, 1981.

[18] D. Schmidt and B. Steffen. Program Analysis as Model Checking of Abstract Interpretations. In G. Levi, editor, *Proceedings of the 5th International Symposium on Static Analysis (SAS)*, number 1503 in Lecture Notes in Computer Science. Springer–Verlag, 1998.

A Content of Package `de.rwth.domains.templates`

This is the current list of public classes in the package:

BitVectorElement		de.rwth.domains.CompleteLattice
BitVectorLattice		de.rwth.domains.CompleteLattice
ComposedFunction		de.rwth.domains.Function
ConstantFunction		de.rwth.domains.Function
DualPOSet		de.rwth.domains.POSet
FlatCompleteLattice		de.rwth.domains.CompleteLattice
FunctionSet		de.rwth.domains.Set
FunctionPOSet	FunctionSet	de.rwth.domains.POSet
FunctionCompletePOSet	FunctionPOSet	de.rwth.domains.CompletePOSet
IdentityFunction		de.rwth.domains.Function
KillGenBitVectorFunction		de.rwth.domains.Function
LiftedPOSet		de.rwth.domains.POSet
LiftedCompletePOSet	LiftedPOSet	de.rwth.domains.CompletePOSet
LiftedCompleteLattice	LiftedPOSet	de.rwth.domains.CompleteLattice
NumberSet		de.rwth.domains.Set
SimpleSet		de.rwth.domains.Set
IntegerPOSet	SimpleSet	de.rwth.domains.POSet
StackSet		de.rwth.domains.Set
StackPOSet	StackSet	de.rwth.domains.POSet
StackPreLattice	StackPOSet	de.rwth.domains.PreLattice
SumSet		de.rwth.domains.Set
SumPOSet	SumSet	de.rwth.domains.POSet
TabledFunction		de.rwth.domains.Function
TrivialPOSet		de.rwth.domains.POSet
TupleElement		
TupleSet		de.rwth.domains.Set
TuplePOSet	TupleSet	de.rwth.domains.POSet
TupleCompletePOSet	TuplePOSet	de.rwth.domains.CompletePOSet
TupleLattice	TuplePOSet	de.rwth.domains.Lattice
TupleCompleteLattice	TupleLattice	de.rwth.domains.CompleteLattice

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

95-11 * M. Staudt / K. von Thadden: Subsumption Checking in Knowledge Bases
95-12 * G.V. Zemanek / H.W. Nissen / H. Hubert / M. Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology
95-13 * M. Staudt / M. Jarke: Incremental Maintenance of Externally Materialized Views
95-14 * P. Peters / P. Szczurko / M. Jeusfeld: Business Process Oriented Information Management: Conceptual Models at Work
95-15 * S. Rams / M. Jarke: Proceedings of the Fifth Annual Workshop on Information Technologies & Systems
95-16 * W. Hans / St. Winkler / F. Sáenz: Distributed Execution in Functional Logic Programming
96-1 * Jahresbericht 1995
96-2 M. Hanus / Chr. Prehofer: Higher-Order Narrowing with Definitional Trees
96-3 * W. Scheufele / G. Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
96-4 K. Pohl: PRO-ART: Enabling Requirements Pre-Traceability
96-5 K. Pohl: Requirements Engineering: An Overview
96-6 * M. Jarke / W. Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
96-7 O. Chitil: The ς-Semantics: A Comprehensive Semantics for Functional Programs
96-8 * S. Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
96-9 M. Hanus (Ed.): Proceedings of the Poster Session of ALP'96 — Fifth International Conference on Algebraic and Logic Programming
96-10 R. Conradi / B. Westfechtel: Version Models for Software Configuration Management
96-11 * C. Weise / D. Lenzkes: A Fast Decision Algorithm for Timed Refinement
96-12 * R. Dömges / K. Pohl / M. Jarke / B. Lohmann / W. Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
96-13 * K. Pohl / R. Klamma / K. Weidenhaupt / R. Dömges / P. Haumer / M. Jarke: A Framework for Process-Integrated Tools

96-14 * R. Gallersdörfer / K. Klabunde / A. Stolz / M. Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996

96-15 * H. Schimpe / M. Staudt: VAREX: An Environment for Validating and Refining Rule Bases

96-16 * M. Jarke / M. Gebhardt, S. Jacobs, H. Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization

96-17 M. Jeusfeld / T. X. Bui: Decision Support Components on the Internet

96-18 M. Jeusfeld / M. Papazoglou: Information Brokering: Design, Search and Transformation

96-19 * P. Peters / M. Jarke: Simulating the impact of information flows in networked organizations

96-20 M. Jarke / P. Peters / M. Jeusfeld: Model-driven planning and design of cooperative information systems

96-21 * G. de Michelis / E. Dubois / M. Jarke / F. Matthes / J. Mylopoulos / K. Pohl / J. Schmidt / C. Woo / E. Yu: Cooperative information systems: a manifesto

96-22 * S. Jacobs / M. Gebhardt, S. Kethers, W. Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

96-23 * M. Gebhardt / S. Jacobs: Conflict Management in Design

97-01 Jahresbericht 1996

97-02 J. Faassen: Using full parallel Boltzmann Machines for Optimization

97-03 A. Winter / A. Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

97-04 M. Mohnen / S. Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

97-05 * S. Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

97-06 M. Nicola / M. Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

97-07 P. Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

97-08 D. Blostein / A. Schürr: Computing with Graphs and Graph Rewriting

97-09 C.-A. Krapp / B. Westfechtel: Feedback Handling in Dynamic Task Nets

97-10 M. Nicola / M. Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

97-13 M. Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

97-14 R. Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

97-15 G. H. Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

98-01 * Jahresbericht 1997

98-02 S. Gruner/ M. Nagel / A. Schürr: Fine-grained and Structure-oriented Integration Tools are Needed for Product Development Processes

98-03 S. Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

98-04 * O. Kubitz: Mobile Robots in Dynamic Environments

98-05 M. Leucker / St. Tobies: Truth — A Verification Platform for Distributed Systems

98-07 M. Arnold / M. Erdmann / M. Glinz / P. Haumer / R. Knoll / B. Paech / K. Pohl / J. Ryser / R. Studer / K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects

98-08 * H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen

98-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien

98-10 * M. Nicola / M. Jarke: Performance Modeling of Distributed and Replicated Databases

98-11 * A. Schleicher / B. Westfechtel / D. Jäger: Modeling Dynamic Software Processes in UML

98-12 * W. Appelt / M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web

98-13 K. Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation

99-01 * Jahresbericht 1998

99-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version

99-03 * R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager

99-04 M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

99-07 Th. Wilke: CTL+ is exponentially more succinct than CTL

99-08 O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures

2000-01 * Jahresbericht 1999

2000-02 Jens Vöge / Marcin Jurdzinski: A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-04 Andreas Becks / Stefan Sklorz / Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach

2000-05 Mareike Schoop: Cooperative Document Management

2000-06 Mareike Schoop / Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling

2000-07 * Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages

2000-08 Thomas Arts / Thomas Noll: Verifying Generic Erlang Client-Server Implementations

2001-01 * Jahresbericht 2000

2001-02 Benedikt Bollig / Martin Leucker: Deciding LTL over Mazurkiewicz Traces

2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig / Martin Leucker / Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus

2001-05 Benedikt Bollig / Martin Leucker / Thomas Noll: Regular MSC Languages

2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic

2001-07 Martin Grohe / Stefan Wöhrle: An Existential Locality Theorem

2001-08 Mareike Schoop / James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling

2001-09 Thomas Arts / Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs

2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures

2001-11 Klaus Indermark / Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung

2002-01 * Jahresbericht 2001

2002-02 Jürgen Giesl / Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems

2002-03 Benedikt Bollig / Martin Leucker / Thomas Noll: Generalised Regular MSC Languages

2002-04 Jürgen Giesl / Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting

2002-05 Horst Lichter / Thomas von der Maßen / Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines

2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.