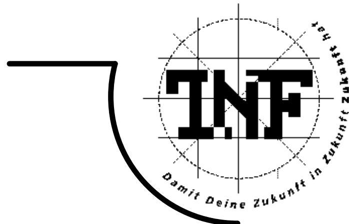
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



Automated Formal Static Analysis and Retrieval of Source Code

MASTER'S THESIS

for obtaining the academic title

MASTER OF SCIENCE

in

INTERNATIONALER UNIVERSITÄTSLEHRGANG: ENGINEERING & MANAGEMENT

Composed at *ISI – Hagenberg*

Handed in by: *Mădălina Eraşcu, 0656410*
Finished on: *July 2008*

Scientific Management:

Name of adviser: *a.Univ.-Prof. Dr. Tudor Jebelean*

Grade:

Name of the grader: *a.Univ.-Prof. Dr. Tudor Jebelean*

Linz, July 2008

Automated Formal Static Analysis and Retrieval of Source Code

Master's Thesis

Mădălina Eraşcu

Advised by:

a.Univ.-Prof. Dr. Tudor Jebelean

International School for Informatics
Johannes Kepler University, Linz, Austria

Abstract

In this thesis two approaches to source code analysis are theoretically investigated and implemented in two prototype systems: formal static analysis and retrieval. An integration of the formal static analysis prototype into the code search prototype is designed.

The formal static analysis method is based on forward symbolic execution and functional semantics. It systematically generates the verification conditions which are necessary for program correctness.

We formalize the notions of syntax, semantics, partial correctness and termination of imperative recursive programs in a purely logic manner. The partial correctness and the termination principles are expressed in the underlying theory of programs. The termination property is expressed as an induction principle depending on the structure of the program with respect to recursion and it plays a central role in the existence and the uniqueness of the function computed by the program, without which the total correctness formula is trivial due to inconsistency of the assumptions. The method is implemented in a verification condition generator (`FwdVCG`) which generates the proof obligations that insure the correctness of the program. A formulae simplifier is then applied and reduces them to [system(s)] of equalities and/or inequalities.

Another achievement in this thesis is the integration of a source code retrieval engine *Mindbreeze Code Search* into an information retrieval system *Mindbreeze Enterprise Search*. The integration required slight modifications of the information retrieval system architecture and components: a database, a source code custom crawler, a new data source representing the source code files category were integrated. The context interface of the Query Service was enhanced to provide context items specific to source code files category.

One of the components of the custom crawler is a tagger which extracts rapidly programming languages constructs. The crawled source code is made available for retrieval by being structured and inserted into an index or database. A new data source, representing the source code files, was integrated into the system by deploying on the server a context provider, category icon and category descriptor source code category specific. Moreover, we had to provide to this new data source, category specific icons and menus. To this aim, we improved the Query Service Context Interface with category icons and menus specific for the source code category specific.

Keywords : program verification, symbolic execution, forward reasoning, functional semantics, *Theorema*, tagging, parsing, crawling, indexing, *Mindbreeze Enterprise Search*.

Acknowledgements

Foremost, I would like to thank Tudor Jebelean and Bruno Buchberger.

From Professor Jebelean, my thesis advisor, I learned new and interesting things from the courses he taught and the discussions we had. Moreover, I was nicely surprised by his proposal to continue my master thesis work in a Ph.D. thesis. Thank you very much for your clear explanations, challenging ideas, permanent encouragements, help and support!

I thank Professor Buchberger for accepting me in the *Theorema* group, for giving me the possibility to finish my master studies at JKU Linz in the first series of the International School for Informatics graduates, for all he tried and did for the ISI students.

I am very grateful to the Mindbreeze Software GmbH for the financial support during the second year of master studies and also for giving me the opportunity to work on an interesting project, as a part of my master thesis. Special thanks to Jakob Praher, from the Mindbreeze company, for his help in understanding the technologies they use and for sharing his implementation ideas with us. Also the discussions with my colleague, Laci Lukacs, helped me during the work on this project. Thank you Laci!

Many thanks to the members of *Theorema* group for their constructive critics, promising ideas and all discussions from the seminar.

I would like to thank my professors from Timişoara, Viorel Negru and Dana Petcu, for their help and support.

The „guilty" persons for beginning my master studies at JKU were Florin Fortiş, my bachelor thesis supervisor, and Laura Kovács; they advised me to apply for an Erasmus-Socrates scholarship at JKU, told me about RISC and the interesting things the people are working on here. Thank you!

My RISC colleagues Manuel Kauers and Veronika Pillwein helped me very much with LaTeX and give me some technical advices regarding my thesis. I wish to thank them too.

I can not forget to mention my RISC friends, especially Dana, Camelia and Mădălina and all the friends I made here in Austria. With them I spent a lot joyful moments.

The spiritual and psychological support came from God and my family. Vă multumesc buni, Iulică, tati şi mai ales ţie Iti pentru că aţi crezut întotdeauna în mine!

Contents

1 Introduction **1**

2 Program Verification by Symbolic Execution in *Theorema* **7**

2.1 Background . . . 7

2.1.1 Program Verification . . . 7

2.1.2 Symbolic Execution . . . 9

2.1.3 Program Verification in the *Theorema* System . . . 12

2.2 Forward Symbolic Execution in the *Theorema* System . . . 13

2.2.1 Basic Notions . . . 13

2.2.2 A Meta-Logic for Reasoning about Imperative Programs . . . 13

2.3 The Simplification of the Verification Conditions . . . 18

2.4 Implementation and Examples . . . 21

2.4.1 Greatest Common Divisor using the Euclidean Algorithm . . . 22

2.4.2 Solving First Degree Equations . . . 24

2.4.3 Solving Second Degree Equations . . . 25

3 Code Search Integration Facility into *Mindbreeze Enterprise Search* **27**

3.1 Background . . . 27

3.1.1 Information Retrieval . . . 27

3.1.2 Source Code Retrieval . . . 28

3.1.3 Mindbreeze Enterprise Search . . . 32

3.2 Integration of Mindbreeze Code Search into Mindbreeze Enterprise Search . . . 32

3.2.1 Crawling and Indexing . . . 32

3.2.2 Query Service Context Provider Interface Enhancements . . . 41

3.2.3 CodeSearch Data Source Integration . . . 45

3.2.4 Mindbreeze Code Search - Use Cases . . . 48

4 Conclusions **53**

References **56**

Chapter 1

Introduction

From the hand-proof from the 1950's until the nowadays sophisticated automated tools, the main goal was the same: proving program (algorithm, software) correctness.

Program testing and debugging lost from the very beginning the war in being the suitable techniques for proving the program correctness. The only hope remained in the *program verification*. But because ***„Everything interesting about the behavior of programs is undecidable."*** [paraphrase of H.G. Rice, 1953] (from A. Mœller slides on *An Introduction to Analysis and Verification of Software*), the task that had to be solved by the program verification techniques is a very challenging one.

We are interested in the imperative program verification using a formal static analysis method which uses an axiomatic approach in the Hoare triple style; we are given the input (I_P) and the output (O_P) specification for an imperative program P and we want to show that the program fulfils its specification. We approach this problem using automated theorem proving, namely we automatically generate the verification conditions which arise from the program analysis and try to prove them automatically.

Following the ideas of the axiomatic approach used for the generation of the verification conditions (symbolic execution and forward reasoning techniques) and for computing the program function (functional semantics method), we developed, in a logical manner, the syntax, the semantics, the partial correctness and the termination for the imperative programs which contain `Return` statements, assignments (including recursive calls) and conditionals (`If` with one and two branches).

For expressing these notions we use, besides the underlying theory of the programs – object theory, the environment of a meta–theory, in which the notions about reasoning about programs are expressed. In this way, one could reason also about the system which would automatize these theoretical notions.

The program termination problem is known to be undecidable, but some gains were achieved in the last years in [CS02], [B. 06a], were the main goal was to find termination proofs for programs (*liveness* property) and in [HJMS], [B. 06b], were the emphasis is to prove that a program is not error prone (*safety* property).

We approach the termination property in a purely logical manner, in the underlying theory of programs, without requiring any model of computations of programs. For example, in the case of

the primitive recursive programs, we formulated the termination principle as an induction principle developed from the shape of recursion. The termination principle insures also the existence and the uniqueness of the function computed by the program.

The method is implemented and tested in a prototype system composed by two parts: (i) a *verification conditions generator* (`FwdVCG`) which generates proof obligations that are checked for validity and (ii) *a simplifier*. The prototype `FwdVCG` is built on top of the computer algebra system *Mathematica* and uses the *Theorema* procedural language for writing imperative programs.

With more impact in industry, we integrated in an information retrieval system (*Mindbreeze Enterprise Search*) a facility for code search retrieval with the purpose of helping the programmer in software development (code reusability, code comprehension, code quality, etc.).

The integration was made by building a custom crawler, adding a database besides the existing index, enhancing the Query Interface.

One of the components of the custom crawler is a tagger which crawls source code files from the repositories subject to code retrieval. The necessity of the tagger usage came up from the desire that we want to have, in a short amount of time, structured information from the files of the repositories. It came up that the capabilities of the tagger are limited; more powerful semantic information, like the relationships between the programming languages objects, were impossible to be retrieved. And this feature urged to be implemented because it adds supplementary functionality to the system (context actions, query by reformulation). At this purpose, a parser was used.

The next step was to merge the information obtained from tagging and parsing into an uniform representation. Files with *XML* representation were used for this purpose, which after parsed by a *DOM* parser and filtered by the *Mindbreeze Filter Service*, are stored in the index or in the database (more precisely the relevant information – hit-types and their meta-data).

From the user point of view, the Query Service Context Interface had to be enhanced with context icons and menus specific to the source code elements. This enhancement was equivalent to building a context provider, source code files specific, which after being deployed on the server together with a category icon and category descriptor, makes possible the user interaction with the new feature of the system.

The verification conditions generator can be used for verifying the correctness of programs which were crawled. For more effective operation, we have to:

- either include in our approach the `while` – loops, either to built a translator such that every loop is unfolded to `If` and `recursive call` statements.

- keep in mind that in the parsing process the methods and their specification are not siblings nodes in the abstract syntax tree and an algorithm which associates to a method the right specification has to be built. Moreover, the parser has support only for *Java* programming language.

- built a theorem prover such that the proving process to be completely done for various classes of problems.

Contributions of the Thesis

The statical approach for checking the program correctness is based on forward symbolic execution and functional semantics, but, additionally, gives formal definitions in a meta-theory for the meta-level functions (describing the semantics, the partial correctness and the termination condition of the programs) and predicate (defining the syntax of the programs) which characterize the object computation.

While most of the work which treats the problem of forward symbolic execution and functional semantics describes textually these principles ([BEL75]), we formalize them in predicate logic because the formalization gives the possibility of reflective view of the system by describing how the data (the state, the program, the verification conditions) are manipulated and by introducing a causal connection between the data and the status of computation (a certain statement of the program determines a certain computation of the function describing the semantics, of the functions generating the verification conditions and of the termination condition to be performed).

We mention that our approach keeps the verification process very simple: the verification conditions insuring the partial correctness are first order logic formulae, the termination principle is expressed as an induction principle in the underlying logic of the programs, without introducing any model of computation like, for example, the Scott fix-point theory ([LSS84]).

Approaches for solving the correctness of symbolic executed programs exists due to [LSS84, Top75, Deu73]; for the imperative programs containing *assignments*, *conditionals* and `while` *loops bounded* on the number of times they are executed, the proof of correctness is given by analyzing the verification conditions on each branch of the program. For the programs containing loops with unbounded number of iterations, the branches of the program are infinite and have to be traversed inductively in order to prove/disprove their correctness. In the inductive traversal of the tree, additional assertions have to be provided, called *inductive assertions*. But the inductive assertions method applies to partial correctness proofs ([LSS84]), while our approach concentrates in proving the total correctness of programs.

For code search engines there is a lot of expertise both theoretical and practical ([GAL^{+}06]), where the main issues are emphasized as being the accuracy and the reuse of the results. Some results were already achieved in the prototypes like *Maracatu* (as part of RiSE – Reuse in Software Engineering), but the main objective is to provide a framework for helping organizations in all aspects of implementing a reuse program.

Regarding this problem, our task was to integrate in an information retrieval system (*Mindbreeze Enterprise Search (MES)*) the facility of source code retrieval. This was relatively easy achievable due to the loosely-coupled architecture of the *MES system*.

Our tasks consisted in: (i) the construction of a custom crawler, specialized for source code files crawling, (ii) the integration of a new data source, representing source code files category, and (iii) the enhancing of Query Service Context Interface such that it provides context items (context icons, context menus, context actions) specific to the new category integrated.

One of the components of the custom crawler is a tagger which crawls the relevant information (hit – types and their metadata) from the source code files repositories. The tagger is applied first because it structures the information fast and has support for many programming languages. Its disadvantage (the metadata retrieved have not very powerful semantic meaning) required the integration of another component, namely a parser. But the information obtained from tagging

and parsing has not an uniform structure. Files with XML structure solved this problem.

Next step was to make persistent the information crawled by inserting it in a index or a database.

From the tagger side, all the information crawled is inserted into the index. Therefore, the source code files are tagged, the output files from the tagging process are transformed into an *XML* structure, a *DOM* parser was built such that the information from the *XML* files is inserted into the index.

The integration of the new data source, representing the source code files, was possible by deploying on the server side a category icon, category descriptor and a context provider. The deployment was realized by using an existing *Mindbreeze* tool called `mesextension`, we only had to provide it their implementation.

The result set matching the query of the user and involving source code category has to have category items specific to this category. For this purpose, the Query Service Context Provider Interface was enhanced to provide category icons, menus and actions source code files category specific.

All these features are available in a prototype implementation.

Structure of the Thesis

The **Chapter 2** presents the theoretical basis and the implementation of a method for formal static verification of the imperative programs as follows.

In **Section 2.1** we introduce and motivate the necessity of the software verification. We are interested in proving/disproving the software correctness using a static analysis method based on logical inference rules where the main challenge is the automatization of the whole verification process. The informal description of the method and, additional, references to existing symbolic execution systems, are presented in **Section 2.1.2**.

Section 2.1.3 contains the presentation of the *Theorema* environment and of the existing approaches for program verification in the system, environment constituting the basis of the verification conditions generator and of the simplifier described in sections 2.3 and 2.4.

In **Section 2.2** we describe the method for static program analysis, first by introducing some preliminary notions, like the syntax and semantics of the program, and then how the partial correctness and termination can be insured. All these notions are completely formalized in predicate logic. The formalization is done in a [meta –] logic expressive enough for allowing reflection about the system which automates the method.

The method presented in **Section 2.2** is implemented in a prototype environment written in *Mathematica*, using the *Theorema* working language. The prototype (verification conditions generator and simplifier) is exemplified in sections 2.3 and 2.4.

The **Chapter 3** introduces the theoretical basis and the implementation of the system specialized in code search retrieval.

In 3.1 and 3.2 we present the benefits of information retrieval, in particular code search retrieval: we present a short survey on the techniques developed during the years for the accuracy and fast retrieval of data and on some systems for source code retrieval which implement the latest research approaches.

Next section starts by describing the underlaying system (*Mindbreeze Enterprise Search* (MES)) into which the code search facility will be integrated; it is an information retrieval system based on a client-server technology, whose loosely coupled architecture allows the integration of custom components. Thus, the integration of a new component, for making possible the search within the data source representing source code files, can be done easily: a source code crawler has to be developed (**Section 3.3.2**), the data source representing the crawled data has to be deployed on the server (**Section 3.3.4**) and the context interface has to be enhanced with context actions specific for the new integrated data source (**Section 3.3.3**).

User interface and use cases illustrating the retrieval process from simple text search to data-drilling operations and also the re-query by context actions operation are presented in **Section 3.3.5**.

In the last chapter (**Chapter 4**) we conclude by emphasizing what was achieved in this thesis and by presenting some ideas for further work, both from program analysis and retrieval points of view.

Chapter 2

Program Verification by Symbolic Execution in *Theorema*

2.1 Background

2.1.1 Program Verification

The desire of the software developers was, is and will be to write programs without bugs. For the achievement of this goal they try to combine the following techniques: programming language design, debugging, testing.

Programming language design represents the main step in writing a correct software. The facilities of the nowadays programming languages (type systems, abstract data types, inheritance and encapsulation for object oriented programming, etc.) provide writing software at high level of abstraction and implicitly reduce the number of possible errors.

By *debugging*, one can reduce the number of bugs in a software program in a systematic way such that it behaves as expected.

Testing is an empirical step towards the software verification. It is performed with the intention of finding software bugs but it can not provide the certainty of software correctness.

Nighter of these techniques, nor their combination give a software correctness proof.

Program verification is, instead, the technique which insures or disproves the correctness of a computer program with respect to a specification.

We are interested in verifying programs using *theorem proving*, more specifically *automated theorem proving*.

2.1.1.1 Theorem Proving in Program Verification

We approach the problem of program (algorithm) verification from the formal static point of view, that is, we analyze the program without executing it. More precisely, we use an approach in the Hoare ([Hoa69]) like style.

In this formal system, the program correctness problem is formulated as follows: using a calculus involving the program statements, the verification conditions which arise from program analysis are generated and proven to be theorems. They are logical formulae and state the fact that

the postcondition of the program is a logical consequence of a set of statements (the *axioms* and *hypotheses*) which were collected during the program analysis.

Theorem proving becomes interesting for industrial software verification only if the techniques developed for the verification process are implemented in a computer program which means that the theorems concerned in the verification process are automated. Obviously, the systems which would automatize the verification process (called automated theorem provers), have to be adapted and combined with other application systems ([Sch00]).

The trends in automated theorem proving for program verification are: development of methods for proving that the algorithms used in software are correct, the high degree of automation of the correctness proof techniques, invariant generation, integration with model checking and decision procedures, etc.

2.1.1.2 Program Verification Software Systems. State-of-the-Art

KeY

The KeY system is a software tool integrating design, implementation, formal specification (UML/ OCL and JML), and formal verification (dynamic logic, decision procedures) of object-oriented software (with focus on Java) as seamlessly as possible.

The KeY system uses a deductive verification component (which also can be used as a standalone prover) which combines symbolic execution, first-order reasoning, arithmetic simplification and external decision procedures for the program proving purposes (with focus on programs complying the Java Card standard).

The desired features of the system, as expressed in [ABH^{+}07] are: the separate integration (as much as possible) of an automated and interactive prover, a better user interface for presentation of proof states and rule application, a higher level of automation.

Spec#

The *Spec#* is a static program verifier codenamed Boogie developed at Microsoft Research. It generates the verification conditions for a Spec# (a C# extension with non-null types, checked exceptions and throw clauses, methods specification, invariants) program and uses an automatic theorem prover (*Simplify*) as well as a set of decision procedures developed internally (*Zap*) to prove/disprove their validity ([LM08]). Proving/disproving the verification conditions yields to correct/incorect program.

The main contribution of the system is that allows reasoning about invariants also when callbacks are present.

ESC/Java (or more recently ESC/Java2)

ESC/Java2 ([KPC]) is a static analysis tool used for finding run-time errors in JML-annotated Java programs. The programmer can annotate the program with a special kind of comments called *pragmas* (kinds of specification) in order to help the programmer in finding the errors quickly. The goal of the system is not concentrated in rigorous program verification, but more in helping the programmer to test and review the code.

ESC/Java2 translates JML-annotated Java code into series of proof obligations and then uses the theorem prover *Simplify* to prove these obligations. The proving process is completely hidden from the user.

From the practical point of view, *ESC/Java2* is incomplete with respect to the Java language: no support for multi-threading).

Theoretically, it is not sound nor complete. Incompleteness is due to the lack of power of the current theorem prover, unsoundness results from the lack of support for integer overflow or multi-threaded execution.

2.1.2 Symbolic Execution

2.1.2.1 Method Description

Before the symbolic execution technique was used in program proving, the existing approaches used the notion of a domain for the variables involved in the program execution.

In program proving using symbolic execution ([Kin75]), the numeric values for the input variables are supplied with symbolic values and the expressions standing for the program variables are represented by symbolic expressions.

Three notions are involved in the program proving using this approach: state, path condition, program counter.

A *state* contains symbolic expressions for each variable used in the program.

A *path condition* represents a set of assumptions that the inputs must satisfy in order to reach the respective branch. Basically, the path condition is a conjunction of predicates upon the symbolic values of the input variables. There exists a path condition corresponding to each program path.

The *program counter* determines which statement will be executed next.

For the generation of the path condition it turned out that *forward reasoning* (used in the the majority of the systems implementation – see **Section 2.1.2.2**) is more suitable than *backward reasoning* ([How73]), because it follows naturally the execution of a program.

The *symbolic execution tree* can also characterize the execution of a procedure: each program statement represents a node and each transition between statements is a directed arc. From nodes representing conditionals there is more than one arc leaving it; the node associated with the first statement has not incoming arcs, the terminal node has no outgoing arcs.

If the procedure contains `while` loops whose number of iterations depend on the input variables then the associated symbolic execution tree is infinite.

Symbolic execution is used in many approaches for program analysis: path domain checking, partition analysis, program reduction, program testing, but we are interested in *applying symbolic execution in program proving*.

2.1.2.2 Symbolic Execution Systems

The early symbolic execution systems (EFFIGY, SELECT, ATTEST, Interactive Programming System, etc.) use theorem proving for checking the correctness of programs.

EFFIGY ([Kin75]) follows the rules and principles of the symbolic execution approach. The language supported by EFFIGY is a simple imperative programming language. The input values for the programs can be both symbolic and concrete values.

For handling module calls, EFFIGY uses the lemma approach: once the module is proved to be correct with respect to its specification, then the specification is used each time the module is invoked.

EFFIGY generates verification conditions for each path of the program whose truth value is determined using a formulae simplifier ([Kin70]) and a theorem prover ([KF70]).

SELECT ([BEL75]), developed after EFFIGY, incorporates the features of a theorem prover. Programs written in SELECT use a Lisp subset statements.

SELECT provides: (i) an expressions simplifier and (ii) a proof that an output specification of the program is fulfilled by a path of the program when the variables are supplied with symbolic values.

During the program analysis, the program paths are analyzed for feasibility thus the initial number of paths is reduced significantly.

The limitation of the SELECT system is that it does not have the minimum requirements for program abstraction, namely module calls. The implementation just replaces during the program analysis each module call with the module's source code itself, approach named macro-expansion. This approach is not reasonable because it produces an explosion of paths to be analyzed.

SELECT solves also the loop's problem (by specifying the number of times a loop is executed) and the arrays indexed by symbolic values problem (by introducing virtual paths). The drawback of the virtual path approach is that increases the flow graph of a program: a new branch is added for each element of the array.

ATTEST ([Cla76]) was used for the verification of Fortran programs thus it was a step forward for the usage of symbolic execution in larger programs. The system uses an inequality checker for formulae simplifications.

Interactive Programming System ([ADL$^+$79]) is used for testing programs in a subset of imperative language statements. Although the language is not too rich, it was integrated into Olivetty system and used for commercial purposes.

The system uses a typed programming language, the programs are structured in modules. Each module is composed by a declaration of types, procedures and data, and can contain calls of other modules.

The system is interactive, allowing the user to save the current state of the program, to choose a certain branch in the execution of the program, to restore a previously saved state.

The path condition (symbolic expression) is simplified and proved using provers for the theory of equality and propositional calculus, and an algebraic simplifier.

More recently developed symbolic execution systems combine various techniques in order to verify the correctness of industrial software.

For example, **DART** (**Directed Automated Random Testing**) combines random testing and dynamic test generation using symbolic execution to verify C programs.

The system generates all the paths involved in a program using a simplified approach introduced in [God97]: all the paths in the program are tested systematically. The drawbacks of the symbolic execution, namely imprecision of static analysis and theorem provers, are avoided in the implementation of *DART*. *DART* supplies the symbolic input values with the results obtained by the dynamic test generation and checks if they have the expected result on the program.

A program is seen as a black-box and from here the big advantage of *DART*: it does not require any information about the termination of the modules involved.

An important practical applicability of symbolic execution for verifying safety-critical software systems ([CPDGP01]) is ***SAVE (Symbolic execution Aided Verification Environment)***. It handles with programs written in Safer-C ([Hat95]) programming language, a subset of C. SAVE has the following components:

1. *Execution Model Generator* takes as input a program written in *Safer–C* and builds the corresponding execution model, with the user intervention. The module is basically a symbolic executor with features similar to *EFFIGY* and the model is, actually, a symbolic execution tree that can be visualized due to a publicly available graph layout tool, *Dotty*. The nodes of the tree represent states.

2. *PDL Property Checker* takes as input is an expression in *PDL (Path Description Language)* language which is tested over the model generated by the *Execution Model Generator*. It generates all paths of the execution model that satisfy the input specification.

3. *Execution Model Inspector* is used by the symbolic executor each time the symbolic states are updated and has the role of updating the data memorized by the symbolic execution tree: symbolic values associated to the program variable and the path condition.

SAVE was used for industrial applications; it managed to prove some important properties of *TCAS (Traffic Alert and Collision Avoidance System)* – an on–board aircraft conflict detection and resolution system used by all US commercial aircraft.

The drawback of the *SAVE* system (as mentioned in [KPV03]) is that it uses dedicated tool (e.g. *PDL*) in order to perform the program analysis and also does not handle analysis of concurrent programs.

The authors of [KPV03] exploit this drawback in their system, defining a translator which allows symbolic execution to be performed using a model checker without the aid of a dedicated tool. The translator's output can be symbolically executed by any model checker that supports nondeterministic choice. Program analysis is performed then by manipulating logical formulae on program data values (using a decision procedure).

The system operates as follows: the original program is translated such that it can be executed using any model checker that supports backtracking and nondeterministic choice to perform symbolic execution. Code instrumentation and was done on top on *Korat* ([BKM02]). Then, the model checker used, *Java PathFinder* ([VHBP00]), checks the translated program using its state (heap configuration, path condition on primitive fields and thread scheduling) exploration approaches. A path condition is checked for satisfiability each time it is updated using appropriates decision procedures from the *Omega* library ([Pug92]). If the path condition is unsatisfiable, the model checker backtracks. The translated program can be supplied with specification in a language known by the model checker or a predicate abstraction upon which is checked the correctness. Counterexamples that violate the specification are provided while the correctness is checked. Test data are generated for the program paths that are witnesses of the specification criteria. Recent progress of this work was made in [PV04] by dealing with automatically generation of loop invariants for programs dealing also with arrays and dynamically allocated structures.

PREfix ([BPS00]) is also based essentially on symbolic execution. It is a tool used for detecting dynamic memory bugs of C/C++ commercial applications.

2.1.3 Program Verification in the *Theorema* System

Theorema ([BCJ+06]) is a computer aided mathematical software which is being developed at the Research Institute for Symbolic Computation (RISC) in Hagenberg, Austria.

The system offers support for computing, proving and solving mathematical expressions using specified knowledge bases, by applying several simplifiers, solvers and provers in natural style, which imitate the heuristics used by human provers

Composing, structuring and manipulating mathematical texts is also possible in the Theorema system, using labeling (`Definition, Theorem, Proposition`).

For our approach (imperative program verification), it is very important that the *Theorema* system provides a very expressive way to express *algorithms*: they are written in the language of predicate logic with equality as rewrite rules.

Theorema provides elegant proofs (because of natural style inferences used) in the verification process of programs. Moreover, being built on top of the computer algebra system *Mathematica* ([Wol03]), it has access to many computing and solving algorithms.

Currently, *Theorema* system has support for imperative and functional program verification.

Imperative program verification using *Hoare logic* and *weakest precondition strategy* was started in [Kir99] and continued with [Kov07].

The verification environment built in [Kir99] aims at educational purposes, namely for formal program specification, annotations, correctness proofs. It provides a *Theorema* language for writing imperative programs together with their specification, a tool for executing them and also a tool for the generation of the verification conditions (`VCG`). The verification conditions are proved/disproved to be valid using the various *Theorema* simplifiers and provers.

Insight to the problem of invariant generation is given by [Kov07], where logical, combinatorial and algebraic techniques work together for the automatic generation (using the `Aligator` package) of the invariants for loops containing assignments, sequences and conditionals.

Functional program verification environment in the *Theorema* system ([PJ03]) considers a main function (with its specification) and a tail recursive auxiliary function (without specification). A method which generates automatically the specification of the auxiliary function using algebraic techniques is developed. The specification is then used for generating the verification conditions for both the auxiliary and the main functions using fixpoint theory technique.

2.2 Forward Symbolic Execution in the *Theorema* System

We integrated in the *Theorema* system a new verification environment based on symbolic execution ([Cow88, Kin76, HK76]), forward reasoning ([Dro90, LSS84]) and functional semantics [McC63] for imperative program verification.

2.2.1 Basic Notions

A *state* σ is a set of replacements of the form $\{var \rightarrow expr\}$ with the meaning that every program variable is initialized. We call the initialized variables – active variables. We also write sometimes $\{\overline{var} \rightarrow \overline{expr}\}$ instead of $\{var_1 \rightarrow expr_1, var_2 \rightarrow expr_2, ...\}$.

Statements are used for building up *programs*. We use the following programming language constructs: the abrupt statement `Return`, assignments (including *recursive calls*) and conditionals (`If` with one and two branches). Recursive calls and conditionals insure the universality of the programming language.

The *specification* of a program P is the tuple $\langle I_P, O_P \rangle$, where I_P is a first-order logic *input condition predicate* and O_P is a first-order logic *output condition predicate*. The program correctness is proved relatively to its specification.

Forward reasoning was chosen in conjunction with symbolic execution for program analysis because it follows naturally the execution of programs. The principle is as follows: we start from the precondition of the program and apply a set of inference rules, depending on the type of the program statement currently analyzed, and obtain a conjunction of formulae. The postcondition of the program must be a logical consequence of the modified input condition.

In the program verification approaches using *functional semantics*, the flowchart of a program is transformed into a function from its input state to its output state. We use this approach for computing the program function which helps in defining the program semantics.

The approach that we integrated in the *Theorema* system is purely logical. We assume that the properties of the constants, functions and predicates which are used in the program are specified in an *object theory* Υ. (By a *theory* we understand a set of formulae in the language of predicate logic with equality.) For the purpose of reasoning about imperative programs we construct a certain *meta–theory* containing the properties of the meta–predicate Π (which checks a program for syntactical correctness) and the meta–functions Σ (which defines the semantics of a program), Γ (which generates the verification conditions) and Θ (which generates the termination condition). The programming language constructs (statements), the program itself, as well as the terms and the formulae from the object theory are *meta–terms* from the point of view of the meta-theory, and they behave like *quoted* (because the meta–theory does not contain any equalities between programming constructs, and also does not include the object theory).

2.2.2 A Meta-Logic for Reasoning about Imperative Programs

2.2.2.1 Syntax

We define the meta–level predicate Π for checking the appropriate syntax for the programs and, additionally, if every variable used in the program is active and if every branch of the program has a `Return` statement.

The formulae composing the meta–definitions below are to be understood as universally quantified over the meta–variables of various types as described in the sequel: We denote by $t \in \mathcal{T}$ a term from the set of object level terms, by $v \in \mathcal{V}$ a variable from the set of variables, by $V \subset \mathcal{V}$ the set of active variables and by φ an object level formula. P_T, P_F, P are tuples of statements representing parts of programs: P_T is the tuple of statements executed if φ is evaluated to $True$, P_F in the case of $False$ evaluation of φ, while P represents the rest of the program to be executed. The tuple P_F can be also empty, case which corresponds to the *one branch* `If` *statement*.

The meta–function *Vars* returns the set of variables which occur in a term.

Definition 2.2.1.

1. $\Pi[P] \Longleftrightarrow \Pi[\{\bar{x}\}, P]$

2. $\Pi[V, \langle \texttt{Return}[t] \rangle \smile P] \Longleftrightarrow Vars[t] \subseteq V$

3. $\Pi[V, \langle v: \ = t \rangle \smile P] \Longleftrightarrow \bigwedge \left\{ \begin{array}{l} Vars[t] \subseteq V \\ \Pi[V \cup \{v\}, P] \end{array} \right.$

4. $\Pi[V, \langle \texttt{If}[\varphi, P_T, P_F] \rangle \smile P] \Longleftrightarrow \bigwedge \left\{ \begin{array}{l} Vars[\varphi] \subseteq V \\ \Pi[V, P_T \smile P] \\ \Pi[V, P_F \smile P] \end{array} \right.$

5. $\Pi[V, P] \Longleftrightarrow \mathbb{F}$, in all other cases

The 2-arity predicate symbol Π analyzes the program P statement by statement, updating eventually the set of active variables. The first argument is used, in conjunction with the function *Vars*, for inactive variables detection.

The initial active variables are the input variables of the program (Definition 2.2.1.1).

A term a term t can be *return*ed (Definition 2.2.1.2) only if the variables occurring in the term construction are active.

An *assignment* (Definition 2.2.1.3) is performed with active variables only and the result consists of a new active variable which can be used further in the computations.

The *conditional* (Definition 2.2.1.4) requires that the conditional expression φ must contain active variables and the P_T, P_F and P branches have to fulfill the requirements of the predicate Π. If P_F is the empty tuple, we have the case of `If` with one branch.

These definitions are conditionals equalities. If no definitions 2.2.1.1–2.2.1.4 applies then the Definition 2.2.1.5 is applied.

2.2.2.2 Semantics

The meta–level function Σ creates an object–level level formula with the shape:

$$F[P] : \underset{x:I_P}{\forall} \bigwedge \{p_i[x] \Rightarrow (f[x] = g_i[x])\}_{i=1}^{n}. \tag{2.1}$$

(We denoted by „$\bar{x} : I_f$” in the condition “$\bar{x}$ satisfies I_f”.)

Here f is a new (second order) symbol – a name for the function defined by the program. In the case of recursive programs, f may occur in some p_i's and g_i's.

Each of the n paths of the program has associated a object–level formula $p_i[x]$ – the accumulated `If`–conditions on that path, and the object–level term $g_i[x]$ – the symbolic expression of the return value obtained by composing all the assignments (symbolic execution). Note that $p_i[x]$ and $g_i[x]$ do not contain other free variables than x.

The computing idea for the program semantics Σ is as follows: Σ works by forward symbolic execution on all branches of the program, using as *state* the current substitution for the active variables. Σ produces a conjunction of clauses – conditional definitions for $f[\overline{x}]$. Each clause depends on the accumulated [negated] conditions of the `If` statements leading to a certain `Return` statement, whose argument (symbolically evaluated) represents the corresponding value of $f[\overline{x}]$.

Definition 2.2.2.

1. $\Sigma[P] = \underset{\bar{x}}{\forall}\left(\Sigma[\{\bar{x} \rightarrow \bar{x_0}\}, P]_{\{\bar{x_0} \rightarrow \bar{x}\}}\right)$

2. $\Sigma[\sigma, \langle \mathtt{Return}[t] \rangle \smile P] = (f[\bar{x_0}] = t\sigma)$

3. $\Sigma[\sigma, \langle v := t \rangle \smile P] = \Sigma[\sigma \circ \{v \rightarrow t\sigma\}, P]$

4. $\Sigma[\sigma, \langle \mathtt{If}[\varphi, P_T, P_F] \rangle \smile P] = \bigwedge \begin{cases} \varphi\sigma \Longrightarrow \Sigma[\sigma, P_T \smile P] \\ \neg\varphi\sigma \Longrightarrow \Sigma[\sigma, P_F \smile P] \end{cases}$

When the execution of the program starts, all the input variables become active by instantiating them with corresponding symbolic values. After the program is processed all the input variables become universally quantified (Definition 2.2.2.1).

A `Return` statement (Definition 2.2.2.2) determines the computation of the output state. From this state we are interested in the program function and its computed values (the argument t of the `Return` statement).

The *assignment statement* (Definition 2.2.2.3) updates the current state.

For a *conditional* (Definition 2.2.2.4), there are two different expressions for the program function to be computed: one when the symbolic formula $\varphi\sigma$ might hold and one in the opposite case.

Remarks.

1. The way Σ handles the `If` statement ensures: $\bigvee_{i=\overline{1,n}} p_i = I_f$ (all branches are covered) and $\underset{i \neq j}{\forall} p_i \wedge p_j = \mathbb{F}$ (branches are mutually disjoint).

2. The program function Σ effectively translates an imperative program into a *functional* program. From this point on, one could reason about the program using the Scott fixpoint theory ([LSS84], pag. 86), however we prefer a purely logical approach.

2.2.2.3 Partial Correctness

We define inductively the meta–level function Γ, generating the verification conditions (at object-level) insuring the partial correctness of the programs.

The function Γ has three arguments:

- σ is the current state of the program;
- Φ is an object-level formula representing the accumulated conditions on the path currently analyzed;
- P is the program (or the rest of it) which is analyzed.

We consider: (i)γ, $\overline{\gamma}$ – a variable or a constant and respectively a sequence of variable and/or constants from the theory Υ, (ii) basic functions h (i.e. functions from the object theory), (iii) additional functions g (i.e. functions computed by other programs), (iv) arbitrary functions u. The symbol y is a new constant name standing for the program function. An expression like $e_{\tau\leftarrow w}$ denotes that τ is replaced by w in e, where w is a new variable name.

The *coherence (safety)* conditions are generated for each call of a function h and state that the function h is called upon arguments which satisfy its input specification (h may be also P, case which corresponds to the recursive call) (see definitions 2.2.3.4 and 2.2.3.5).

The *functional* conditions are generated at the end of each branch, insuring that the return value satisfies the output specification O_P (see definitions 2.2.3.2 and 2.2.3.3).

Definition 2.2.3.

1. $\Gamma[P] = \underset{\bar{x}}{\forall}\big(\Gamma[\{\bar{x} \to \bar{x}_0\},\ I_f[\bar{x}_0],\ P]_{\{\bar{x}_0 \to \bar{x}\}}\big)$

2. $\Gamma[\sigma,\ \Phi,\ \langle \texttt{Return}[\gamma]\rangle \smile P] = \Big(\Phi \Rightarrow O_f[\bar{x}_0, \gamma\sigma]\Big)$

3. $\Gamma[\sigma,\ \Phi,\ \langle \texttt{Return}[t_{\tau\leftarrow u[\overline{\gamma}]}]\rangle \smile P] = \Gamma[\sigma,\ \Phi,\ \langle w := u[\overline{\gamma}],\ \texttt{Return}[t_{\tau\leftarrow w}]\rangle \smile P]$

4. $\Gamma[\sigma,\ \Phi,\ \langle v := \gamma\rangle \smile P] = \Gamma[\sigma \circ \{v \to \gamma\sigma\},\ \Phi,\ P]$

5. $\Gamma[\sigma,\ \Phi,\ \langle v := h[\overline{\gamma}]\rangle \smile P] =$
$$\bigwedge \begin{cases} \Phi \Rightarrow I_h[\overline{\gamma}\sigma] \\ \Gamma[\sigma \circ \{v \to h[\overline{\gamma}\sigma]\},\ \Phi \wedge I_h[\overline{\gamma}\sigma],\ P] \end{cases}$$

6. $\Gamma[\sigma,\ \Phi,\ \langle v := g[\overline{\gamma}]\rangle \smile P] =$
$$\bigwedge \begin{cases} \Phi \Rightarrow I_g[\overline{\gamma}\sigma] \\ \Gamma[\sigma \circ \{v \to c\},\ \Phi \wedge I_g[\overline{\gamma}\sigma] \wedge O_g[\overline{\gamma}\sigma, c],\ P] \end{cases}$$

7. $\Gamma[\sigma,\ \Phi,\ \langle v := t_{\tau\leftarrow u[\overline{\gamma}]}\rangle \smile P] = \Gamma[\sigma,\ \Phi,\ \langle w := u[\overline{\gamma}],\ v := t_{\tau\leftarrow w}\rangle \smile P]$

8. $\Gamma[\sigma,\ \Phi,\ \langle \mathtt{If}[\varphi_{\tau\leftarrow u[\overline{\gamma}]},\ P_T,\ P_F]\rangle \smile P] =$
$$\Gamma[\sigma,\ \Phi,\ \langle w := u[\overline{\gamma}],\ \mathtt{If}[\varphi_{\tau\leftarrow w},\ P_T,\ P_F]\rangle \smile P]$$

9. $$\Gamma[\sigma,\ \Phi,\ \langle \mathtt{If}[\varphi,\ P_T,\ P_F]\rangle \smile P] = \bigwedge \left\{ \begin{array}{l} \Gamma[\sigma,\ \Phi \wedge \varphi\sigma,\ P_T \smile P] \\ \Gamma[\sigma,\ \Phi \wedge \neg\varphi\sigma,\ P_F \smile P] \end{array} \right.$$

The verification conditions corresponding to the program P are generated as follows: we start analyzing the program P and, in the first step (Definition 2.2.3.1), we create the substitution $\{\bar{x} \rightarrow \bar{x_0}\}$ (all the input variables are initialized in the initial state) and the path condition $I_P[\bar{x_0}]$ (the input condition has to be fulfilled before the program analysis).

The meta-level function Γ analyzes each statement of the program as follows:

- a composed term (definitions 2.2.3.3, 2.2.3.7 and 2.2.3.8) is first decomposed such that no nested functions are present. After the decomposition, we analyze its parts separately.

- a `Return` statement (Definition 2.2.3.2) finishes the execution of the program on the current path and a functional verification condition is generated. The formula representing the postcondition depends on the input variable and the term returned by the `Return` statement. The statements after the `Return` statement are ignored;

- a *constant* or *variable assignment* (Definition 2.2.3.4) updates the substitution σ, while a *function* (definitions 2.2.3.5 and 2.2.3.6) and term assignments (Definition 2.2.3.7), have to be treated in the following way: safety verification conditions have to be generated for every function (including f): the arguments of them must satisfy the respective input condition and afterwards the analysis of the program continues with the input condition of the function as new assumption ;

- a *conditional analysis* determines the forking of the program into two branches: one when φ evaluates to $True$ and one in the opposite case. The branch P_F of the program might be also the empty tuple, case which corresponds to the `If` *statement with one branch.*

The order of the above clauses of Γ has a semantic meaning. Namely, we use this as an abbreviation for additional conditions which should be added to the clauses of the definition in order to specify that, for instance, the equality from the Definition 2.2.3.9 is applied only if no subterm of φ is of the form $u[\overline{\gamma}]$ – as specified in the clause from the Definition 2.2.3.7.

2.2.2.4 Termination

We want to generate verification conditions which insure that a program is correct with respect to a specification composed of two object–level formulae: the input condition $I_f[x]$ and the output condition $O_f[x, y]$. Apparently, the correctness could be expressed as: "The formula $\underset{x}{\forall} I_f[x] \Rightarrow O_f[x, P[x]]$ is a logical consequence of the theory Υ augmented with $\Sigma[P]$ and with the verification conditions." However, this always holds in the case that $\Sigma[P]$ is contradictory to Υ, which may happen when the program is recursive. Therefore, *it is crucial that the existence (and possibly the uniqueness) of a f satisfying $\Sigma[P]$ is a logical consequence of the object theory augmented with the verification conditions.* More concretely, before using $\Sigma[P]$ as an assumption,

one should prove $\underset{f}{\exists}\Sigma[P]$. The later is ensured by the termination condition which is expressed as an induction scheme developed from the structure of the recursion.

The meta–function Θ generates the termination condition (for simplicity of presentation we assume that `Return`, assignments, and `If` conditions do not contain composite terms – the elimination of these by introducing new assignments can be done as in the definition of Γ).

Definition 2.2.4.

1. $\Theta[P] = \big(\underset{\bar{x}:I_f}{\forall} \Theta[\{\bar{x} \rightarrow \bar{x_0}\},\, \mathbb{T},\, P]_{\{\bar{x_0} \leftarrow \bar{x}\}}\big) \implies \underset{\bar{x}:I_f}{\forall} \pi[\bar{x}]$

2. $\Theta[\sigma,\, \Phi,\, \langle \mathtt{Return}[\gamma]\rangle \smile P] = \big(\Phi \Rightarrow \pi[\bar{x_0}]\big)$

3. $\Theta[\sigma,\, \Phi,\, \langle v := \gamma\rangle \smile P] = \Theta[\sigma \circ \{v \rightarrow \gamma\sigma\},\, \Phi,\, P]$

4. $\Theta[\sigma,\, \Phi,\, \langle v := h[\overline{\gamma}]\rangle \smile P] = \Theta[\sigma \circ \{v \rightarrow h[\overline{\gamma}\sigma]\},\, \Phi,\, P]$

5. $\Theta[\sigma,\, \Phi,\, \langle v := f[\overline{\gamma}]\rangle \smile P] = \Theta[\sigma \circ \{v \rightarrow y\},\, \Phi \wedge O_f[\overline{\gamma}\sigma, y] \wedge \pi[\overline{\gamma}\sigma],\, P]$

6. $\Theta[\sigma,\, \Phi,\, \langle v := g[\overline{\gamma}]\rangle \smile P] = \Theta[\sigma \circ \{v \rightarrow y\},\, \Phi \wedge O_g[\overline{\gamma}\sigma, y],\, P]$

7. $\Theta[\sigma,\, \Phi,\, \langle \mathtt{If}[\varphi,\, P_T,\, P_F]\rangle \smile P] = \bigwedge \begin{cases} \Theta[\sigma,\, \Phi \wedge \varphi\sigma,\, P_T \smile P] \\ \Theta[\sigma,\, \Phi \wedge \neg\varphi\sigma,\, P_F \smile P] \end{cases}$

One single formula is generated, which uses a new constant symbol π standing for an arbitrary predicate. The function Θ operates similarly to Γ by inspecting all possible branches and collecting the respective `If` conditions (in Φ). Moreover it collects the characterizations by output conditions of the values produced by calls to additional functions (Definition 2.2.4.6), including for the currently defined function f. However, in the case of f (Definition 2.2.4.5) one also collects the condition $\pi[\overline{\gamma}\sigma]$ – that is the arbitrary predicate applied to the current symbolic values of the arguments of the recursive call to f. On each branch, the collected conditions are used as premise of $\pi[\bar{x_0}]$, and then the conjunction of all these clauses (after reverting to free variables $\bar{x}$) is universally quantified over the input condition and is used as a premise in the final formula.

2.3 The Simplification of the Verification Conditions

The output of the verification conditions generator is logical formulae which are harder to see that they are valid/invalid without a [automatic] proof. We are interested mainly in numerical programs and, as a consequence, the verification conditions generated contain also equalities and inequalities between the program variables. Our goal is to simplify these conditions until system(s) of equalities and inequalities point when one can apply computer algebra and combinatorics techniques in order to reason about the program correctness.

In the first step of the verification conditions simplification process we applied logical inferences, namely truth constants elimination and natural style inference rules.

The truth constants appear, usually, when safety verification conditions are generated in order to insure the input condition of the basic functions. A list with the possible occurrences of the truth constants, either on the left-hand-side, either on the right-hand-side of the logical connectives, is applied recursively to the initial set of verification conditions, until the formulae do not change anymore.

Example 2.1.

```
...
(*truth constants elimination on the rhs*)
•[lhs_ ⇒ True]:→ •[True],
•[lhs_⇒ False]:→ •[Not[lhs]],
•[lhs_⇔ True]:→ •[lhs],
•[lhs_⇔ False]:→ •[Not[lhs]]
...
```

In the next step we normalize the formulae by transforming the equalities and inequalities such that they have 0 on the right-hand-side and $\geq$, $>$, $=$, $\neq$ occur only. Afterwards, the inequalities and equalities from the assumptions or from the goal are then rearranged such that the order is: $\neq$, $=$, $\geq$ and $>$.

Some simplifications involving the relationships between $\neq$, $=$, $\geq$, $>$ are done in the next steps:

Example 2.2.

```
•[And[pre___,g_≠ 0,body___,g_=0,post___]]:→ •[False],
•[And[pre___,g_≠ 0,body___,g_≥ 0,post___]]:→•[And[pre,g>0,body,post]],
•[And[pre___,g_≠ 0,body___,g_>0,post___]]:→• [And[pre,g>0,body,post]],
...
```

For the inference rules:

Quantifiers elimination

From the assumptions	From the goal
$\frac{\Phi,\phi_{x\leftarrow a}\vdash\Psi}{\Phi,\underset{x}{\exists}\phi\vdash\Psi}\ (\exists\vdash)$ a is new	$\frac{\Phi\vdash\Psi,\psi_{x\leftarrow t}}{\Phi\vdash\Psi,\underset{x}{\exists}\psi}\ (\vdash\exists)$ t has to be found
$\frac{\Phi,\phi_{x\leftarrow t}\vdash\Psi}{\Phi,\underset{x}{\forall}\phi\vdash\Psi}\ (\forall\vdash)$ t has to be found	$\frac{\Phi\vdash\Psi,\psi_{x\leftarrow a}}{\Phi\vdash\Psi,\underset{x}{\forall}\psi}\ (\vdash\forall)$ a is new

we introduce new skolem constants and meta-variables, namely:

- skolem constants for universally quantified formulae in the goal and existentially quantified formulae in the assumptions;
- meta-variables for universally quantified formulae in the assumptions and existentially quantified formulae in the goal.

Meta-variables correspond to the problem of finding the witness terms in natural deduction. In the implementation of the meta-variables method, one constructs first a partial proof containing the meta-variables and then appropriate substitutions (concrete terms) for the meta-variables transform the partial proof into a concrete proof.

Our simplifier generates partial proof, but not a complete proof.

Example 2.3.

For example in the program which computes the factorial of a natural number with the precondition $n \geq 0$ *and the postcondition* $\underset{m \geq 0 \wedge m < n}{\overset{}{\underset{m}{\forall}}} \underset{k \geq 0}{\underset{k}{\exists}} (y = k * m)$*, one of the verification conditions is:*

$$n \geq 0 \wedge n \neq 0 \wedge True \wedge n-1 \geq 0 \wedge \underset{m \geq 0 \wedge m < n-1}{\underset{m}{\forall}} \underset{k \geq 0}{\underset{k}{\exists}} (t2 = k * m) \wedge True \Longrightarrow \underset{m \geq 0 \wedge m < n}{\underset{m}{\forall}} \underset{k \geq 0}{\underset{k}{\exists}} (n * t2 = k * m)$$

After the simplification of the truth constants and the introduction of the skolem and meta-variables, the formula becomes:

$$\begin{array}{l}(n > 0 \wedge n-1 \geq 0 \wedge ((m_0^* \geq 0 \wedge (-m_0^*) + (n-1) > 0 \Rightarrow (t2 = k_2^* * m_0^*)) \wedge \\ \quad (m_0^* \geq 0 \wedge (-m_0^*) + (n-1) > 0 \Rightarrow k_2^* \geq 0)) \Rightarrow (m_1 \geq 0 \wedge (-m_1) + n > 0 \Longrightarrow k_1^* \geq 0)) \bigwedge \\ \quad (n > 0 \wedge n-1 \geq 0 \wedge ((m_0^* \geq 0 \wedge (-m_0^*) + (n-1) > 0 \Rightarrow (t2 = k_2^* * m_0^*)) \wedge \\ \quad (m_0^* \geq 0 \wedge (-m_0^*) + (n-1) > 0 \Rightarrow k_2^* \geq 0)) \Rightarrow (m_1 \geq 0 \wedge (-m_1) + n > 0 \Longrightarrow (n * t2 = k_1^* * m_1)))\end{array}$$

In the previous example we also use the inference rule:

$$\frac{\dfrac{\Phi \Rightarrow \Psi, C[x^*] \qquad\qquad \Phi \Rightarrow \Psi, \psi}{\Phi \Rightarrow \Psi, C[x^*] \wedge \psi}}{\Phi \Rightarrow \Psi, \underset{C[x]}{\underset{x}{\exists}} \psi}$$

Some of the verification conditions generated do not contain universal and existential logical connectors. Therefore some of the formulae which do not involve skolem constants and meta-variables can be simplified using the Mathematica built in `FullSimplify` construct. Before the application of the `FullSimplify`, we transformed the formulae from the *Theorema* into the *Mathematica* syntax.

Example 2.4.

For the algorithm which computes x^n *using few operations (algorithm known as* ***Binary Powering*** *because is trying to write* n *using binary notation), we were able to prove that the algorithm is correct using our version of simplifier.*

We present in the following the sequence of transformations after the logical and algebraic simplifications:

$$n \geq 0 \Rightarrow (1 = x^0)$$
$$n > 0 \Rightarrow (x = x^1)$$
$$True, True$$
$$n > 0 \wedge (Mod[n, 2] = 0) \wedge n - 1 > 0 \wedge (-n) + 1 > 0 \Rightarrow n \geq 0$$
$$n > 0 \wedge (Mod[n, 2] = 0) \wedge n - 1 > 0 \wedge (-n) + 1 > 0 \Rightarrow ((x^2)^{\frac{n}{2}} = x^n)$$
$$True, True, True$$
$$n > 0 \wedge Mod[n, 2] \neq 0 \wedge n - 1 > 0 \wedge (-n) + 1 > 0 \Rightarrow n - 1 \geq 0$$
$$True$$
$$n > 0 \wedge Mod[n, 2] \neq 0 \wedge n - 1 > 0 \wedge (-n) + 1 > 0 \Rightarrow (x * (x^2)^{\frac{n-1}{2}} = x^n)$$

In this version, the verification conditions are written in the Theorema version (quoted formulae). We transformed them into the Mathematica syntax and apply the built-in `FullSimplify`*, we reduced them to the truth constant $True$, one for each formula. With a special list simplification rule, we reduced them to one truth constant $True$.*

In the next sections, we exemplify the features of the verification conditions generator and formulae simplifier on three small programs. We have chosen them in order to distinguish the features of the system.

2.4 Implementation and Examples

The prototype environment (called `FwdVCG`) which follows the theoretical basis presented in **Section 2.2** is built on top of the computer algebra system *Mathematica* and uses the existing *Theorema* environment for imperative program verification which supposes that:

- the programs are considered procedures with input parameters and mandatory return values (output parameters);
- expressions are *Theorema* boolean and arithmetic expressions;
- the specification and the program are identified by specific commands: `Pre, Post` and `Program` respectively.

The verification conditions generator takes as input a program together with its specification and generates the corresponding proof obligations which arise from its analysis.

For specifying a program in the *Theorema* system we use the command `Program` and for writing the specification, the keywords `Pre, Post`. All these constructs and also the syntax of the imperative programs is based on [Kir99].

The input variables are denoted by a „↓" in front of them. The input specification is expressed in terms of the input variables and the output specification in terms of the input variables and the function computed by the program via the `Return` (output variable).

In the next examples we show how the verification conditions and the termination condition is generated for basic, additional and recursive functions respectively.

2.4.1 Greatest Common Divisor using the Euclidean Algorithm

The next example computes the greatest common divisor of two natural numbers.

```
1. Program["GCD", GCD[↓ a, ↓ b]],
2.  Module[{}],
3.   If[a = 0,
4.    Return[b]];
5.   If[b != 0,
6.    If[a > b,
7.     a := GCD[a - b, b],
8.     a := GCD[a, b - a]]];
9.   Return[a]],
10.  Pre → a ≥ 0 ∧ b ≥ 0,
11.  Post→ ∃∃_{k1k2}((a = k1 ∗ y) ∧(b = k2 ∗ y))
```

The automatically generated verification conditions and termination condition for the previous example are generated with the command:

$$\texttt{FwdVCG}[\texttt{Program}[''GCD'']]$$

and are the following universally quantified first order logic formulae:

$$(a \geq 0 \wedge b \geq 0) \wedge (a = 0) \Rightarrow \underset{k1k2}{\exists\exists}((a = k1 * b) \wedge (b = k2 * b)) \tag{2.2}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \Rightarrow a \geq b \tag{2.3}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \wedge a \geq b \Rightarrow a - b \geq 0 \wedge b \geq 0 \tag{2.4}$$

$$\begin{aligned}&(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \wedge a \geq b \wedge (a - b \geq 0 \wedge b \geq 0) \wedge \\ &\quad \underset{k1k2}{\exists\exists}((a - b = k1 * \mathrm{y1}) \wedge (b = k2 * \mathrm{y1})) \Rightarrow \underset{k1k2}{\exists\exists}((a = k1 * \mathrm{y1}) \wedge (b = k2 * \mathrm{y1}))\end{aligned} \tag{2.5}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a \not> b \Rightarrow a \geq b \tag{2.6}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a \not> b \wedge a \geq b \Rightarrow a \geq 0 \wedge b - a \geq 0 \tag{2.7}$$

$$\begin{aligned}&(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a \not> b \wedge a \geq b \wedge (a \geq 0 \wedge b - a \geq 0) \wedge \\ &\quad \underset{k1k2}{\exists\exists}((a = k1 * \mathrm{y2}) \wedge (b - a = k2 * \mathrm{y2})) \Rightarrow \underset{k1k2}{\exists\exists}((a = k1 * \mathrm{y2}) \wedge (b = k2 * \mathrm{y2}))\end{aligned} \tag{2.8}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge \neg(b \neq 0) \Rightarrow \underset{k1k2}{\exists\exists}((a = k1 * a) \wedge (b = k2 * a)) \tag{2.9}$$

The termination condition is:

$$\Big(\underset{\substack{a,b \\ a\geq 0, b\geq 0}}{\forall} \bigwedge \begin{cases} a = 0 \Rightarrow \pi[a,b] \\ (a \neq 0 \wedge b \neq 0 \wedge a > b \wedge \pi[a-b,b]) \Rightarrow \pi[a,b] \\ (a \neq 0 \wedge b \neq 0 \wedge a \not> b \wedge \pi[a,b-a]) \Rightarrow \pi[a,b] \\ (a \neq 0 \wedge b = 0) \Rightarrow \pi[a,b] \end{cases} \Big) \Longrightarrow \Big(\underset{\substack{a,b \\ a\geq 0, b\geq 0}}{\forall} \pi[a,b] \Big)$$

Description. The previously verification conditions were generated by analyzing each path of the program and applying the notions of program syntax, semantics, partial correctness and termination introduced previously.

Before the program starts to be analyzed, the substitution $\{a \to a_0, b \to b_0\}$ and the formula representing the accumulated assumptions $a_0 \geq 0 \wedge b_0 \geq 0$ are created. After the program is completely analyzed, the input variables are universally quantified thus the reverting substitution is done.

On the path 10, 1, 2, 3, 4, 11, the verification conditions were generated as follows: we add to the formula $a_0 \geq 0 \wedge b_0 \geq 0$ the conjunct $a = 0$, corresponding to the $True$ evaluation of the `If` conditional. The `Return` statement determines the generation of the functional verification condition 2.2, where the value of y from the postcondition was replaced by the value returned by the program on this branch, namely b.

On the path 10, 1, 2, 5, 6, 7, 9, 11, the assignment from the line 7 requires a term decomposition from the innermost to the outermost function symbol. The assumptions collected before the term analysis is the formula $a \geq 0 \wedge b \geq 0 \wedge a \neq 0 \wedge b \neq 0$. The specification of the functions composing the term `GCD[a-b, b]` has to be fulfilled. First the function „-" is analyzed, generating the safety condition 2.3, insuring the respective precondition. The assumptions on this path are updated with the specification of the function „-", namely $a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \wedge a \geq b$.

The analysis of the function GCD is done similarly: the precondition has to hold for the arguments of the recursive call, the output value as well. The functional verification condition (2.5) is generated, where all the occurrences of the function GCD are replaced by the new constant $y1$.

The path 10, 1, 2, 5, 8, 9, 11 is analyzed similar to the path 10, 1, 2, 5, 6, 7, 9, 11 and the path 10, 1, 2, 9, 11, similar to the path 10, 1, 2, 3, 4, 11.

The termination condition is obtained similarly to the verification conditions. It is an implication of two first order logical formulae (we consider that the predicate π is fixed), both universally quantified upon the input variables. The right hand side is always the predicate π with the input variables as arguments and the left hand side is a conjunction of formulae, each conjunct (having the shape of an implication) corresponding to the analysis of a single path. If on the path there exists a recursive call then the new introduced predicate symbol π occurs in the both sides of the implication within the conjunct (the second and the third conjunct corresponding to the second and the third path of the program). The π occurring on the left hand side has the arguments of the recursive call on the respective branch. If there is not a recursive call then the π occurs only on the right hand side (the first and the forth conjunct).

The simplified list of verification conditions is:

$((a = 0) \wedge b \geq 0 \wedge a \geq 0 \Longrightarrow (a = k1_0^* * b)) \bigwedge ((a = 0) \wedge b \geq 0 \wedge a \geq 0 \Longrightarrow (b = k2_0^* * b))$

$True$

$(a > 0 \wedge b \neq 0 \wedge b \geq 0 \wedge a - b > 0 \Longrightarrow b \geq 0) \bigwedge (a > 0 \wedge b \neq 0 \wedge b \geq 0 \wedge a - b > 0 \Longrightarrow$
$a - b \geq 0)$

$(a > 0 \wedge b \neq 0 \wedge b \geq 0 \wedge (b = k2_0 * t2) \wedge a - b > 0 \wedge (a - b = k1_0 * t2) \Longrightarrow (a = k1_1^* * t2)) \bigwedge$
$(a > 0 \wedge b \neq 0 \wedge b \geq 0 \wedge (b = k2_0 * t2) \wedge a - b > 0 \wedge (a - b = k1_0 * t2) \Longrightarrow (b = k2_1^* * t2))$

$True$

$(a > 0 \wedge b \neq 0 \wedge b \geq 0 \wedge -(a - b) \geq 0 \Longrightarrow a \geq 0) \bigwedge (a > 0 \wedge b \neq 0 \wedge b \geq 0 \wedge -(a - b) \geq 0$
$\Longrightarrow b - a \geq 0)$

$(a > 0 \wedge b \neq 0 \wedge b \geq 0 \wedge (a = k1_1 * t4) \wedge b - a \geq 0 \wedge (b - a = k2_1 * t4) \wedge -(a - b) \geq 0$
$\Longrightarrow (a = k1_2^* * t4)) \bigwedge (a > 0 \wedge b \neq 0 \wedge b \geq 0 \wedge (a = k1_1 * t4) \wedge b - a \geq 0 \wedge (b - a = k2_1 * t4) \wedge$
$-(a - b) \geq 0 \Longrightarrow (b = k2_2^* * t4))$

$(a > 0 \wedge b \geq 0 \Longrightarrow (0 = k2_3^* * a)) \bigwedge (a > 0 \wedge b \geq 0 \Longrightarrow (a = k1_3^* * a))$

2.4.2 Solving First Degree Equations

```
1. Program["FirstDegreeSolver", FDS[↓ a, ↓ b]],
2.  Module[{}],
3.   If[a = 0,
4.    If[b = 0,
5.     Return[ℝ];
6.     Return[∅];
7.   Return[{-(b/a)}]];
8.  Pre → True,
9.  Post→ ∀x((x ∈ y) ⟺ (a * x + b = 0))
```

The verification conditions generated are:

$$True \wedge (a = 0) \wedge (b = 0) \Rightarrow \underset{x}{\forall}\Big(x \in \mathbb{R} \Leftrightarrow (a * x + b = 0)\Big)$$

$$True \wedge (a = 0) \wedge (b \neq 0) \Rightarrow \underset{x}{\forall}\Big(x \in \{\} \Leftrightarrow (a * x + b = 0)\Big)$$

$$True \wedge a \neq 0 \Rightarrow \underset{x}{\forall}\Big(x \in \left\{-\frac{b}{a}\right\} \Leftrightarrow (a * x + b = 0)\Big)$$

The termination condition generated is:

$$\Big(\underset{a,b}{\forall} \bigwedge \left\{ \begin{array}{l} True \wedge (a = 0) \wedge (b = 0) \Rightarrow \pi[a, b] \\ True \wedge (a = 0) \wedge (b \neq 0) \Rightarrow \pi[a, b] \\ True \wedge (a \neq 0) \Rightarrow \pi[a, b] \end{array} \right. \Big) \Longrightarrow \underset{a,b}{\forall} \pi[a, b]$$

The simplified list of verification conditions is:

$True, True,$

$$a \neq 0 \Longrightarrow (x_{12} = (-\frac{b}{a}) \Rightarrow (a * x_{12} + b = 0)) \bigwedge (a \neq 0 \Longrightarrow ((a * x_{12} + b = 0) \Rightarrow x_{12} = (-\frac{b}{a})))$$

2.4.3 Solving Second Degree Equations

```
1. Program["SecondDegreeSolver-calling-FirstDegreeSolver",SDS[↓ a,↓ b,↓ c],
2. Module[{delta ,sqrtDelta,x1,x2,sol},
3.  If[a==0,
4.   sol := FDS[b,c]; Return[sol],
4.   delta :=(b*b)-(4*a*c);
5.   If[delta =0,
6.    x1 := -(b/(2*a));x2:=x1,
7.   If[delta > 0,
8.    sqrtDelta:=Sqrt[delta]; x1 := (-b+sqrtDelta)/(2*a); x2 := (-b-sqrtDelta)/(2*a),
9.    sqrtDelta := Sqrt[-delta ]; x1 := (-b-(i*sqrtDelta))/(2*a); x2:= (-b+(i*sqrtDelta))/(2*a)]];
10.  sol:={x1, x2};Return[sol]]],
11. Pre ← True,
12. Post ← ∀_x((x ∈ y) ⟺ (a * x^2 + b * x + c = 0))
```

We display the verification conditions after simplification.

$True,$

$$\Big((x_1^* = FDS[b,c]) \Rightarrow (b * x_1^* + c = 0) \bigwedge (b * x_1^* + c = 0) \Rightarrow (x_1^* = FDS[b,c])\Big) \Longrightarrow \Big((x_4 = FDS[b,c]) \Rightarrow (b * x_4 + c = 0) \bigwedge (b * x_4 + c = 0) \Rightarrow (x_4 = FDS[b,c])\Big),$$

$True, True, True, True, True, True, True,$

$$a \neq 0 \wedge (b * b - 4 * a * c = 0) \Longrightarrow \Big((a * x_5^2 + b * x_5 + c = 0) \Rightarrow (x_5 = -\frac{b}{2 * a}) \vee ((a * x_5^2 + b * x_5 + c = 0) \Rightarrow (x_5 = -\frac{b}{2 * a})) \bigwedge (((x_5 = -\frac{b}{2 * a})) \Rightarrow (a * x_5^2 + b * x_5 + c = 0)) \vee ((x_5 = (-\frac{b}{2 * a})) \Rightarrow (a * x_5^2 + b * x_5 + c = 0))\Big),$$

$$a \neq 0 \bigwedge b * b - 4 * a * c > 0 \bigwedge (-b * b) + 4 * a * c > 0 \Longrightarrow b * b - 4 * a * c \geq 0,$$

$True, True, True, True, True, True, True, True, True,$

$$a \neq 0 \wedge b * b - 4 * a * c > 0 \wedge (-b * b) + 4 * a * c > 0 \Longrightarrow \Big((((a * x_6^2 + b * x_6 + c = 0) \Rightarrow (x_6 = \frac{(-b) + (\sqrt{b * b - 4 * a * c})}{2 * a})) \vee ((a * x_6^2 + b * x_6 + c = 0) \Rightarrow (x_6 = \frac{(-b) - (\sqrt{b * b - 4 * a * c})}{2 * a}))) \bigwedge (((x_6 = \frac{(-b) + (\sqrt{b * b - 4 * a * c})}{2 * a}) \Rightarrow (a * x_6^2 + b * x_6 + c = 0)) \vee ((x_6 = \frac{(-b) - (\sqrt{b * b - 4 * a * c})}{2 * a}) \Rightarrow (a * x_6^2 + b * x_6 + c = 0)))\Big),$$

$$True,$$
$$a \neq 0 \bigwedge b*b-4*a*c>0 \bigwedge (-b*b)+4*a*c>0 \bigwedge -(b*b-4*a*c) \geq 0 \Longrightarrow$$
$$-(b*b-4*a*c) \geq 0,$$
$$True, True, True, True, True, True, True, True, True, True, True,$$
$$a \neq 0 \wedge b*b-4*a*c>0 \wedge -(b*b-4*a*c) \geq 0 \wedge (-b*b)+4*a*c>0 \Longrightarrow$$
$$\Big((((a*x_7^2+b*x_7+c=0) \Rightarrow (x_7 = \frac{(-b)-\mathbf{i}*(\sqrt{-(b*b-4*a*c)})}{2*a}))\vee$$
$$((a*x_7^2+b*x_7+c=0) \Rightarrow (x_7 = \frac{(-b)+\mathbf{i}*(\sqrt{-(b*b-4*a*c)})}{2*a}))) \bigwedge$$
$$(((x_7 = \frac{(-b)-\mathbf{i}*(\sqrt{-(b*b-4*a*c)})}{2*a}) \Rightarrow (a*x_7^2+b*x_7+c=0))\vee$$
$$((x_7 = \frac{(-b)+\mathbf{i}*(\sqrt{-(b*b-4*a*c)})}{2*a}) \Rightarrow (a*x_7^2+b*x_7+c=0)))\Big)$$

The simplified termination condition is:

$$\forall_{a,b} \bigwedge \begin{cases} (x_1^* = FDS[b,c]) \Rightarrow (b*x_1^*+c=0) \bigwedge (b*x_1^*+c=0) \Rightarrow (x_1^* = FDS[b,c]) \Rightarrow \pi[a,b] \\ a \neq 0 \wedge (b*b-4*a*c=0) \Rightarrow \pi[a,b] \\ a \neq 0 \wedge b*b-4*a*c>0 \wedge (-b*b)+4*a*c>0 \Rightarrow \pi[a,b] \\ a \neq 0 \wedge b*b-4*a*c>0 \wedge -(b*b-4*a*c) \geq 0 \wedge (-b*b)+4*a*c>0 \Rightarrow \pi[a,b] \end{cases} \Longrightarrow \forall_{a,b} \pi[a,b]$$

We described a prototype implementation of a system which includes the capabilities of a verification conditions generator and formulae simplifier.

Using logical inference rules and algebraic manipulations, we were able to simplify the verification conditions. From this point on, relations between the program variables can be proved using advanced algebraic, combinatorial methods and decision procedures in various theories.

Chapter 3

Code Search Integration Facility into *Mindbreeze Enterprise Search*

3.1 Background

3.1.1 Information Retrieval

The necessity of information storage and retrieval earned attention since the amounts of information increased and it should be fast and accurate retrieved. Therefore, requirements of efficient methods for information retrieval were necessary; important data are then not ignored and the force of work and effort is not wasted. Moreover, the implementation of the information retrieval methods into computers (*retrieval systems*) provides effective, intelligent and fast search results within a huge amount of data.

The scenario of information storage and retrieval is as follows ([Rij79]): there exists a repository of documents and a person formulates requests (queries). The result of the query it is a set of documents satisfying the query. The perfect result can be obtained by reading all the documents, keeping in mind the interest one and discarding the others.

Introducing the computers to perform the retrieval of relevant documents for one needs, was, in a first phase, not to efficient because the search algorithms used *textual search* instead of using the semantics of the documents.

Next step was to characterize and structure the documents in such a way that it is relevant to a query and more, to be accessible in a reasonable amount of time. For the second feature, *indexing* came out to be a good technique. Nowadays, the information retrieval systems implement complex indexing algorithms, but there is still the question of the accuracy of the data retrieved.

Artificial Intelligence disciplines help in the accuracy problem by combining both syntax (*automatic text categorization [LRS99]*) and semantics (*semantic inference, ontology reasoning* [dB03]) of a document trying to overcome the query formulation problem.

Query Formulation Techniques

While most of the time was spent on constructing retrieval algorithms because it was thought that they play the central role in retrieving relevant information, William Frakes and Thomas Pole

proved by their work that constructing meaningful queries is also very important.

The most popular technique for formulating the queries, widely implemented in the code search engines, is *query by reformulation* ([Hen94]).

Query by reformulation technique is so popular because the retrieval of relevant information involves not only relationships between queries and their results in the incipient phase (when the user introduces the query), but also after (when the result set is displayed), when it is necessary an interaction between the user and the information retrieval system. This interaction yields to complex information retrieval.

The benefit of this technique is that it keeps the query language very simple by avoiding the big number of syntactical and semantical terms.

3.1.2 Source Code Retrieval

Source code retrieval, as a branch of the information retrieval, became a very important topic both in industry and in the research area in the last decades, mainly because of the large software systems developed for the market. Bright ideas applied in practice (e.g. source code engines) work together in the information retrieval, classification and extraction and therefore improving or solving the tasks of *software analysis, programming language objects lookup, code duplication, plagiarism detection,* etc.

In *software analysis*, source code retrieval helps the software developers in finding the right information that they need to modify, especially in the so-called legacy systems, and gives a quick and deep understanding of the software system. The problem with the legacy systems is that they are hard to understand because of the lack of the documentation and of the accelerated ageing of the technologies. Therefore, a good maintenance of the legacy systems consumes the software budget.

Source code retrieval is also important for *code reusability*, especially in large scale software products were there might exist the desire to consult the realization of the tasks from the old versions and their use or adaptation in the newer ones. In [Mil92], there are presented some issues, but also research problems, of the information retrieval in source code analysis for lookup purposes: *location* and *retrieval problem* (there have to be methods for fast search and access into libraries), *adaptation problem* (a software developer has to understand the code snippet in order to be able to modify it), etc.

The mechanism behind the source code retrieval ([MMM95]) is depicted in **Figure 3.1** and can be seen separately from user's and technical side, or combined.

1. From the user side: a problem is the subject of querying as it is understood by the user.
2. From the technical side, for retrieval purposes, the information from a repository must be encoded (indexed).
3. The query is compared with the indexes by the matcher and returns the instances that match the query.

There exists some issues from the both sides, all involving the loss of information:

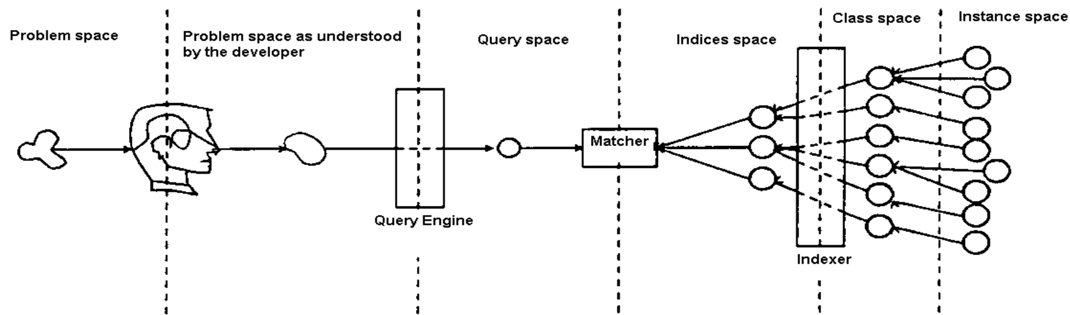


Figure 3.1: Source Code Retrieval Model

- different users can have different understandings of the same problem;
- the query language used by the users might be not too expressive for their needs;
- information can be lost during the indexing process.

3.1.2.1 History and State of the Art

The work from the early 90's proposed *classification schemes* for the classification and storage of the source code.

A *classification scheme* provides a network of predefined relationships, thus introducing some semantic information absent in free-text analysis.

Such classification schemes are: facet-based ([PD91]), sampling behavior ([PP93]), automatic retrieval ([MBK91]), user adaptable ([Hen97]), etc.

3.1.2.2 Facet-Based Scheme

The idea behind this scheme ([PD91]) is that the source code objects can be classified (faceted) in a limited number, and retrieved using a keyword associated to each facet can be attached a keyword (**T**able 3.1).

Callable	Object Type	Container
Method	Enum Field Interface Class	File Package

Table 3.1: A Simplified Faceted Scheme for Java Programming Language Objects

3.1.2.3 Sampling Behavior

In [PP93] the property that the source code, in distinction to the ordinary text, can be executed is exploited: a new method, called *behavior sampling* is proposed for the *automated retrieval of programming languages constructs*.

The method has to steps: in the first step, the user specifies a name to be searched for together with the specification. From the result set the user chooses those objects that can be executed and provides input values for them. The routines are then executed (second step) with these input values and give a certain result. Any routine whose output over the input matches the output specified by the user is retrieved, together with the specification.

3.1.2.4 Query Formulation Techniques in Source Code Retrieval

A solution to a stringent problem, *How to formulate queries that output the desired result?*, is given in [Hen94].

The author emphasizes that the problem with formulating queries comes from the fact that the user specifies them thinking of „what“ he wants as result while the objects are stored by functionality performed („how“) and these two „to do“ terms rarely coincide.

As a solution, these two techniques are combined in: *retrieval by reformulation* and *spreading activation*.

In *retrieval by reformulation* the user is asked to refine his previous queries while the intermediate results are stored for being consulted. The result set matches exactly the query.

By *spreading activation*, the result set matches partially the query. The partial matching relation is expressed using relations between the objects, not only information about them.

Latest approaches use technical and cognitive approaches to effective and quick code search retrieval ([YF02]).

From the cognitive point of view is well-known that a user has more levels of understanding the content of the source code files from a repository (**Figure 3.2**).

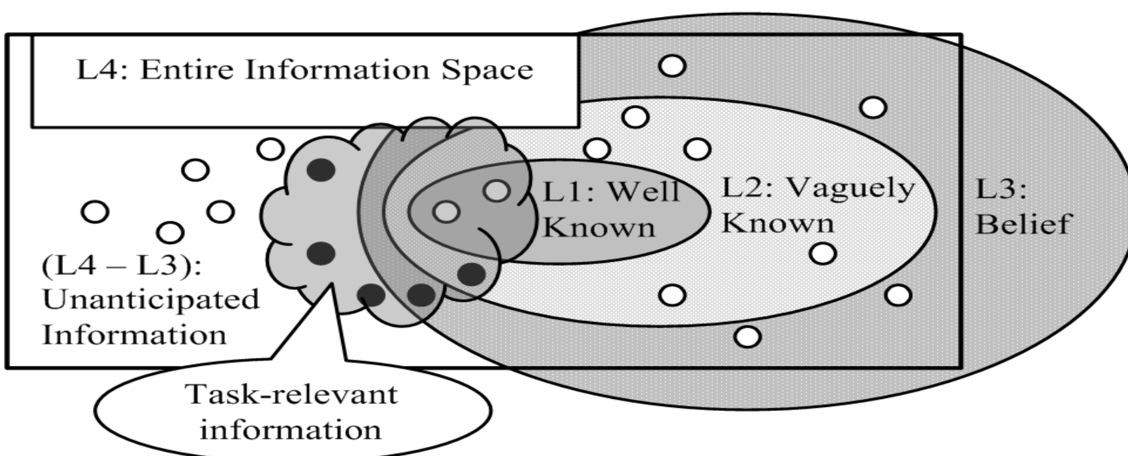


Figure 3.2: User’s Different Knowledge Levels

The programming objects known by the user are represented by L1, L2, L3; some are much known (L1) some are less (L3). The relevant information for the user (information presented in the cloud) could be found on the levels L3 and/or L4.

An ideal method should retrieve just the task-relevant information.

When faced to a search problem, the user concentrates his attention to the information from L2 and L3, forgetting about L3 and L4 where the result space is present. Hence, the repository has to be kept active to anticipate the user needs . The active feature of the repository is realized by repository exploration with implicit queries – the partially written programs.

These ideas were implemented in the *CodeBroker* system. It autonomously retrieves and displays the classes and methods from a repository. Moreover, it can be used as a programmers assistant in learning Java API or supports the components reuse for in-house repositories.

3.1.2.5 Code Search Engines

CodeBroker

CodeBroker ([YF02]) can be considered in the last generation of code search engines, developed in 2002. It has an *interface agent* (runs permanently in background) and a back-end *search engine*.

The interface agent has the role of inferring and extracting queries by permanent monitoring of the changes made by the developers. The search results obtained in the previous sessions are still included in the current search result although are not interesting for the current session.

The indexer memorizes items from the Java documentation (obtained from Javadoc).

Whenever a Javadoc comment or an object signature is entered in the interface agent, CodeBroker displays the results in three different abstraction layers.

The first level displays first predefined n programming language objects by their relevance. If the user is interested in the items of the results set, he has to interact with the second display level.

The second level is activated by mouse movements and allows also the refinement of the queries.

The third level displays complex information about the objects in a separate HTML window.

Maracatu

A notable success in the industry was recorded by the code search engine Maracatu.

The tool is based on the client-server technology; the client is a plug-in integrated in Eclipse IDE and the server is a web-application responsible for accessing the source code repositories.

It is based on pure text retrieval but also on the powerful faceted technique.

The new version of the system ([VDM$^+$07]) introduces *folksonomy* concepts in the existing approach. In this new model of code search engine, the users are encouraged to add new keywords (process known as *tagging*) to the new defined properties of a programming object. It is thought that in this way the gap between user's needs and the real storage scenario ([LdPdA04]) is avoided.

Some characteristics of the system are:

- the *tag cloud* area is displayed in the client interface; it represents the set of user defined tags that are frequently used in the querying process by the users and helps searching by tags;

- the *database* assures the persistence of some tags that are specific to some objects; they are accessible all the time by the tool;
- the *folksonomy classifier* insures the storage of the (predefined/user-defined) tags in a database.

3.1.3 Mindbreeze Enterprise Search

Mindbreeze Enterprise Search (MES) is a software product developed within Mindbreeze Software GmbH Linz. It can be used for searching within the file system, e-mail systems, content management systems, etc.

Architecture

The system has a *distributed architecture*: there is a main machine responsible for the configuration (via a web-based interface) of all the others nodes and services involved in the system installation.

The functionality of the system is ensured by a number of four services: *crawler, filter, index, query*, easy configurable.

MES Services One of the biggest features of the system is that custom data sources can be integrated if valid security certificate is provided.

The *Crawler Service* reads data from a specific data source and sends it to the *Filter Service*.

The *Filter Service* extracts data provided by the *Crawler Service* using a set of filters for standard file formats.

The *Index Service* permits the indexing (storage into a dictionary) of the information provided by the Crawler Service and it is accessible to the *Query Service*, responsible to generate a result set from the data provided by the Index Service.

3.2 Integration of Mindbreeze Code Search into Mindbreeze Enterprise Search

Mindbreeze Code Search (MCS) is a subsystem which is integrated into the existing *MES* infrastructure as a new data source (see **Section 3.2.3**) as well as from the crawling (see **Section 3.2.1.1**) and query enhancement by context interface (see **Section 3.2.2**) points of view.

MES infrastructure will suffer a small modification for *MCS Engine* integration, namely a database will be integrated (see **Section 3.2.1.7**).

The *MCS* architecture is depicted in the **Figure 3.4**.

3.2.1 Crawling and Indexing

One of the main attributes of the code search engines is to crawl as much information as possible such that it can be queried by the user.

In the crawling process we have to keep in mind the fact that source code files belong to the *structured documents* category. Therefore, they contain embedded information in their structure. Because of the mixture of *contents* (including natural language and programming language constructs) and *structure*, it is difficult and not recommendable that their content is stored directly in the *index* (*inverted index*) (facility provided by *MES*) or in the *database* (which will be integrated

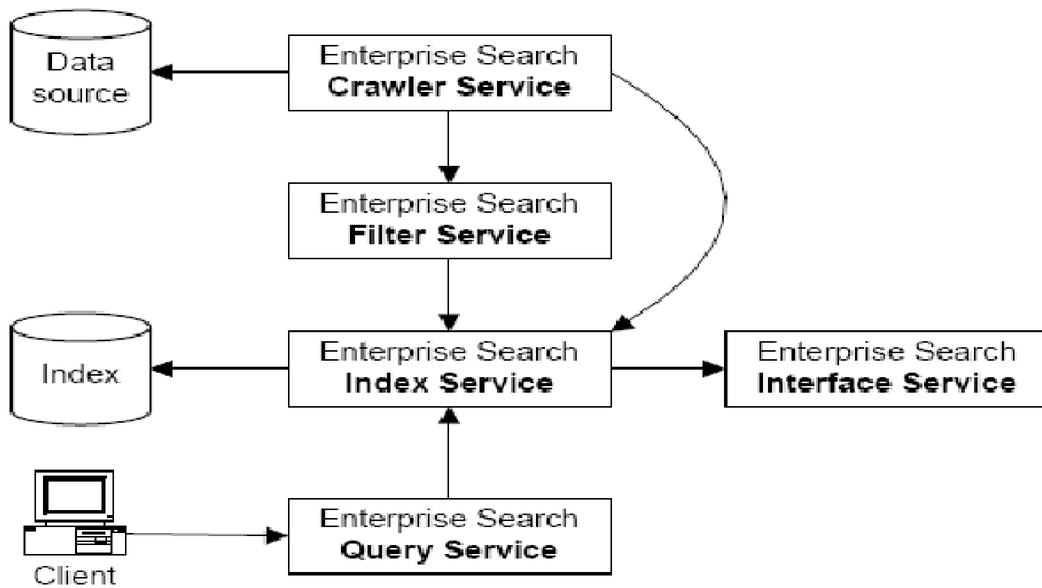


Figure 3.3: Mindbreeze Enterprise Search Services, Communication and Dependencies (from [***])

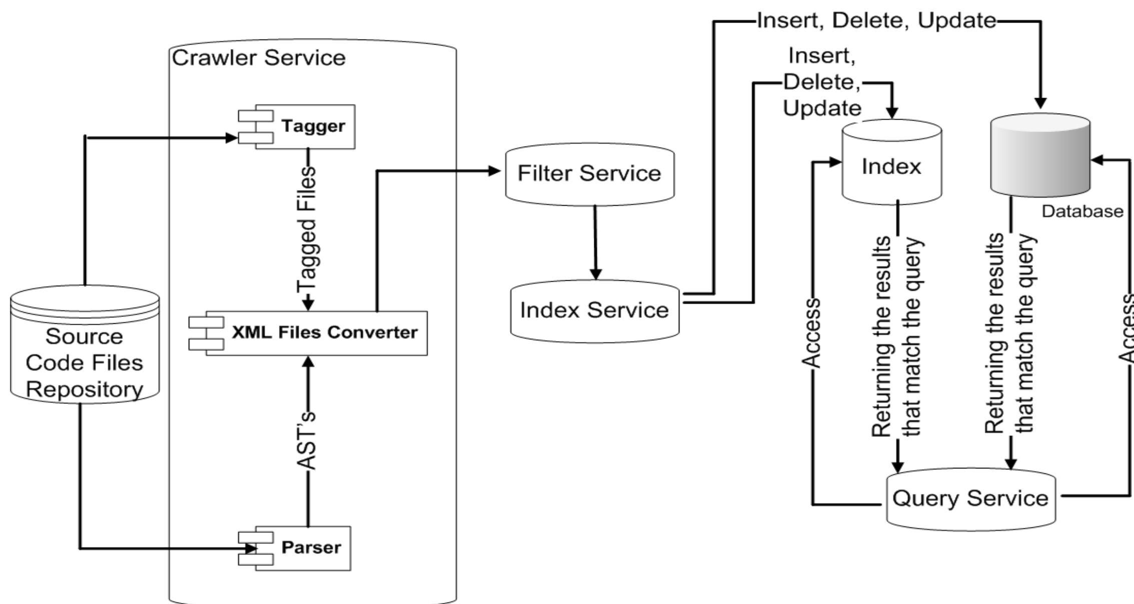


Figure 3.4: *Mindbreeze Code Search* Architecture

in the Mindbreeze Code Search). Therefore, as stated in many research papers ([Wil94]), it is better to structure the knowledge for improving significantly the retrieval results.

3.2.1.1 Crawling

Crawling is the operation of reading raw-data from a data source. The crawled data are sent to the Filter Service and then indexed.

In *MCS* the crawling phase involves two operations:

1. extract from the repository subject to crawling the *relevant objects* (*hit-types*) and *their attributes* (*metadata*) ;
2. structure the information retrieved.

Hit-Types Extraction

Hit-types and their metadata extraction is done using two tools: the tagger (*CTAGS*) and the parser (*Rats!*).

The tagger is applied first upon a repository of source code files insuring fast structuring of information, while the parser is applied in a second phase, adding more structured information, especially relationships between the hit types.

The tagger is specialized in retrieving *static information* about the hit types (e.g. signature, type, some modifiers, etc.), excepting the case of the hit-type „class" for which some *dynamic information* are obtained: if there are nested classes, the name(s) of the outer class(es) is obtained.

The parser is used mostly for *dynamic information* retrieval (relations between hit types) but also to obtain static information above the tagger capabilities.

The information retrieved by the tagger and parser can be structured on three levels:

- *first order information* - static information obtained from tagging;
- *second order information* - static information obtained from parsing;
- *third order information* - dynamic information.

CTAGS File Processing

As we mentioned above, the tagger CTAGS is used for a first phase processing. It is a program that attach labels (tags) to the relevant programming language objects existent in a source code file. The relevant information is kept in the form of a structured text file.

The tagger has support for many programming languages. We were interested mostly in: Java, C, Php, Javascript and Lisp.

For illustrating the tools usage and the other concepts we consider the following Java source-code snippet (`Person.java`):

Example 3.1.

```
package com.mycompany;

public class Person{
  public static class Address{
    public String street;
    public String location;
    public String zipcode;

    public int hashCode() { /* ... */ }
  }

  public String firstname;
  public String lastname;
  // ...

  public int hashCode(){
    int h = 23;
    h += h * 23 + ((firstname == null)? 0 :firstname.hashCode());
    h += h * 23 + ((lastname == null)? 0: lastname.hashCode());
    //...
    return h;
  }
}
```

CTAGS output file structure
The structure of a CTAGS file is: `tag_name⟨TAB⟩file_name⟨TAB⟩ex_cmd;"⟨TAB⟩extension_fields` where:

- *tag_name* is the name of the tag;
- *file_name* is the name of the file which is currently processed by CTAGS;
- *ex_cmd* is a command used to locate the tag within a file (search pattern or line number);
- *extension_fields* supplementary information for the object tag_name.

The *CTAGS* output file for the **Example 3.1** is:

```
Person.java c:\ctags57\Person.java 1;" file line:1 language:Java
...
Person c:\ctags57\Person.java /^public class Person{$/;" class line:2
language:Java

Address c:\ctags57\Person.java /^public static class Address{$/;" class
line:3 language:Java class:Person access:public
...
```

This output was obtained running *CTAGS* with the following options:

- `-u`: determines the listing of the tags increasingly after the line numbers;
- `--excmd=pattern` determines the type of *ex_cmd*, in this case the fully qualified name;
- `--extra=f` displays an extra line with information about the file analyzed;
- `--fields=aiKlmnsSt` adds, where applicable, extension fields for each tag: `+a` (access/ export information), `+i` (inheritance information), `+K` (kind of the tag information), `+l` (source file language information), `+m` (implementation information), `+n` (line number of the tag information), `+s` (scope information), `+S` (signature information), `+t` (”typedef” field information);
- `--tag-relative=yes` displays the file path relative to the directory containing the tag file, rather than relative to the current directory.

Rats! File Processing
A parser was chosen for retrieving more detailed information about the hit-types. This is possible because the parser analyzes the structure of tokens and determines the grammatical structure with respect to a formal grammar. Therefore, the hierarchy of the input text is captured and transformed into an abstract representation (usually syntax trees).

The *Rats!* parser generates grammars only in *Java* and *C* but was chosen as a tool for files preprocessing because is easily extensible.

Its features are:

- organizes the grammar into modules;
- generates parsing expressions instead of context-free grammars;
- generates automatically abstract syntax trees.

The integration of the parser was necessary because:

- the information that the tagger retrieves has not a very powerful semantic meaning: they refer to some static attributes of a single hit-type;
- we want to express queries where it is necessary to know the relationships between the hit-types;
- we want to extract also the comments from the source code files.

The information obtained for a hit-type from parsing is merged with the existing information obtained from tagging (for the same hit-type). Consistent merging is possible due to an identification algorithm (see **Section 3.2.1.4**) used by both tagger and parser: the information obtained from parsing is associated to the right hit-type whose metadata are already indexed. If the information expresses some relations between the hit-types, then it is stored into the database, otherwise it is indexed into the dictionary.

The processing of the source code files with the Rats! parser and its integration into the *MCS* (Crawler Service) is presented in detail in [Luk08].

3.2.1.2 Information Structuring

The tagger and the parser work together in order to retrieve as much information (static and dynamic) as possible for a hit-type. Because the information retrieved by the parser and the tagger does not have a uniform representation, we have to unify the representation of their output data.

This issues can be solved by structuring the information in a file with XML structure. The benefits gained are:

- we keep the structure of the document apart from the content;
- we obtain a lot of structured information.

Therefore a XML structure is given to the *CTAGS* output file, a XML element for each *CTAGS* output file entry. The information stored for each element (corresponding to a hit-type) is enriched by the parser, the only hard point is to add information to the right hit-type.

We came up with the following XML structure for the indexing needs:

Example 3.2.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<file>
    ...
    <Document category="CodeSearch" categoryclass="field" catinst="C:\ctags57\"
        key="tag:mindbreeze.com,2008/codesearch/unit:C:\ctags57\XMLToIndex.java#
            java:codesearch.indexer:XMLToIndex.Test:x#32"
        securitytoken="" title="x">
        <Metadata>
            <Meta key="file" value="C:\ctags57\XMLToIndex.java" />
            <Meta key="language" value="java" />
            <Meta key="name" value="x" />
            <Meta key="file" value="C:\ctags57\XMLToIndex.java" />
            <Meta key="type" value="field" />
            <Meta key="line" value="32" />
            <Meta key="language" value="Java" />
            <Meta key="class" value="XMLToIndex.Test" />
            <Meta key="access" value="default" />
            <Meta key="modifiers" value="int" />
        </Metadata>
        <Content>int x = 6;</Content>
    ...
</Document>
```

The *Document element attributes* have the following meaning:

- *category* – the name of the data source;
- *categoryclass* – the type of the hit-type analyzed;

- *catinst* – the directory which is crawled and indexed;
- *key* – the unique identifier of the hit-type;
- *security token* – unique identifier used for authorization purposes;
- *title* – the title of the hit-type.

Each *Document element* has a *Metadata* and a *Content nodes* and each *Metadata node* contains a set specific metadata with *key, value* attributes.

3.2.1.3 Indexing

In the *MCS* system, indexing means storing in an index (dictionary) data obtained from the *Filter Service* such that the *Query Service* is able to perform search within it.

Indexing is done by issuing a *FilterService* object and an indexing method.

Example 3.3.

```
try{
    FilterService filter = null;
    filter = InitializeMindbreeze();
    filter.indexRawData(content.getBytes(), metaData, indexURL,
                        "CodeSearch", catInst, key,
                        categoryClass,
                        "txt", title,
                        getCalendarFromDate(new Date()),
                        getCalendarFromDate(new Date()), null);
    catch (ServiceException ex) {}
    catch (MalformedURLException e) {}}
...
```

The method *indexRawData* is called once per translation unit (file). Its parameters are filled in by parsing the XML files previously created (see **Section 3.2.1.2**).

3.2.1.4 Creating and Referring Hit-Types Unique Keys

The hit-types metadata like category, category instance, language, return type, type, etc. do not uniquely identify a hit-type. Therefore, a unique identification of the hit-type has to exist for the scenarios when the user or system requests it for operations: displaying it in the client together with specific metadata (client requests) or insertion, update, deletion (system requests, more precisely IndexService requests). These scenarios involve working with an *inverted index (index)*, therefore there must exist hit-type unique identification.

Moreover, the uniqueness property of the hit-type key has to be related to the *category* and *categoryinstance* metadata of the object.

3.2.1.5 Hit-Types Keys at Defining Side

Before inserting a metadata into the index a unique key has to be constructed. We construct the key before structuring the content of a translation unit in a XML file.

A hit-type key has, at the defining side, the following structure:

```
tag:mindbreeze.com,2008:codesearch/unit:pathToFile#
    programmingLanguageName:hitType1Name:...:hitTypeKName#
    hitTypeKLineNumber
```

The key is structured in three parts: the first and the last are used for determining the hit location in the file, while the second one defines the hit-type programming language membership as well as hierarchical level of the hit-type analyzed.

The first part of the key contains a tagging prefix used for making distinction between the type of files indexed.

The hierarchical level is mostly helpful for object oriented programming hit types keys construction, where hierarchical structure is well-defined.

3.2.1.6 Hit-Types Keys at Referring Side

Referring the key of a hit-type appears in the the situations:

1. when re-queries are necessary such that the results fulfill the exact requirements of the user;

2. when syntactic ambiguities are present.

Case 1. Consider the following use-case: **Give me all the methods from the class Y.**
To solve this query the user has to perform the following steps:

1. textual search: „Y“;

2. restrict the result set by choosing just the *callable* artifact;

3. select the menu entry *Get methods* for the classes from the result set which might be interesting for the user.

Remark. In this prototype version, we categorized the hit-types in the following artifacts: *file* – includes the hit-type *file*, *package* – includes the hit-type *package*, *comment* – includes all types of *comments*, *object-type* – includes the hit-types: *class, enumerate, enum, macro, field* and *callable* – includes the hit-types *function* and *method*.

When the selection of a menu entry is present (context action) a reference to a hit-type specified by name is needed. We do not know anything about its key but we can define a pattern for it, more precisely a query which returns all the hit-types with the name *hitTypeName*, no matter on which hierarchy level and line number.

Case 2. We consider the following code snippet:

```
class ClassA{
    static void caller(){
            ClassB.callme(new ArrayList());
}
}
class ClassB{
    static void callme(java.utils.List list){
// ...
}
static void callme(myutils.List list){
// ...
}
}
package myutils;
public class List{
//...
}
```

In the previous example, the analyzer does not know which method *callme()* to take into consideration when the method *caller()* is called - no reference to any class List is explicitly presented, although the calls *callme(ArrayList)*, *callme(List)*, *callme(Object)* are all correct. Thus, the key generated uses only the name of the method.

In both cases there exists references to hit-types, without really knowing the exact one.

To solve this issues, we came up with a pattern for the keys generated at the referring side:

```
tag:mindbreeze.com,2008:codesearch/unit:*#java:*:hitTypeName#*
```

Although this introduces a certain amount of ambiguity, we prefer this query shape of the keys generated at the referring side because it is hit-type programming language membership independent.

3.2.1.7 The Database

A database is integrated in the *MES* architecture for storing information about the relations between the hit-types, thus for solving queries which involve third order information.

In this prototype version of *MCS* we used *SQLite* database engine.

The reasons why *SQLite* was used is that it is small, fast, simple and reliable, reasonable features for our needs.

Its important characteristics are:

- *zero-configuration* – it does not necessitates any setup procedure, no server procedure has to be started, stopped, or configured, no permission to be set;
- *serverless*, but allows multiple applications to access it;
- *single database file* is a disk file that if accessible, then it can read anything from the database;

We created tables for each CodeSearch artifact, for some of their metadata and the relations that were established between the hit-types of the artifacts. For instance, the table `callables` refers to the *callables* artifacts, the table `returns` refers to its return value, the table `is_descendant` refers to the artifact one level up in the hierarchy.

The client application that connects to the *SQLite* database engine uses a generic database access class for this purpose.

More details about the integration can be found in [Luk08].

3.2.2 Query Service Context Provider Interface Enhancements

Scenarios like 3.2.1.6, involving relations between the hit-types, are not possible with simple text and faceted search using the existing Mindbreeze Query Language.

How the services and interfaces interact as well as where is necessary an enhancement of the infrastructure or interfaces involved is depicted in the **Figure 3.5**.

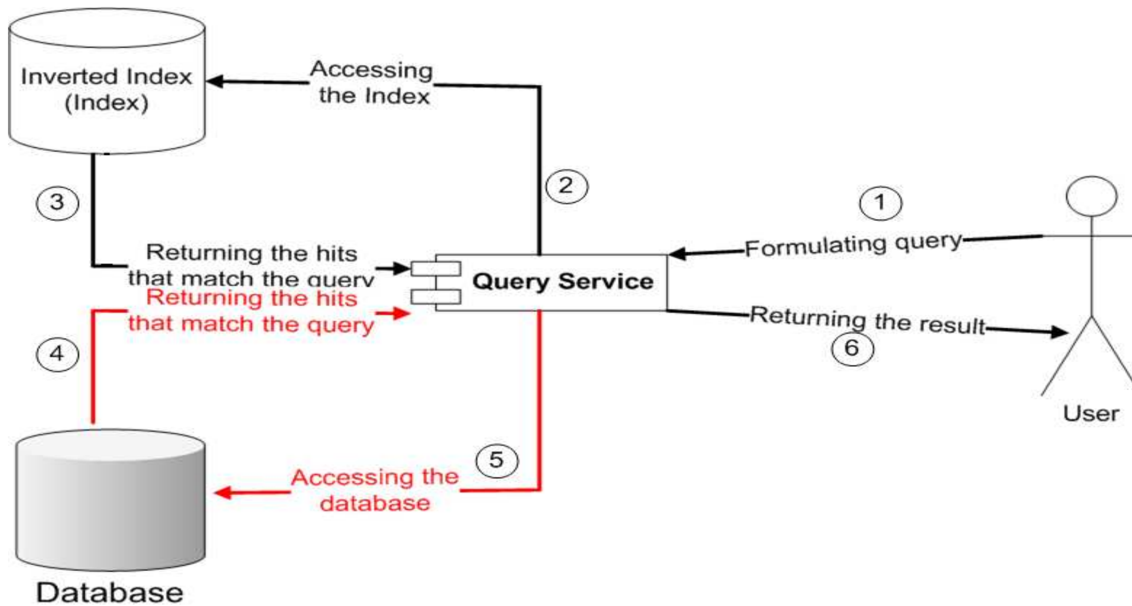


Figure 3.5: Scenario requiring Query Service Context Provider Interface Enhancements

What components of the *MCS* architecture interact and how is explained as follows.

Step 1. A query is introduced in the client interface by the user. The query is taken over by the Query Service which creates a Query object, an instance of the IQueryFactory interface and of Constraint class.

Step 2. The query might be performed into the index. In this case the user is querying static information about a hit-type. This kind of query is done by simple and faceted text searching, when the central role is played by the query language.

We assume the structure of the index like in **Table 3.3**.

Step 3. The hit types fulfilling the query are a subset of the hit types stored into the index **Table 3.3**.

id_1	$metadata_{id_1}$
id_2	$metadata_{id_2}$
...	...

Table 3.2: A Sample Index

id_{13}	$metadata_{id_{13}}$
id_{55}	$metadata_{id_{55}}$
id_{78}	$metadata_{id_{78}}$

Table 3.3: A Sample Result Set

Step 4. (Optional) We might want to refine the results obtained at **Step 3**. For this, it is necessary to formulate re-queries, either by textual and faceted search, either by certain actions on the individual hit-types from the result set.

We call *context action* the type of re-query that is performed on the hit-types from the result set by choosing a certain entry from the menu (context menu) associated to each of them.

Each hit-type from the result set is characterized by an unique id, and corresponding context icon and menu.

id_{13}	$ContextIcon_{id_{13}}$	$ContextMenu_{id_{13}}$
id_{55}	$ContextIcon_{id_{55}}$	$ContextMenu_{id_{55}}$
id_{78}	$ContextIcon_{id_{78}}$	$ContextMenu_{id_{78}}$

Table 3.4: A Sample Result Set and their Context Items

For context actions type re-queries, the context interface has to be enhanced.

Depending on the type of the context action (relationships between the hit-types), the database is accessed.

Step 5. If the user is asking for dynamical information about a hit-type (queries involving relations between the hit-types) then the query is performed into the database. In this case the Query Service Context Provider Interface has to be upgraded such that it provides CodeSearch specific items.

Step 6. The results are displayed to the user.

The enhancement of the Query Service interface with context actions is done by deploying on the server side an appropriate context provider which provides context items CodeSearch artifacts specific.

The implementation of CodeSearch context provider contains:

1. metadata keys for the identification of the appropriate context item for a search hit-type;

2. the types of items it provides: context icons, context menus, context actions;

3. the context items for a search hit (or artifact).

For building up a context provider we provide implementations for a ContextIcon class, namely Icon, and a ContextMenu class, namely Menu.

The class CodeSearchContextProvider, which implements the context provider, initializes the classes Icon and Menu thus implementing the three types of context items: context menu, context action and context icon.

A ContextIcon class represents artifact icons and context menu entries icons.

A ContextMenu class represents context menus, a set of context actions.

A ContextAction class represents a single context menu entry.

The two classes (Menu and Icon) (see **Figure 3.6**) provide parameterless constructors which are called via the CodeSearchContextProvider constructor.

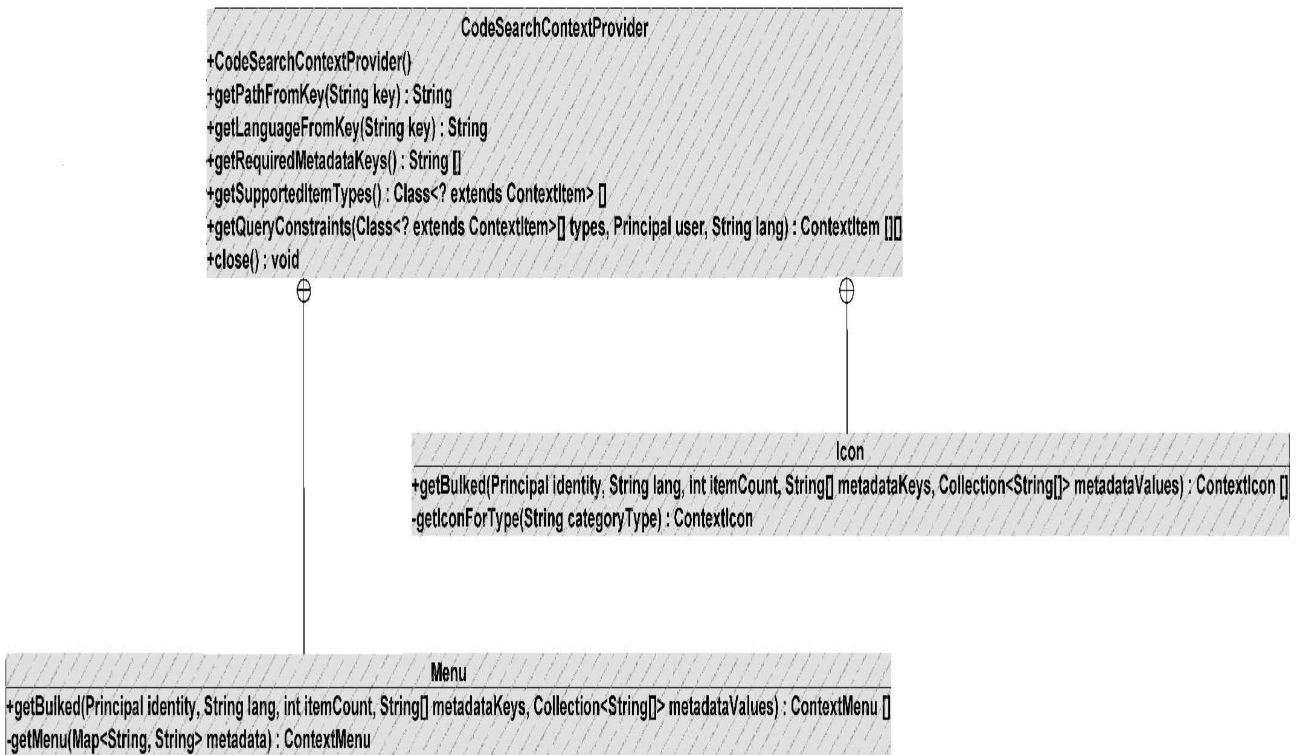


Figure 3.6: CodeSearchContextProvider Class and Inner Classes

The class Menu implements specific context menus and actions for the hit-types from the CodeSearch category. The methods implemented by this class are:

- public ContextMenu[] getBulked(Principal identity, String lang, int itemCount, String[] metadataKeys, Collection<String[]>) – returns a context menu for the given hits. It uses the method getMenu() for retrieving the hit-types stored in a Map structure.

- `private ContextMenu getMenu(Map<String, String> metadata)` – creates a context menu for the hit type specified by its metadata. For creating a suitable pattern action for a menu entry, the unique key of a hit type plays an important role. The way we approach the key construction turned out to be very fruitful because information like the category class, category instance, programming language needed for constructing pattern actions are easily identifiable in the key construction.

  I want to exemplify here with two types of re-query context actions; one type uses the key of the hit-type and the a special construct for formulating re-queries, similar to the Mindbreeze Query Language.

Example 3.4.
[Sample Context Actions Implementation]

```
//default context action
ContextAction openAction = new ContextAction();
openAction.setIcon(null);
openAction.setName("Open");
openAction.setPattern(getPathFromKey(metadata.get(MetadataKeys.DOCUMENT_KEY)));
actions.add(openAction);

String cc = metadata.get(getCategoryClass());
if (cc.equals("callable")){
    ContextAction similarCallableinLang = new ContextAction();
    similarCallableinLang.setIcon(null);
    similarCallableinLang.setName("Similar Callable in the Same Language");
    similarCallableinLang.setPattern("tag:mindbreeze.com,2007/actions/query/
        signature:"+ metadata.get(getSignature())+ " AND language:"+
        metadata.get(getLanguage()));
    actions.add(similarCallableinLang);
    }
...
```

The methods `getLanguage()` and `getSignature()` use the method `getRequiredMetadataKeys()` in order to retrieve the respective metadata for the hit-type. We mention that they have to be written exactly in the same way as they are stored in the index.

Example 3.5.
[The metadata keys required by this context provider]

```
public String[] getRequiredMetadataKeys() {
    return new String[] {
        MetadataKeys.HIT_GROUP_ID, MetadataKeys.CATEGORY_CLASS,
        MetadataKeys.DOCUMENT_KEY, "signature",
        "language", "categoryclass"
        } ;
    }
```

The class `Icon` provides icons specific for every search hit. It contains methods for retrieving a collection of search hits and then provides them icons depending on their categoryclass.

- `public ContextItem[][] getBulked(Class<? extends ContextItem>[] types, Principal arg1, String arg2, int itemCount, String[] metadataKeys, Collection<String[]> metadataValues)` – returns an array of context items for the hits specifies by their keys and values;
- `ContextIcon getIconForType(String categoryType)` – returns a context icon artifact specific.

These classes are becoming active after instantiating them in the constructor of the `CodeSearchContextProvider` class.

The `CodeSearchContextProvider` class represents a custom implementation of the existing `AbstractContextProvider` class and implements also the following methods:

- `public String[] getRequiredMetadataKeys()` – returns the metadata which are necessary for identifying the hit type;
- `public Class<? extends ContextItem>[] getSupportedItemTypes()` – returns an array of context items specific to the CodeSearch category;
- `public ContextItem[][] getQueryConstraints(Class<?extends ContextItem>[] types, Principal user, String lang)` – returns an array of context items for each CodeSearch artifact and creates, for each of them, a ContextSearchInConstraint object. A ContextSearchInConstraint object distinguishes the type of artifacts displayed to the end users: by query constraints involving a single search term - the name of the artifact and by unique names and icons for the artifacts.
- methods for retrieving parts of the key of the hit-type needed for various context actions implementation.

The auxiliary class `AbstractTypeInfo.java` provides implementation for a method returning the contents of the files representing images which is a mandatory information for context icons creation. It is used by the classes `FileTypeInfo, PackageTypeInfo, CallableTypeInfo, ObjectTypeInfo, CommentsTypeInfo` respectively, which creates `ContextIcon` objects for each CodeSearch artifact.

The static structural view of the subsystem which implements the CodeSearch context provider is depicted in the UML diagram from the **Figure 3.7**:

3.2.3 CodeSearch Data Source Integration

In this chapter we present how the functionality of *MES* can be extended by adding custom components to the existing services.

We integrated into the system the custom data source CodeSearch, standing for the files which have a certain structure, namely are written in a programming language.

The integration of the custom data sources, in particular CodeSearch category, depends on:

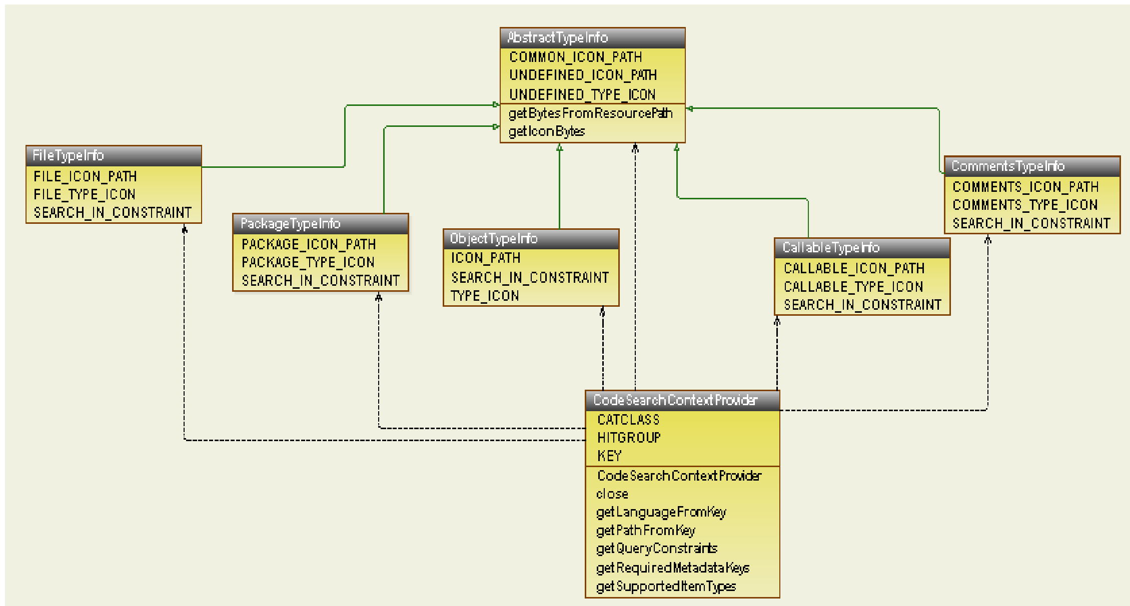


Figure 3.7: CodeSearch Context Provider Classes

1. a unique category identifier and category display name;

2. the description of the metadata for each CodeSearch artifact;

3. a graphical icon representing the category in the user interface.

These requirements are solved by deploying on the server a *category descriptor file* and *a category icon file*, respectively.

Moreover, we want that the results displayed to the end users to have context specific information. For this purpose, we have to provide to the Query Service (again by deploying on the server) an appropriate *context provider implementation* for the CodeSearch data source.

Deployment is done by the command line tool `mesextension` used for the infrastructure configuration. In was created during the *MES* development and used installing or uninstalling context or authorization provider plug-ins, category descriptors and category icons.

Usage: `mesextension [OPTIONS] install|uninstall`

3.2.3.1 Deploying the CodeSearch Category Descriptor, Category Icon and Context Provider

Deploying the CodeSearch Category Descriptor

A category descriptor is a XML file which contains the unique identifier, the display name and descriptions for each artifact for the data source CodeSearch.

A fragment of the CodeSearch category descriptor (`CodeSearchCategoryDescriptor.xml`) is listed:

Example 3.6.

```
...
categorytype id="objecttype">
    <metadata>
        <metadatum id="location" selectable="false">
            <name xml:lang="de">Ort</name>
            <name xml:lang="en">Location</name>
        </metadatum>
        <metadatum id="author" selectable="false">
            <name xml:lang="de">Autor</name>
            <name xml:lang="en">Author</name>
        </metadatum>
        <!--  specific to callable -->
        <metadatum id="file" selectable="true">
            <name xml:lang="de">File</name>
            <name xml:lang="en">File</name>
        </metadatum>
...
        <metadatum id="class" selectable="true">
            <name xml:lang="de">Klasse</name>
            <name xml:lang="en">Class</name>
        </metadatum>
        <metadatum id="modifiers" selectable="true">
            <name xml:lang="de">Modifiers</name>
            <name xml:lang="en">Modifiers</name>
        </metadatum>
...
    </metadata>
</categorytype>
```

The deploying command is:

```
mesextension --interface=categorydescriptor
--category=CodeSearch --file=codeSearchCategoryDescriptor.xml install
```

The schema for a category descriptor file is in **Table 3.2.3.1** (from [***], simplified).

Deploying the CodeSearch Category Icon

Mindbreeze Enterprise Search provides an interface that links a category icon with an existing category, the only requirements being: the file representing the icon must have PNG format with the image dimensions 16x16 pixels.

```
mesextension --interface=categoryicon --category=CodeSearch
--file=mcs-logo-16x16.png install
```

category@id	The unique category identifier.
category/name	A display name for the category.
category/metadata	The category metadata.
category/metadatum	The category metadatum (optional)
category/metadatum@id	The category metadatum identifier
category/metadatum@selectable	Specifies that the metadatum should be selectable in *MCS* enabled applications. This attribute can be taken into consideration for search result refinement, for example.
category/metadatum/name	A display name for the meta datum

Table 3.5: Category Descriptor Schema Documentation

Deploying the CodeSearch Context Provider

There are two steps for installing the CodeSearch context provider:

1. properly installation of the context provider.
   `mesextension --interface=context --category=CodeSearch --library=mcs-context.jar install`

2. providing access to it for insuring that the results returned by the system obey the authorization rules of the original data source.
   `mesextension --interface=access --category=CodeSearch --library=CodeSearchAuthorization.jar install`

3.2.4 Mindbreeze Code Search - Use Cases

We present the system as it is shown to the user and how interaction with it is possible, including explorative search and data-drilling operations. We exemplify with two use-cases: *Find all the hit-types named* ***sort*** and *Find all the methods named* ***sort***.

We will consider the following Java code snippet for exemplification:

Example 3.7.

```
package algorithms;
...
public class Sort{
  ...
  public static void sort (double[] a){
    ...
    sort(a, 0, a.length-1);
  }
  public static void sort (double[] a, int left, int right){
    ...
  }
```

...

Use Case 1 - Find all the hit-types named sort
This example is a typical text search in the CodeSearch data source which is provided to the main node.

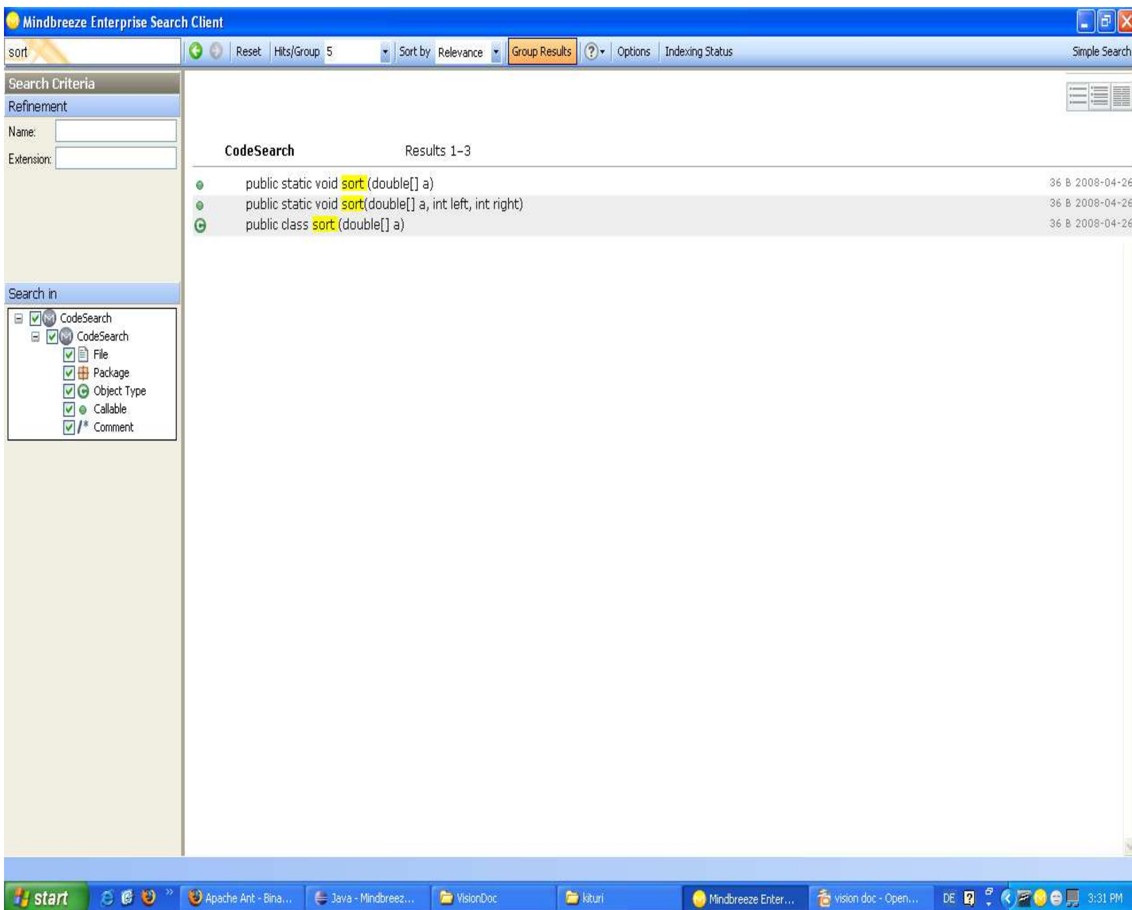


Figure 3.8: Find all the hit-types named **sort**

With this result set (see **Figure 3.8**), the user can perform drill operations, namely refinements (in the current state are displayed those common for all the artifacts), can display the information about it on different levels (current is displayed the first level) and can choose that just a subset of artifacts is displayed.

Use Case 2 - Find all the methods named sort
This example is representative for illustrating the explorative search in the Mindbreeze Code Search system: for retrieving a certain hit-type, from a certain artifact, first is performed a text based search and then, from the domain of results, the user can choose the task-relevant artifacts.

- *Step 1* is identical to the *Use Case 1* (see **Figure 3.8**)

- in *Step 2* user chooses „callable“ artifact from the predefined artifacts and gets the desired results (see **Figure 3.9**)

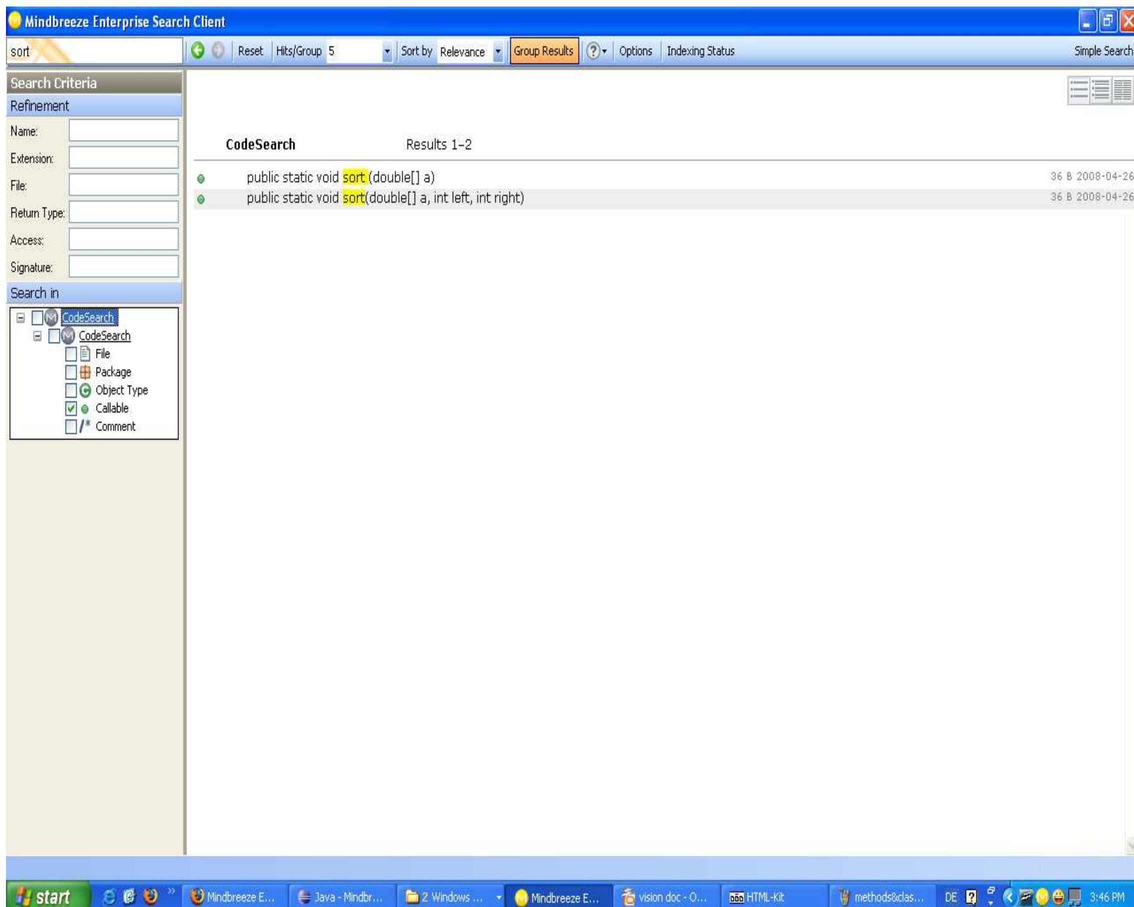


Figure 3.9: Find all the methods named **sort**

Representation of Information on Levels

MCS Engine is also able to display different information about a hit-type, depending on the level chosen. This facility is inherited from the base product (*MES*), we just defined the structure of information on each level.

Level 1: On this level we display for each item of the result set the title, the size of the file where it is declared and the modification date (see **Figure 3.9**).

Level 2: Besides the information presented at the *Level 1*, we display the [available] information corresponding to the fields in the refinements section see (see **Figure 3.10**).

Level 3: In this level we add also some representative code for the hit-type (from its body), for example the other hit types with the same name, or similar (see **Figure 3.11**).

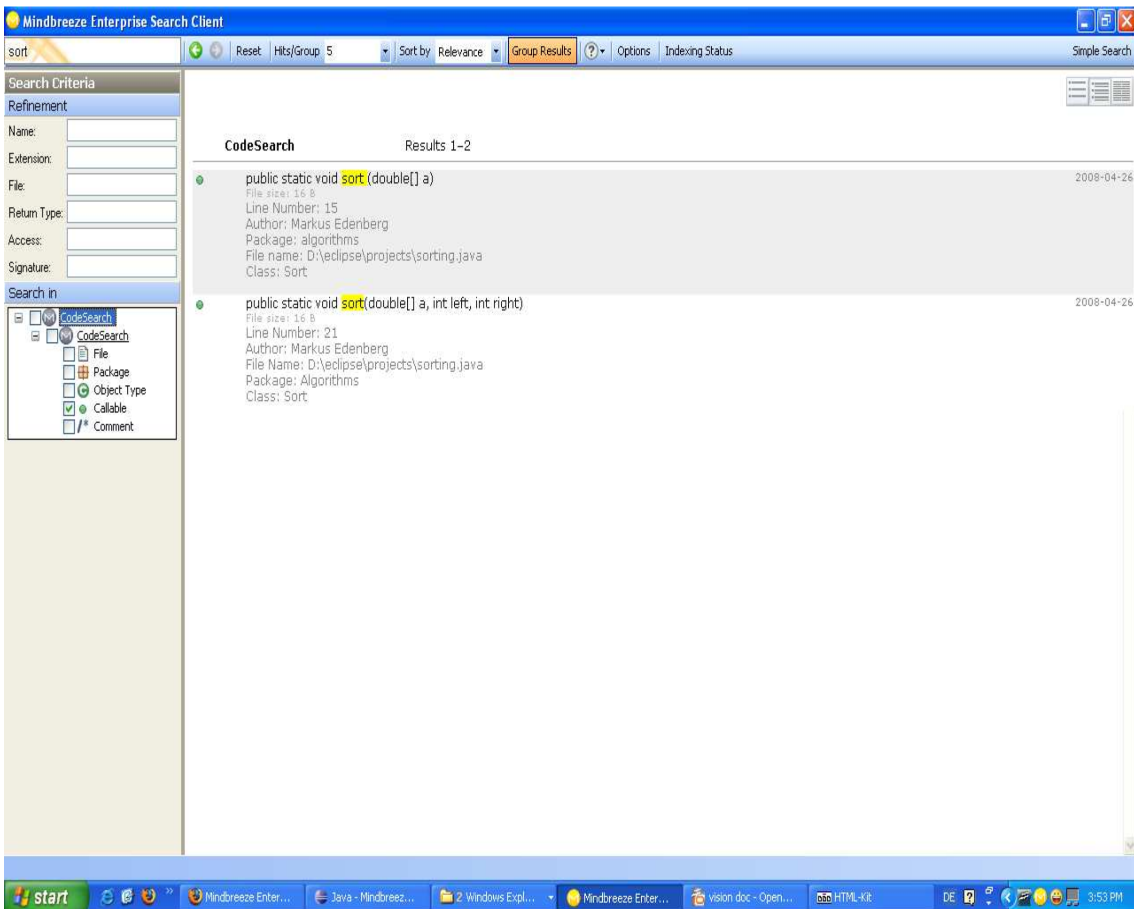


Figure 3.10: Find all the methods named **sort** (information level: 2)

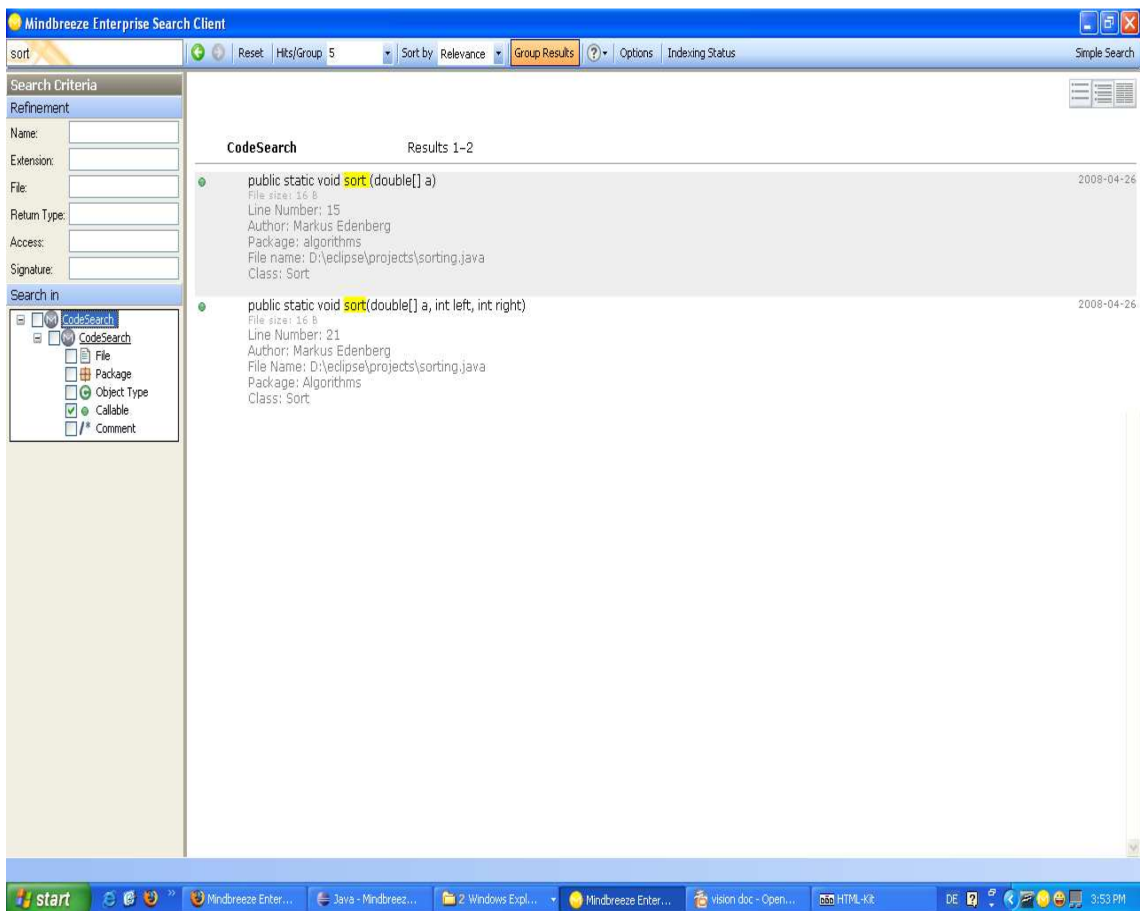


Figure 3.11: Find all the methods named **sort** (information level: 3)

Chapter 4

Conclusions

In this thesis we presented two approaches for analyzing the source code: formal static analysis and retrieval.

The formal static analysis method of source code is based on forward symbolic execution and functional semantics. A prototype environment (`FwdVCG`) combining these techniques was implemented which generates the verification conditions necessary for checking the imperative program correctness. We implemented it in the *Theorema* system and we tested it on programs written in our mini-programming language.

For checking the validity of the verification conditions we built a prototype simplifier that reduces the verification conditions to (systems of) equalities and inequalities. The simplified formulae were obtained using first order logic inference rules, truth constants and simple algebraic simplifications.

As future work we want to apply and automate advanced algebraic and combinatorial algorithms and methods in order to check their validity. A promising approach seems to be the Fourier-Motzkin Elimination method ([Sch86]), especially for systems of inequalities with small number of constraints and variables because of the time complexity reasons.

The automated retrieval of source code objects was done by extending the functionality of the *MES*, namely: (i) implementing a custom crawler, CodeSearch category specific, that interacts with MES services; (ii) integration of the CodeSearch data source into the *MES* infrastructure for making possible the search within the source code files; (iii) enhancing the user interface with CodeSearch category specific context actions and data-drilling operations.

Some of the open functionalities of the *MCS* are: (i) the implementation of complex context actions which require the information stored in the database; (ii) the synchronization of the index with the data obtained from crawling to avoid inconsistencies; (iii) add authentication and authorization rules CodeSearch category specific, etc.

We also propose a design solution for the integration of `FwdVCG` into *Mindbreeze Code Search*, which shows how the formal methods can be used in real software programs rather than being used for verification of small programs.

Problem Specification. We are interested in the retrieval of the callables artifacts and their specification from the source code files repositories crawled.

Due to the Rats! parser, we are able to retrieve just the Java methods and the JML specifica-

tions. After the retrieval of their content, they have to be transformed into *Theorema* syntax and sent as input to `FwdVCG` which generates the corresponding verification conditions.

Retrieving Methods and Specifications. The attempts of retrieving the methods together with their specification is above the capabilities of the tagger. The problem is solved by the parser which is able to retrieve JML specifications and methods body from the Java source code files.

***XForm* Queries for Abstract Syntax Tree Transformations.** For querying and transforming the abstract syntax trees for source code files we use *XForm*. The language allows queries for retrieving meaningful metadata about the hit-types, but we are interested in the queries for special type comments retrieval (JML-like specification) and content of the methods.

The language *XForm* has a syntax and semantics similar to *XPath 2.0* XML query language, but additional has support for abstract syntax trees (AST) transformations and queries.

An *XForm query* is composed of a serie of one or more comma-separated expressions, each returning a sequence of tree items. All *XForm* queries generate ordered sets of tree items that match the specified query template. A tree item may be a node within the AST, a string, or null. Nested sequences are not allowed.

The model of tree built by the parser is in the **Figure 4.1**

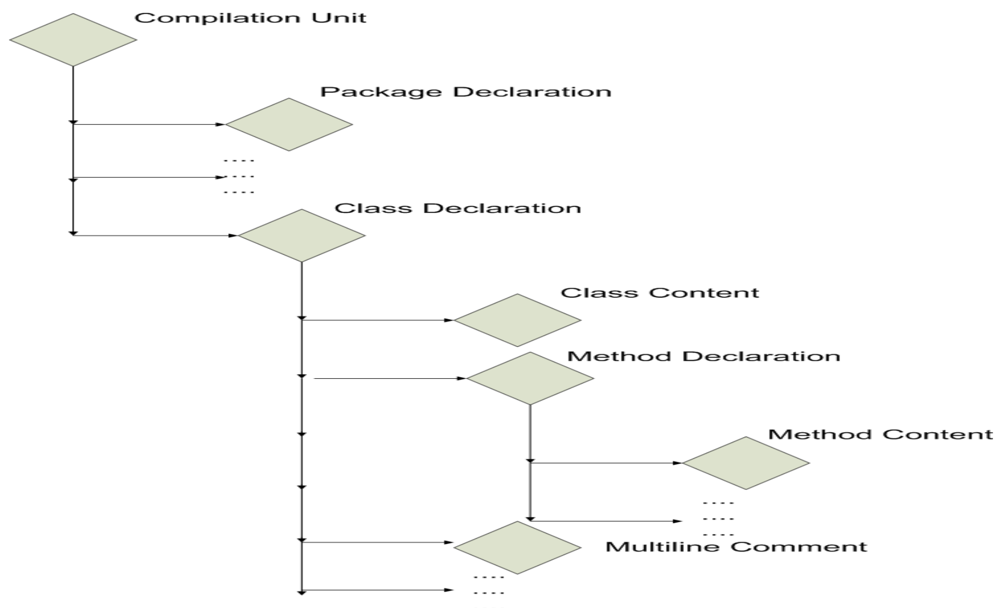


Figure 4.1: The Model of the Syntax Tree

When parsing a Java source code file, the model of AST built has the disadvantage that the *comment (specification) node* is not located right before or after the method declaration; they are inserted as nodes on the same level as the *class declaration node*. Because of this drawback the AST's processing and the retrieval of the right specification for a method necessitates a long processing time and a complex algorithm that associates to a method the right specification.

The implementation details for retrieving the content of the methods and the specifications can be found in [Luk08].

How the database is used. For each method and specification we build the unique key and retrieve its content. For making persistent the information, we insert it in the database, where the tables for *callables* and *specifications* have the structure from the tables **4.1** and **4.2**.

```
CREATE TABLE IF NOT EXISTS callables (
key INTEGER PRIMARY KEY,
hit_def_key TEXT NOT NULL,
specification_id INTEGER)
```

Table 4.1: Callables Table

```
CREATE TABLE IF NOT EXISTS specifications (
key INTEGER PRIMARY KEY,
hit_def_key TEXT NOT NULL)
```

Table 4.2: Specification Table

In the *callables* table there must exist a column where the *id* of the corresponding specification has to be inserted, if the specification exists.
With the query:
`SELECT callables.hit_def_key, specifications.hit_def_key FROM callables, specifications WHERE callables.specification_id = specifications.key`,
one can obtain the name of the method and the corresponding specification that are further transformed into *Theorema* syntax, syntax recognized by our verification conditions generator.

For full integration of the methods, the Java methods have to be transformed such that the `Return` statement occurs on every branch. On the other side, we must add support for the `while` statement in the program analysis with `FwdVCG`.

Bibliography

[***] ***. *Mindbreeze Enterprise Search 3.0.1 SDK*. Mindbreeze Software GmbH, Honauerstrasse 4, 4020 Linz, Austria.

[ABH+07] W. Ahrendt, B. Beckert, R. Hähnle, P. Rümmer, and P. Schmitt. Verifying object-oriented programs with KeY: A tutorial. In *5th International Symposium on Formal Methods for Components and Objects, Amsterdam, The Netherlands*, volume 4709 of *LNCS*, pages 70–101. Springer, 2007.

[ADL+79] P. Asirelli, P. Degano, G. Levi, A. Martèlli, U. Montanari, G. Pacini, F. Sirovich, and F. Turini. A flexible environment for program development based on a symbolic interpreter. In *ICSE '79: Proceedings of the 4th international conference on Software engineering*, pages 251–263, Piscataway, NJ, USA, 1979. IEEE Press.

[B. 06a] B. Cook and A. Podelski and A. Rybalchenko. Termination proofs for systems code. *SIGPLAN Not.*, 41(6):415–426, 2006.

[B. 06b] B. Gulavani and T. Henzinger and Y. Kannan and A. Nori and S. Rajamani. Synergy: A new algorithm for property checking. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 117–127, New York, NY, USA, 2006. ACM.

[BCJ+06] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[BEL75] R. Boyer, B. Elspas, and K. Levitt. SELECT - a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245, New York, NY, USA, 1975. ACM.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM.

[BPS00] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience*, 30(7):775–802, 2000.

[Cla76] L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *Software Engineering, IEEE Transactions on*, SE–2(3):215–222, 1976.

[Cow88] D. Coward. Symbolic execution systems – a review. *Softw. Eng. J.*, 3(6):229–239, 1988.

[CPDGP01] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé. Using symbolic execution for verifying safety-critical systems. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 142–151, New York, NY, USA, 2001. ACM.

[CS02] M. Colón and H. Sipma. Practical Methods for Proving Program Termination. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 442–454, London, UK, 2002. Springer-Verlag.

[dB03] J. de Bruijn. Using ontologies – enabling knowledge sharing and reuse on the semantic web. Technical Report DERI-2003-10-29, DERI, 2003.

[Deu73] L. Deutsch. *An interactive program verifier*. PhD thesis, University of California - Berkley, USA, 1973.

[Dro90] G. Dromey. *Program derivation: the development of programs from specifications*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[GAL+06] V. Garcia, E. Almeida, L. Lisboa, A. Martins, S. Meira, D. Lucredio, and R. Fortes. Toward a code search engine based on the state-of-art and practice. In *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 61–70, Washington, DC, USA, 2006. IEEE Computer Society.

[God97] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM.

[Hat95] L. Hatton. *Safer C: Developing Software for in High-Integrity and Safety-Critical Systems*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[Hen94] S. Henninger. Using Iterative Refinement to Find Reusable Software. *IEEE Software*, 11(5):48–59, 1994.

[Hen97] S. Henninger. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. *ACM Trans. Softw. Eng. Methodol.*, 6(2):111–140, 1997.

[HJMS] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction.

[HK76] S. Hantler and J. King. An Introduction to Proving the Correctness of Programs. *ACM Comput. Surv.*, 8(3):331–353, 1976.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.

[How73] W. Howden. Methodology for the authomatic generation of program test data. Technical Report 41, Standford Artificial Inteligence Laboratory, Standford, California, 1973.

[KF70] J. King and R. Floyd. An interpretation oriented theorem prover over integers. In *STOC '70: Proceedings of the second annual ACM symposium on Theory of computing*, pages 169–179, New York, NY, USA, 1970. ACM.

[Kin70] J. King. *A program verifier*. PhD thesis, Pittsburgh, PA, USA, 1970.

[Kin75] J. King. A new approach to program testing. In *Proceedings of the international conference on Reliable software*, pages 228–233, New York, NY, USA, 1975. ACM.

[Kin76] J. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[Kir99] M. Kirchner. Program Verification with the Mathematical Software System Theorema. Technical Report 99-16, July 1999.

[Kov07] L. Kovacs. *Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema*. PhD thesis, RISC, Johannes Kepler University Linz, Austria, October 2007. RISC Technical Report No. 07-16.

[KPC] J. Kiniri, E. Poll, and D. Cok. Design by Contract and Automatic Verification for Java with JML and ESC/Java2.

[KPV03] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing, 2003.

[LdPdA04] D. Lucredio, A. do Prado, and E. de Almeida. A Survey on Software Components Search and Retrieval. In *EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference*, pages 152–159, Washington, DC, USA, 2004. IEEE Computer Society.

[LM08] R. Leino and R. Monahan. Program Verification Using the Spec# Programming System. Tutorial presented at ETAPS, 2008.

[LRS99] W. Lam, M. Ruiz, and P. Srinivasan. Automatic Text Categorization and Its Application to Text Retrieval. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):865–879, 1999.

[LSS84] J. Loeckx, K. Sieber, and R. Stansifer. *The foundations of program verification*. John Wiley & Sons, Inc., New York, NY, USA, 1984.

[Luk08] L. Lukacs. Automated Modeling and Analysis of Object Oriented Source Code. Master's thesis, Johannes Kepler University, Linz, Austria, 2008. ongoing.

[MBK91] Y. Maarek, D. Berry, and G. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng.*, 17(8):800–813, 1991.

[McC63] John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[Mil92] K. Mills. Requirements Engineering for Software Reuse, 1992.

[MMM95] H. Mili, F. Mili, and A. Mili. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, 1995.

[PD91] R. Prieto-Díaz. Implementing Faceted Classification for Software Reuse. *Commun. ACM*, 34(5):88–97, 1991.

[PJ03] N. Popov and T. Jebelean. A Practical Approach to Verification of Recursive Programs in Theorema. In T. Jebelean, V. Negru, and A. Popovici, editors, *Proceedings of SYNASC'03 (International Workshop on Symbolic and Numeric Algorithms for Scientific Computing Timisoara, Romania, 2003)*, pages 329–332. Mirton, October 2003.

[PP93] A. Podgurski and L. Pierce. Retrieving Reusable Software by Sampling Behavior. *ACM Trans. Softw. Eng. Methodol.*, 2(3):286–303, 1993.

[Pug92] W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[PV04] C. Pasareanu and W. Visser. ”verification of java programs using symbolic execution and invariant generation”. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software*, volume 2989 of *LNCS*. Springer-Verlag, 2004.

[Rij79] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 1979.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, USA, 1986.

[Sch00] J. Schumann. *Automated Theorem Proving in Software Engineering*. Springer-Verlag, 2000.

[Top75] R. Topor. *Interactive Program Verification using Virtual Programs*. PhD thesis, University of Edinburgh, Scotland, 1975.

[VDM+07] T. Vanderlei, F. Durao, A. Martins, V. Garcia, E. Almeida, and S. Meira. A Cooperative Classification Mechanism for Search and Retrieval Software Components. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 866–871, New York, NY, USA, 2007. ACM.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, 2000.

[Wil94] R. Wilkinson. Effective Retrieval of Structured Documents. In *SIGIR '94: Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 311–317, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[Wol03] S. Wolfram. *The Mathematica Book, Fifth Edition*. Wolfram Media, 2003.

[YF02] Y. Ye and G. Fischer. Supporting Reuse by Delivering Task-Relevant and Personalized Information. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 513–523, New York, NY, USA, 2002. ACM.

CURRICULUM VITAE

MĂDĂLINA ERAŞCU

Personal Data

Full Name:	Mădălina Eraşcu
E-mail address:	merascu@risc.uni-linz.ac.at
Date and Place of Birth:	October 10, 1983, Oraviţa, Romania
Nationality:	Romanian.

Education

- **Master Studies** (2006 – now) in Computer Science
  - at Johannes Kepler University Linz, Austria, as Erasmus – Socrates student
  - at International School for Informatics (Johannes Kepler University) – Informatics: Engineering and Management
    Master thesis: „Automated Formal Static Analysis and Retrieval of Source Code"
    Thesis advisor: a.Univ.-Prof. Dr. Tudor Jebelean
- **Bachelor Studies** (2002 – 2006) in Computer Science
  West University of Timişoara, Romania.
  Bachelor thesis: „XML Web Services using ADO.NET"
  Thesis advisor: Lect. Dr. Florin Fortiş
- **High School Studies** (1998 – 2002)
  Theoretical High School "General Dragalina", Oraviţa, Romania.

Publications and Talks

- M. Erascu and T. Jebelean. *Practical Program Verification by Forward Symbolic Execution: Correctness and Examples.* In: Austrian-Japan Workshop on Symbolic Computation in Software Science, Bruno Buchberger, Tetsuo Ida, Temur Kutsia (ed.), pp. 47-56. 2008.
- M. Erascu and T. Jebelean. *Verification of Imperative Programs using Forward Reasoning.* Apr., 11-14, 2007. Contributed talk at SFB Statusseminar, Strobl, Austria.
- M. Erascu and T. Jebelean. *Verification of Imperative Programs using Symbolic Execution and Forward Reasoning in the Theorema System.* RISC Linz. Technical report no. 07-12, July 1 2007. Presented at First Austria-Japan Workshop on Symbolic Computation and Software Verification, Linz, Austria, July 1st 2007

Eidesstattliche Erklärung

Ich erkläre an Eides statt, da ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfat habe. Ich habe dazu keine weiteren als die angeführten Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet.

Linz, July 2008.

....
Mădălina Eraşcu