Consistency Techniques for Test Data Generation

Thèse

soumise par **Nguyen Tran Sy**

en vue de l'obtention du grade de
Docteur en Sciences Appliquées
au Département d'Ingénierie Informatique
de l' Université catholique de Louvain

Membres du Jury :

Prof. Jean-Didier **Legat** (Université catholique de Louvain, *Président*)

Prof. Yves **Deville** (Université catholique de Louvain, *Promoteur*)

Prof. Baudouin **Le Charlier** (Université catholique de Louvain)

Prof. Michel **Rueher** (Université Nice Sophia Antipolis)

Prof. Charles **Pecheur** (Université catholique de Louvain)

Dr. Bruno **Marre** (CEA Saclay, CNRS)

Université catholique de Louvain

May, 2005

Consistency Techniques for Test Data Generation

Nguyen Tran Sy
Université catholique de Louvain, 2005

Abstract

This thesis[1] presents a new approach for automated test data generation of imperative programs containing *integer*, *boolean* and/or *float* variables. A test program (with procedure calls) is represented by an Interprocedural Control Flow Graph (ICFG). The classical testing criteria (statement, branch, and path coverage), widely used in unit testing, are extended to the ICFG.

Path coverage is the core of our approach. Given a specified path of the ICFG, a path constraint is derived and solved to obtain a test case. The constraint solving is carried out based on a consistency notion.

For statement (and branch) coverage, paths reaching a specified node or branch are dynamically constructed. The search for suitable paths is guided by the interprocedural control dependences of the program. The search is also pruned by our consistency filter. Finally, test data are generated by the application of the proposed path coverage algorithm. We also propose a dynamic approach to statement coverage, by combining random test data generation, program execution and our path coverage method — but no implementation has been realized for this approach.

A prototype system —called COTTAGE and consisting of 13,000 Java lines of code— implements our approach for C programs. For each generated test data, the system also automatically generates an instrumented C program, allowing the user to verify the correctness of the test data. Experimental results, including complex numerical programs from [55], demonstrate the feasibility of the method and the efficiency of the COTTAGE system, as well as its versatility and flexibility to different classes of problems (integer and/or float variables; arrays, procedures, path coverage, statement coverage).

Keywords software testing, test data generation, path coverage, statement coverage, procedures, arrays, constraint satisfaction, consistency

[1]This thesis is an extended version of [63, 64, 62]

Road map

This thesis is divided into four parts, each in turn is divided into chapters. Although the titles of these parts and chapters reflect rather well their contents, we give below a short description for each part and its related chapters.

Part I lays a foundation for all other parts, presenting our problem of test data generation and its context, as well as necessary background on constraint programming used and developed in this work to tackle such problem. We first study, in Chapter 1, the testing context in which the problem of test data generation is raised. Results and contributions of our work are also given. Chapter 2 then describes related work. Chapter 3 provides the background and notations. Finally, Chapter 4 proposes a new framework on interval logic, developed to handle constraints involved in our generation of test data.

In Part II, we describe our consistency approach to the problem of test data generation. We first give, in Chapter 5, an overview of our approach. Chapter 6 illustrates the generation of path constraints (representing constraints for a path of a program under test). Our underlying consistency technique suitable for test data generation is then proposed in Chapter 7. Chapter 8 describes the generation of test data traversing a specified path in the test program. Finally, Chapter 9 deals with the generation of test data to execute a statement in the test program.

In Part III, we present an implementation of our consistency approach (called COTTAGE) and experimental results. Chapter 10 describes the COTTAGE system. Chapter 11 then evaluates the experiments conducted so far on the system.

Part IV, specifically Chapter 12, provides our conclusions and directions for future work. Appendix A describes our detailed experimental results. Appendix B shows the C code of the tested benchmarks.

Acknowledgments

Let me first to express my special thanks to the first person, Yves Deville, who has contributed enormously to the realization of this thesis. Yves is a demanding and broad-viewed supervisor. When I worked with him during the first days of my research, I sometimes got angry at his systematic way of coaching. But with the time, I realized that I was wrong with my first impressions. Yves has in fact played an important role not only in my PhD work with his helpful advices, but also in shaping my personality as a researcher. All in all, what I can say is that he is really my great teacher. I will never forget these fruitful years of working with him, especially for his time and responsibility to this work, as well as his financial support. I would also like to thank his wife, Isabelle, for our regular discussions on life subjects, and her constant encouragements to this work.

My warm thanks then go to the members of my jury —Baudouin Le Charlier, Michel Rueher, Charles Pecheur, Bruno Marre— for their interest in my work, their helpful comments and suggestions of improvement to my first thesis draft. I also thank Jean-Didier Legat, for being President of the jury.

I would like to acknowledge the *Département d'Ingénierie Informatique* of the *Université catholique de Louvain (UCL)* and its staffs for having hosted me during my work here, mainly for the material, financial and mental support. Since acknowledging all the people of the department who have had an impact on my work, would be an impossible task for me, and also because of this limited space, I have to be restricted to only a few whom I recall by this writing. Many thanks, for example, to Elie Milgrom (who is certainly the one who supported me to reach the assistant position I hold now. I met him in Hanoi before joining the UCL, and he was the sole person I had the contact with for my assistant candidature), Steve Uhlig (for giving me the Latex style file of this thesis, for our interesting discussions, and especially for the books, I have borrowed from him, which are an invaluable source of inspiration for my personal development at the spiritual level.), etc. Of course, I have also benefited an enormous support from other colleagues, the secretaries, the technical and logistic staffs, to be able to enjoy a warm environment in our department.

I also acknowledge the precious financial support from the Belgian *Fonds National de la Recherche Scientifique (FNRS)* and the SigSoft CAPS Funding program of the *ACM*, which allowed me to participate at some conferences.

I further thank all my friends, including Isabel Cardenal Flores and my vietnamese friends, with whom I have spent unforgettable moments (of football, cycling, jogging, eating, ...).

Finally, I direct my profound gratitude to my parents for their permanent care all over the years. Thanks also to my sister and other relatives of my family for all their support.

Nguyen Tran Sy

Université catholique de Louvain
May 2005

Contents

List of Tables

List of Figures

List of Algorithms

Part I

Test Data Generation and Background

Chapter 1

Test Data Generation

This thesis presents a method and a system of test data generation for structural criteria in software testing. Our approach is based on consistency techniques developed in this work for test data generation.

1.1 Introduction

1.1.1 Context

The development of reliable programs is one of the most important requirements in today's program construction. Several techniques are used in practice; they are generally divided into the following areas: *program proving* and *program verification & validation.*

Program proving involves formally proving that the program meets its specification without a need to execute the program at all. To do this, one needs to obtain a precise specification of the program behavior, including an *input predicate*, an *output predicate*, etc. The predicates here define thus the correct behavior of the program. That is, for all inputs satisfying the input predicate, any results given by the program must satisfy the output predicate. One then follows a formal proof method for verifying the correctness of the program with respect to its input/output predicates. For example, a well-know formal method —the weakest precondition (*wp*) calculus [17]— can be used for such correctness verifications. First, assertions about the program's variables are made at various points in the program. Then, (automated) theorem proving techniques can be exploited to verify these assertions. Note that input/output predicates is not the only way to do program proving. There are also other methods based on invariants, temporal logic, etc.

By its nature, program proving seems to be the most important technique for producing reliable programs. Since proving the correctness of programs is often a complex and tedious task as described above, especially with realistic programs of certain size, one should avoid doing such a verification work manually as much as possible. However, many practical problems —such as the creation of program assertions, the replacement of human interactions in the theorem proving phase, etc.— are still to be solved to make an automatic tool for routine use. Hence, with the current state of the art in program proving, some drawbacks are possibly present. For instance, if the program cannot be proved correct, we are facing either of the following conclusions: (1) an error in the program, (2) a flaw in the assertions, and (3) a limitation in the theorem proving phase by human or machine. Also, a great care should be taken to machine-dependent issues such as overflow, rounding, etc.

Program verification and validation (*V&V*) includes a wide range of techniques, as well as their automated tools, that help in analyzing and evaluating programs. It should be noted that these techniques are designed to develop a more confident program, but they do not necessarily guarantee the correctness of the program. We name here only a few tools that have been developed in the literature, each analyzing some aspects of programs. For example, the Aristotle system [33] implements many program-analysis functionalities such as control-flow analysis (to calculate control dependences in the program), dataflow analysis (to calculate dataflow dependences), etc. The results of a program analysis are often presented in a readable form, facilitating thus the detection of anomalies in the dataflow, for instance. The Effigy system [44] algebraically represents a path's computations by symbolically executing a path. The Select system [7] generates test data and verifies assertions for program paths.

Among such *V&V* techniques is *abstract interpretation* where one tries to obtain a sound and efficient approximation of the possible program executions (*sound* means that at least all possible executions are covered by the approximation). An application is to check if the program possibly produces unexpected errors, or to prove their absence. For example, the ASTRÉE Analyzer [13] —an abstract-interpretation based static analyzer— is able to prove the absence of runtime errors in large embedded control-command safety critical real-time software. As a part of ASTRÉE, [51] proposes a framework for the detection of floating-point runtime errors such as overflow, division by zero, etc. More precisely, this framework seeks to prove that such runtime errors will not occur in any execution of the analyzed program.

Program (software) testing [40] —a branch of *V&V*— involves operation of a program under controlled conditions (both normal and abnormal conditions) and evaluating the results. It is an expensive and difficult task, accounting for up to 50% of the cost of software development [45] and even more in critical systems. The objective of software testing is to detect faults in the program [16], by making things go wrong with test data, and therefore provide more assurance for the quality of the software. If the software testing phase could be automated, the cost of software development would be significantly reduced. A disadvantage of testing is that we usually take sample test data from the input domain for running, and then carefully checking the execution results. Therefore, we are not sure as to the correct execution with inputs not in the sample. Interestingly, testing can be done even when formal specifications are not given, which is often the case in practice. Furthermore, testing on a real environment with actual data seems to be a complementary technique to formal verification, because a formal proof of correctness cannot assure that the program will run as intended on a given machine.

Notice that testing should be done as close to an equivalence partitioning [14] as possible. *Equivalence partitioning* is a technique involving:

- identifying a finite number of equivalence classes from the input domain of a program,
- devising a single test case for each equivalence class.

Here are the underlying assumptions for the reliability of equivalence partitioning:

- A test for each member of an equivalence class represents thus the test for the whole class, and hence an equivalent of exhaustive testing on the input domain can be expected.
- If one test case in a class detects an error, then all the others in the same class will detect the same error.

We now describe some typical steps [56, 29] used in a complete testing of large software.

- Individual procedures are independently tested during *unit testing.* Unit testing is thus carried out at the *intraprocedural* level, namely its application is limited to the procedure's body.
- The interactions (interfaces) between procedures are tested during *integration testing.* Integration testing is done at the *interprocedural* level. For example, we want to exercise certain paths, definition-use associations, . . . , across many procedures.
- The testing of a complete and integrated system prior to delivery is called *system testing.* The aim is to verify that the system meets its specified requirements.
- Program changes are often needed, after the first deployment of the software, to better meet the new requirements of the user. *Regression testing* consists in testing the modified or new parts of the software, as well as ensuring that no errors have been introduced into the remaining unchanged code. This means that the modified or new parts, and other parts affected by the program changes, should be subject to testing. During regression testing, some existing test cases developed during the previous testing can be reused, and new test cases should be devised to adequately test the modified code.

In this work, we are however concerned with unit testing and integration testing to a certain degree. All the following discussions on testing are also under the hypothesis that tested programs (or test programs, for short) are imperative and deterministic. Other types of programs (concurrent, object-oriented, . . .) based on other programming paradigms are out of the scope of this thesis.

1.1.2 Test data generation

As it is generally impossible to test the entire input domain of the program, *testing coverage-criteria* (testing requirements) are used during the selection of sample test data. Testing requirements can be the following: program statements, program paths, definition-use associations (w.r.t. variables), etc. A testing criterion specifies thus a minimal set of testing requirements, which must be covered by a set of test cases executed during the testing process. *Test data generation* is thus a component of software testing, where one tries to generate test cases covering some testing criteria. We illustrate, in Figure 1.1 (inspired from a figure in [29]), the steps of a typical testing process for a testing criteria [29].

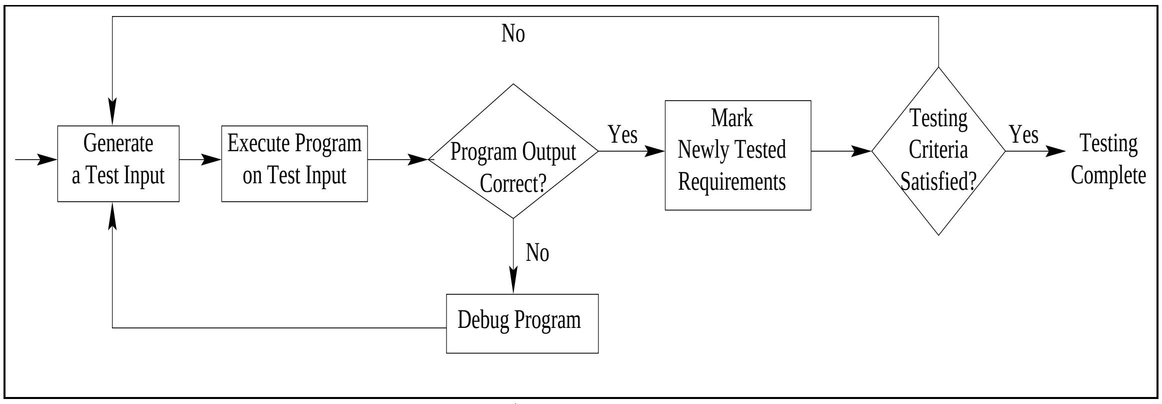

No
Generate a Test Input
Execute Program on Test Input
Program Output Correct?
Yes
No
Mark Newly Tested Requirements
Testing Criteria Satisfied?
Yes
Testing Complete
Debug Program

Figure 1.1: A typical testing process

When the program is run on a generated test input, the trace of its execution is analyzed to provide the set of testing requirements, exercised by the test input. Once the chosen testing criteria is satisfied by the generated test inputs, the testing process with that criteria finishes. After having executed the program on a test input, if the program output differs from the expected output, the program is passed through a debugging process, and then the testing process restarts. Of course, if the software has successfully gone through a greater number of testing criteria, we have more confidence in its reliability.

After an initial testing (where the tester often randomly generates test data), the problem of finding additional test data to satisfy the remaining (not yet covered) testing requirements can be very labor intensive, increasing thus the cost of software testing. Because it is commonly accepted [21, 30, 25] that finding test data exercising certain program elements in complex programs is likely very hard without an automatic test data generator. That is the reason why we propose in this thesis a new approach for test data generation of imperative programs, supported by an automatic test data generation system so as to validate the feasibility of the approach.

1.1.3 Testing Techniques (*structural* versus *functional* testing)

One usually distinguishes *functional testing* (also called specification-based or black-box testing) from *structural testing* (code-based or white-box testing) [14, 12, 25]. Functional testing compares the behavior of a program under test (test program) against a requirements specification, which includes a set of required functions of the intended program. We thus select test data from such required functions. The next step consists in assessing whether all the functions are implemented correctly, by executing the program with the test data. This contrasts with structural testing, which compares the behavior of the test program against the intention of the source code. Therefore, the selection of test data is carried out based on some internal structure (representing the source code) of the program — hence the term structural testing. Of course, the selected test data must be finally executed to check whether the software meets a specification. The fundamental difference between structural testing and functional testing is, however, that the former selects test data from the code while the latter from the specification.

Structural testing and many testing-related notions (e.g. testing criteria) were early introduced in a pioneering work [24]. A mathematical framework for testing [28] was later developed, generalizing [24] by introducing the fault-detecting ability in testing criteria. On the other hand, functional testing with formal specifications was early introduced in [6]. More information on how evolve these two testing techniques can be found in [14, 25].

It is commonly known that the two testing techniques mentioned above, are actually complementary [25], and each of them helps in detecting certain kinds of faults.

On one hand, structural testing may uncover some possible faults in the structure and logic of the program. This bases on the following assumption [12]: faults relate to the control flow of the program, and one can expose faults by varying the control flow. Other assumptions [12, 56] involve unreachable code (dead code), omission of intended functions, typographical errors, etc. A typical example with control-flow related faults is: we often believe that a logical path is not likely to be executed while, in fact, it can be executed. For instance, the instruction "`if (16.0+x==16 && x>0) return 1; else return 0;`" may be believed that only one of its branches (namely "`return 0;`") will be taken for every floating-point number, while we can actually find test data for all of its branches (see our Experiments chapter for the results with this instruction). This means that our wrong assumptions about the control flow and the data flow can lead to design errors that are only uncovered once structural testing starts. Moreover, a program that has been tested with a high structural coverage may not satisfy its specification, due to the omission in the program of some functions intended by the specification. Also, test data generated from structural testing may reveal unwanted functions not concerned by the specification.

On the other hand, functional testing aims to discover faults relating to what the program accomplishes, without regard to how it works internally. In other words, test data generated from the specification would prove useful in testing all intended functions of the program.

Of course, those testing techniques can only expose some plausible faults. They cannot ensure exposing all classes of faults, or showing their absence. That's why other software engineering activities such as code inspection, code verification, etc., are also necessary to guarantee the exactitude of software [56]. By verification, we mean that one must ensure the exactitude of the software with respect to a specific specification by means of formal proofs, as mentioned above.

Testing techniques can also be classified following another dimension [14]: static versus dynamic analysis. *Static analysis* is a testing technique that does not involve the execution of the software with test data, while dynamic analysis requires that the software be executed. See also [14] for a taxonomy (classification) of testing techniques on two dimensions: structural v. functional strategy, and static v. dynamic analysis.

1.1.4 Oracle

Given the test cases generated for some testing criteria, the tester then runs these cases, and compares the execution results with the expected results provided by a specification. If the specification is incomplete, the tester must decide the exactitude of the execution results based on his comprehension of the program. Even in the presence of a complete specification, a work involving running a program with many test cases and evaluation of the results, can be very tedious and error-prone if done without great care. Therefore, the existence of an *oracle* [20, 25] —an automatic procedure to replace the tester in such work— can greatly reduce the effort of testing.

Often, one has to construct an oracle manually from a specification. It is not easy, indeed, to generate an oracle automatically, since one needs to have a complete specification, that in turn should be expressed in some formal form of logic description of the program. Unfortunately, requirements or design specifications sometimes do not exist, or exist in an informal form, or are often incomplete.

Due to practical reasons, it is difficult to obtain a precise oracle, which can determine the exact execution result for a test case. An example is to work with a floating-point arithmetic on a machine, which is usually not sound. Because the result of an operation is usually rounded to make it a floating-point number. And the rounding can be done upwards or downwards, depending on the rounding mode being used. In such case, the most precise expectation, that can be given, is a range of possible values.

1.1.5 Structural Testing

We now focus our attention to structural testing and its criteria. In the testing literature, structural testing techniques usually fall into the following three types [29]: *control-flow based* testing, *data-flow based* testing, and *mutation* (or *fault-based*) testing. Note that the differences between these testing types are only in the coverage measurement, but not in the testing itself. This means that each coverage criterion only defines a set of tested elements to be covered by the generated test data. But the process of running test data and verifying test results is the same for any set of generated test data.

Control-flow testing Control flow criteria consider the elements of the control flow graph of the program —such as nodes, edges, paths, etc.— for coverage. (The control flow graph here captures the control flow of the program.) Structural testing with control-flow criteria will be discussed more in detail in the next subsection.

Data-flow testing Data flow criteria deal with the data-flow dependencies in the program, such as the definition-use associations present in the program. Each occurrence of a variable on the left hand side of an assignment is called a *definition* of the variable, while each subsequent reference of the variable is called a *use*. Exercising a definition-use association can be viewed as traversing a sub-path from a variable assignment to a subsequent reference of the variable. For more general discussions on data flow criteria and their subsumption relationships, see [29].

Mutation testing Mutation (fault-based) testing [14] begins by creating a number of almost identical programs from the original program. These programs are called *mutants*, each is obtained by inserting a (small) change into the original program. The changes here refer to the most frequent faults that may exist. Each mutant and the original program are executed with the same set of test data. The output from each mutant is compared with the output from the original program. If the outputs are different, the mutant is said to be "killed", as the test data has discovered a difference between the programs; otherwise, the mutant is "live". Mutation testing relies on the assumption that if test data discovers the change made to produce the mutant, then the test data will discover more major faults in the program. Hence, if a high proportion of the mutants are killed, then more confidence is gained on the test data, i.e. the program has been well tested. In contrary, a high proportion of live mutants indicates a poor set of test data. In that case, more test data must be generated until the number of live mutants becomes small.

One issue that should be carefully considered in mutation testing is the quality of mutants, or equivalently the quality of faults and fault modeling. Intuitively, a mutant m_1 is better than a mutant m_2, if test data may kill m_2, but not m_1. In other words, the test data may discover the change (fault) introduced in m_2, but not the fault introduced in m_1. Hence, generating test data to kill a set of better mutants will give us more assurance on the testing process, because it is hoped that more types of faults can be discovered by the test data.

[58] proposes a class of fault-based testing criteria. They are however limited to the very rudimentary faults such as constant reference faults, variable reference faults, variable definition faults, operational operator faults, etc. A more general framework on fault modeling and fault seeding, using the program dependence graph, can be found in [34]. In this framework, the program is represented by a program dependence graph(PDG) that explicitly represents both data and control dependencies in a program [22]. Therefore, changes (faults) made to the original program can be viewed as changes to the PDG that can be:

1. changes within the PDG nodes,
2. changes resulting from structural transformations on the original PDG.

Note that all the faults handled by [58] are at the node level, and thus being included in the first class of changes above.

Discussion

It should be noticed that the above three types of structural testing criteria are in general orthogonal in their capacities of fault detection. For example, although control flow criteria is useful in detecting faults related to the control flow [12], it is generally insensitive to other fault classes related to the data flow [67]. Likewise, test cases generated to satisfy a fault-based testing criteria [58] could fail to uncover faults related to the control flow. Therefore, to improve the fault-detecting power of the generated test cases, one should consider combining testing criteria.

1.1.6 Structural Test Data Generation with Control-flow Criteria

In this thesis, we are concerned with structural testing (or more exactly, structural test data generation) with control flow criteria. The control flow of a program is usually represented by a *control flow graph* (CFG), where the nodes are either a decision node or a block of instructions without decision statements, while the edges represent the possible control flow between nodes. To adequately test the program at the structural level, one must consider structural elements (nodes, branches, or paths) of the CFG for coverage. For example, *statement coverage* requires developing test cases to exercise a given set of program statements (a set of nodes). The problem is thus to find, for each program statement, a test case (program input) on which this statement is executed. *Branch coverage* is the dual version where an input must be found, such that the execution traverses a specified edge of the control flow graph associated to the program. And *path coverage* requires test cases to execute certain paths (from the start to some statement). A large variety of other coverage measures, along with their strengths and weaknesses, are given in [12]. Structural testing with control flow criteria thus includes the following phases:

1. the choice of a criteria (statement, branch, path, ...),
2. the identification of a set of nodes, branches or paths, and
3. the generation of test data for each element of this set.

For the second phase, taking the statement-coverage criteria (*all-the-statements*) as an example, we simply choose all the nodes of the control flow graph. Of course, we can take an incremental approach to select elements of the control flow graph, by identifying all the elements already covered on execution of a test case. By this way, we can importantly reduce our effort to find test data covering a criteria.

The notion of a minimally-thorough test has been a subject of discussions over the years. Some examples [14] of what constitutes a minimally-thorough test of a program are given hereafter:

- All statements in the program should be executed at least once.
- All branches in the program should be executed at least once.

Note that these testing considerations were also proposed in the *RTCA/DO-178B* norm for avionic systems [59]. Therefore, at least, one has to achieve one of these to guarantee a necessary test on a program. The best test is, of course, an exhaustive test, where all paths of the program are tested. However, this is impractical with realistic programs for the following reasons: (1) the number of paths can be infinite, if loops are involved in the program; (2) the number of infeasible paths (i.e. not executed by any test data) can be very big, compared with feasible paths. It was reported in [14] that among a sample of programs, of the 1000 shortest paths, only 18 were feasible. Hence, a big effort could be wasted to find test data for infeasible paths. To reduce the above obstacles with the all-paths criteria, we can choose, for instance, only a set of basic paths through the program [56] for test data generation. The idea of constructing a set of basic paths is as follows: A path going through the program is first constructed; Each new path is constructed to traverse branches of the program not yet covered by the current set of paths until all the branches are considered.

The automation of the last phase (automated test data generation) is a vital challenge in software testing for the following reasons. First, generating test data for certain program elements (e.g. profoundly-nested nodes) can be very difficult, although we need only one solution (a test case exercising the selected program element). Of course, finding all solutions is not required by a testing criteria. Second, some program elements (nodes, edges or paths) of the control flow graph may not be executed by any test case. These program elements are said to be non-executable or infeasible. Even with a program without faults, the presence of such non-executable elements is frequent. Furthermore, determining whether a node, an edge or a path is executable, is undecidable in the general case [68] (reduced to the halting problem in computability theory). Therefore, it is likely that we try in vain generating test data for some non-executable program elements.

Although the problem of undecidability in automated (automatic) test data generation, a lot of approaches have been proposed in the testing literature; they were also supported by various test data generators. A test data generator is a software system assisting the tester in the generation of test data. In the next, test data generation for a node will be referred to as statement coverage, for a branch as branch coverage, for a path as path coverage.

For path coverage, the main approaches are: *symbolic evaluation* [10, 44, 11], *program execution based (dynamic)* approaches [46, 30]. A mixed approach integrating ideas from symbolic evaluation and dynamic test data generation, has also been proposed in [53].

For statement coverage, the main approaches go into the following classes: *random test data generation* [19, 67], *program execution based (dynamic)* approaches [21, 47, 32, 54], *constraint-programming* (and more particularly *consistency-based*) approaches [26]. The method in [26] is the nearest method with ours; it is based on *Constraint Logic Programming* (CLP) techniques. Constraint logic programming [42] is a framework integrating two declarative paradigms: constraint solving and logic programming. The test data generation problem for a given statement is translated into constraints, solved by an instance of the CLP scheme: $CLP(\mathcal{FD})$. $\mathcal{FD}$ here stands for Finite Domains, which means that the variables treated by $CLP(\mathcal{FD})$ can only take values in finite domains. Note that the solving of such CLP instance relies on consistency techniques —which are designed to reduce the search space (reducing the domains of the variables)— and a search process. These methods will be presented in detail in Chapter 2 of this thesis.

1.2 Results and Contributions

Among the difficulties in the generation of test data is the presence in the program of arrays, procedure calls, pointers, unstructured control statements (such as goto, break), and floating-point variables. In this work, we propose a consistency-based approach [63, 64, 62] —referred to as the *consistency approach*— for test data generation of imperative programs containing *integer*, *boolean* and *float* variables, arrays, and procedure calls. Path and statement coverage are both handled. As a branch is dual to a statement in the control flow graph, all the results with statement coverage can easily be extended to branch coverage.

Path coverage is the basic bloc of our approach. It is a constraint solving approach based on a consistency notion, e-box consistency, generalizing box-consistency [36] to integer, boolean, and float variables. For statement coverage, paths reaching the specified statement are dynamically constructed using consistency techniques, and the path coverage method is applied on these paths to find suitable test input. We also propose a dynamic approach to statement coverage, by combining random test data generation, program execution and our path coverage method — but no implementation has been realized for this approach.

Our method for path coverage includes the following steps:

1. A path constraint is derived from the specified path of the ICFG. Such a constraint involves integer, boolean and float variables, as well as operations with arrays.
2. The path constraint is solved by an interval-based constraint solving algorithm, generating (very) small boxes containing float solutions of the path constraint. These small boxes are called interval solutions. Note that in interval programming, each variable is associated with an interval representing its domain. And by *box*, we mean the domains of the variables of the path constraint (intervals).
3. A test case is finally extracted from the interval solutions.

A prototype system —called *COTTAGE* (COnsistency Test daTA GEnerator), and consisting of 13,000 Java lines of code— implements our approach for programs written in (a subset of) the C language. In the current implementation, we focus on C programs with integer and float variables, arrays, function calls, and a restricted class of one-dimensional pointers (to simulate by-reference parameters); but without general pointers and/or dynamic data structures.

Contributions

The main contribution of our work is a new approach (based on consistency techniques) to the generation of test data for numeric programs (programs with integer, boolean and float variables) with procedure calls and arrays. This approach handles *path* and *statement* coverage criteria. Specific technical contributions include the following.

- A new system of test data generation, namely COTTAGE as mentioned above. For each generated test data, the system also automatically generates an instrumented *C* program, allowing the user to verify the correctness of the test data. Experimental results, including complex numerical programs from [55], demonstrate the feasibility of the method and the efficiency of the COTTAGE system, as well as its versatility and flexibility to different classes of problems (integer and/or float variables; arrays, procedures, path coverage, statement coverage).
- Inside the system is a constraint solver suitable for test data generation (e.g. dealing with integer, boolean and float variables).

- An extended framework on interval logic so as to handle interval constraints involving at the same time, integer, float and boolean variables, as well as the logical operators such as *AND*, *OR*, *NOT*.

Remark

Floating-point data are only partially handled in COTTAGE due to the following reasons:

- The generated test data are only potential float solutions in *C* (See Definition 3.4 for *float solutions*). Because they have been calculated with the floating-point system of *Java*. And therefore they must be validated in the floating-point system of the programming language used by the program under test — the *C* language in our case.
- In *Java*, the floating-point system uses only one rounding mode *nearest* —see Section 3.5.2 for an overview of floating-point arithmetic— while in *C*, all the four rounding modes as specified by floating-point standards can be used, and the rounding modes other than *nearest* cannot be specified in our Java solver.

The other known method [25, 26], related to our work and also based on consistency, is limited to integer variables, and does not handle interprocedural control dependence. A constraint solver over float numbers has been proposed [50], where it was shown how such a solver could be used for test data generation. The solver is however limited to float variables, and no implementation (using the solver for test data generation) is provided.

Compared with other approaches handling integer and float variables in the literature (e.g. [31, 54]), our approach can be seen as an alternative or as a complement. Our consistency method could be combined with dynamic approaches when searching a test data exercising a specified statement of the program.

1.3 Conclusion

In the first section of this chapter, we first described a general process of software testing. And the role of test data generation was then located in this testing process. We made a distinction between two classes of testing techniques: structural testing versus functional testing. Structural testing deals with some internal structure of the test program, where the source code is subject to an analysis, while functional testing deals with how the program works w.r.t. some specification without caring how the program was coded. These two testing classes are known, however, to be complementary in software testing, as well as in their ability to detect faults. While the focus of our work is on structural testing with control-flow criteria, we also discussed about structural testing with data-flow and fault-based criteria. The problem of test data generation under the testing criteria such as path coverage, statement coverage and branch coverage, was informally defined at the end of the first section. Path coverage has the purpose of finding test data exercising a path, while statement (branch) coverage aims to find test data traversing a node (branch) of the control flow graph of the program. Finally, the results and contributions of our work were given in Section 1.2.

Chapter 2

Related work

This chapter presents the main existing methods of test data generation. It is divided into two parts: related work on path coverage, and related work on statement (branch) coverage.

2.1 Related Work on Path Coverage

Path coverage can be viewed as searching for test data exercising a path of the control flow graph. For path coverage, one finds the following categories.

2.1.1 Symbolic evaluation

Symbolic evaluation (or *symbolic execution*) [44, 10, 7] consists in replacing input variables by symbolic values, and then symbolically evaluating the statements along a path. The output from a symbolic execution includes algebraic expressions over these symbolic values for the output variables, as well as a set of constraints representing the conditions on the path, which must be satisfied for the path to be executed. For short, these constraints are collectively called the *path constraint*. The symbolic output here describes thus a sub-domain of test cases (of the input domain), each executing the specified path. An interesting book on the literature of symbolic execution can be found in [14].

Since the aim is to generate test data, the path constraint is solved to obtain a test case(s). If the path constraint is shown to have no solution, the selected path is thus infeasible. We will summarize hereafter some systems based on this approach, as well as their underlying techniques used in solving the path constraint. Note that [14] makes a detailed comparison of these systems in terms of functionalities; a light comparison of such systems in terms of techniques for solving the path constraint, is also given in [25].

The EFFIGY system Symbolic evaluation was early introduced in [44]. The idea was then developed into a system (called EFFIGY) for testing and debugging of programs written in a simple *PL/I* style programming language. It was shown how such a system can be used not only for program testing, but also for program verification (e.g. user-supplied assertions can be used to generate verification conditions, which are then compared with the results of symbolic evaluation). Because the expressions handled by the system are mainly restricted to integer polynomials over symbolic values, the simplification of expressions —based on algebraic manipulations— can be done easily. The operation of the system was illustrated on some examples, showing how symbolic results were obtained from symbolic inputs. Therefore, using the system to generate test data, the path constraint must be manually solved. It was also discussed that the EFFIGY system could be used for a more general class of expressions. However, it can be an ambitious burden on the expression-manipulation component of the system.

The SELECT system The SELECT system [7] generates test data and creates a symbolic representation of a path's computations, for programs written in a subset of *Lisp*. SELECT is also a pioneering symbolic execution system as EFFIGY. It provides similar facilities to EFFIGY —simplified symbolic expressions for the output variables on a path, statements of correctness for user-supplied assertions, etc.— and, in addition, automatically generates test data. A minor inconvenience of the system is that the path constraint and output variables were represented, after a symbolic execution, as a Lisp list, and hence making difficulty for human reading.

For solving the path constraint, some algorithms have been experimented by the system. The first algorithm (GOMORY 1963) handles systems of linear equalities and inequalities among integer variables. This algorithm uses an integer linear programming to optimize an objective function, with the path constraint serving as constraints.

Since the GOMORY algorithm is limited to integer path constraints, a mixed integer linear programming algorithm (BENDERS 1962) —dealing also with real variables— was then used. However, the BENDERS algorithm is still limited to linear constraints.

The last and also the most promising algorithm, used by SELECT, seems to be a *conjugate gradient* algorithm (also called *hill-climbing* algorithm) that seeks to minimize a potential function constructed from the equalities/inequalities. Because all kinds of computable functional combinations are allowed between program variables. Different ways can be envisioned to construct the potential function from the equalities/inequalities, provided that the desired minimum potential is attained within the path constraint subregion. Note that we can view a path constraint as defining a subregion of the whole input domain, such that any point in the subregion will lead to execution of the corresponding path of the program. It should be emphasized that user interaction is usually required to come up with problems such as: sample data points, decisions involving when to terminate the hill-climbing algorithm (otherwise, it may loop for ever), etc.

One advantage of SELECT is that all paths of the program are automatically explored to generate test data. To avoid a possible exploration of an infinity of paths in the presence of loops , no loop is traversed more than S times, where S is a looping factor supplied by the user. During automatic construction of paths in depth-first, any prefix of a path is always checked by the system for its consistency (i.e. having solutions). If the subpath is proved inconsistent, all of its extensions are thus abandoned immediately. Note that when a decision node is reached, and a branch is taken to continue the path extension, the alternative branch is also examined for its consistency. If so, a backtracking point is established for future analysis of the alternative branch and its extensions.

The ATTEST system A symbolic test data generator for programs written in *ANSI Fortran* —called ATTEST— was developed in [10], where the solving of path constraints is based on a mixed linear programming algorithm for integer and real variables due to GLOVER. Of course, linear programming algorithm can solve only systems of linear constraints, and thereby test data generation is confined to paths that can be described by a set of linear constraints. The general form of a linear programming problem is

$$MAX\ 0(X),$$

subject to

$$AX \leq B \quad \text{and} \quad X \geq 0,$$

where 0 is a linear function called the objective function, X is a N-vector of input variables, B is a M-vector of constants, and A is a $M \times N$ matrix with $N > M$. Hence, the constraints must be transformed into the form $AX \leq B$.

The constraints generated during the symbolic execution can be in a complicated form. Therefore, they are first simplified before any attempt to solve them. The simplification phase here is carried out by *Altran*, a language designed for algebraic manipulations.

Little error checking was also integrated into the system, by modeling some common programming errors by constraints. Such constraints are then added into the path constraint for solving. Therefore, any solution to the augmented path constraint may be an indication of such errors in the program. For example, assume that the allowable indices of an array are between 1 and 100. When an array element $A[I]$ is referenced on the path, the two constraints $I > 100$ and $I < 1$ are created. If either of these constraints is consistent with the path constraint, the program can contain out-of-bounds errors.

The CASEGEN system The CASEGEN system [57] generates test data for *Fortran* programs. To solve a path constraint consisting of nonlinear equalities and inequalities, a procedure —based on systematic trial and error, and random number generation— was developed as follows.
(1) All input variables are arranged in a sequence denoted as $v_1, v_2, \ldots, v_n$.
(2) For each variable v_i, the set s_i of component constraints of the path constraint is constructed such that each constraint of s_i contains variable v_i and variables from the set $\{v_1, \ldots, v_{i-1}\}$ only.
(3) Assuming values for $v_1, \ldots, v_{i-1}$, satisfying all constraints in $s_1, \ldots, s_{i-1}$, have been generated, the procedure assigns a value to v_i according to the forms of constraints in s_i as follows.

- If there is an equality relation in s_i. This will be solved to find a value for v_i. The value for v_i is then propagated in all other constraints of s_i. If an inconsistency is detected, backtracking is done to generate a different value for v_{i-1}. Otherwise, Step (3) is continued to find a value for v_{i+1}; or the procedure terminates when $i = n$, and we obtain thus a test case.
- If there are no equalities in s_i, a random value is generated for v_i according to its type. If all constraints of s_i are satisfied, Step (3) is repeated for v_{i+1}. Otherwise, random number generation is still carried out a fixed number of times before a backtracking to variable v_{i-1} is taken.

Note that it is not precised in the paper which kinds of equalities are actually solved by the procedure. Even with such a precision, the procedure is still an incomplete heuristic.

Other symbolic execution systems The above systems are among pioneering works on symbolic execution, and hence worth a detailed discussion. We will quickly discuss below the main features of some other symbolic execution systems.

The SMOTL system for programs in *Smod* (a *Cobol*-like language) proposes a domain reduction method for solving the system of inequalities on integers (the path constraint). The system of inequalities is repeatedly examined: for each inequality and for each variable of the inequality, the domain of the variable is corrected (reduced). Only the following example is given in the paper to illustrate the idea of domain reduction: if the inequality is $x < y$, and the corresponding domains for integer variables x and y are $[a, b]$ and $[c, d]$, then after the reduction, the domains become $[a, min(b, d-1)]$ and $[max(a+1, c), d]$. The domain reduction process terminates when a fixpoint is reached (no changes in any domain) or some domain becomes empty. The latter case indicates failure in finding a solution. In the former case, two subcases are possible. First, if the system of inequalities contains no arithmetic expressions, then the variables will be assigned the lower bounds of the corresponding domains. Second, if the system of inequalities contains arithmetic expressions, a restricted search (not precised in the paper) is conducted to find a solution from the domains, because an exhaustive search may consume too much time. Therefore, such a restricted search does not guarantee to find a solution, even one actually exists. An advantage with SMOTL is, however, an automatic strategy for combining paths in order to reduce the number of paths covering all branches in the program before such paths are exploited to generate test data.

The SYM-BOL system for programs written in a subset of *Cobol* is described in [14]. By assuming that the path constraint is linear, a linear programming routine (using some sort of linear optimization) is employed to assess path feasibility. A strategy for automatic path selection is integrated: path selection and symbolic execution are undertaken in parallel, allowing thus the intermediate results from the symbolic execution to help in path selection, and hence reducing the risk of selecting infeasible paths.

Other systems making use of symbolic execution —such as DISSECT, EL1, IPS, SADAT, IVTS, UNISEX, the *Fortran* testbed— will not be presented here. See [14] for references of such systems.

Weaknesses of symbolic evaluation

Symbolic evaluation systems are usually limited in handling arrays, indeterminate loops, and procedure calls. Powerful tools may be also required to deal with the simplification and manipulation of complicated algebraic expressions. Moreover, they cannot take into account many machine-dependent issues such as overflow, rounding, division by zero, etc.

Arrays Here is an example of a system having difficulty with arrays, the ATTEST system, where array references that depend on input variables were not handled. Consider, for instance, the following segment of code with an array reference $A[i]$:

```
read(i;
A[1] = 5;
A[2] = 1;
if (A[i] > 3) ...
```

Since $A[i]$ is dependent on input variable i, it is then impossible to determine which array element is referenced. Of course, one could make an enumeration on the possible values of the index — e.g. as proposed in EFFIGY, whenever an ambiguous array reference is encountered, one proceeds with a branch point of N parallel computations, where N is the size of the array. However, it may become too complex in the general case (e.g. many such array references occur in the program).

In CASEGEN, ambiguous array references are retained during symbolic execution, and are only resolved when array indices are given a value during test data generation. To do this, new instances of the array are created whenever array references produce uncertainty. For example, if the current instance is k, then after the statement

$A[M] := P$;

we obtain a new instance $k + 1$ of the array in which

$A_{k+1}[i] = A_k[i]$ for all $i \neq M$

$A_{k+1}[M] = P$.

All such instances of the array allow thus ambiguities to be resolved when test data are generated.

Note that a solution to resolve ambiguous array references during symbolic execution actually exists. We will describe that solution later, just after our discussion now on weaknesses of symbolic execution.

Indeterminate loops Symbolic execution has difficulty in proceeding beyond a loop, where the number of iterations depends on the values of input variables. A rather restrictive strategy [14] to treat such a loop is to create three paths: one that contains zero iteration of the loop; a second with one iteration; a third with two iterations. Another approach (e.g. [57]) commonly adopted by many systems is to symbolically execute the loop k times, where k may be specified by the user. In all cases, constraints generated may be unsatisfiable.

Procedure calls Symbolic systems (e.g. the SELECT system [7]) generally have limited capabilities to handle procedure calls. For instance, procedure calls are typically handled in a macro-expansion manner: substituting the code of the called procedures into the main program. This forecloses any possibility of calls to user-created functions (that are compiled independently from the program under analysis) or built-in functions (e.g. the *sin* function), where the code is normally not available. Of course, as suggested in [7], this situation may be overcome by specifying each subprocedure by input/output assertions. Then, whenever a subprocedure is called, it is simply replaced by such input/output assertions. This way ensures not only program modularity, but an efficient way into the test data generation. However, a main issue in this direction is how to choose a suitable specification language for the assertions (sufficiently strong to express the intent of the tester).

Simplification of symbolic expressions The simplification of symbolic expressions can cause a mismatch between symbolic execution and actual execution on a machine. Consider, for example, the following constraint

$$c(x) \triangleq \frac{x}{3} + \frac{x}{3} + \frac{x}{3} = x$$

By a simplification, $\frac{x}{3} + \frac{x}{3} + \frac{x}{3}$ can be simplified into x, thereby the constraint is always *true*. But execution of $c(1)$ on a computer can be $false$ due to rounding and truncation of intermediate results. Hence, this can cause execution of an unintended branch in the program.

An issue not less important is the simplification and manipulation of complex algebraic expressions, because the simplification relies mainly on rewriting of intermediate expressions. And although this can be done automatically, the time and space complexity may become excessive with realistic programs.

A solution for arrays in symbolic evaluation

A solution for dealing with arrays in symbolic evaluation was proposed in the SYMBAD system for *Ada* programs [11]. In this method, each variable is associated with a set of pairs of the form $< val, val\_constraint >$ during symbolic execution, where val is a symbolic expression and $val\_constraint$ is a boolean stating under which conditions the variable has value val. Therefore,

- a simple variable X is associated with a value-set $\{\langle\beta_1, Q_1\rangle, \ldots, \langle\beta_n, Q_n\rangle\}$, i.e. X has value β_i iff Q_i holds;
- an (one-dimensional) array variable A is associated with a value-set $\{\langle\alpha_1, P_1(i)\rangle, \ldots, \langle\alpha_m, P_m(i)\rangle\}$, where i represents the formal index of the array. This means that the j-th array element has value α_i iff $P_i(j)$ holds.

Note that the elements of a value-set $\{\langle\beta_1, Q_1\rangle, \ldots, \langle\beta_n, Q_n\rangle\}$ are mutually exclusive, satisfying the following:

$$\begin{array}{l} \forall i, j \in (1 \ldots n) \;:\; i \neq j \implies Q_i \wedge Q_j = false \\ Q_i \vee \ldots \vee Q_n = true \end{array}$$

This guarantees that in any state of a symbolic execution, each variable has exactly one value, as expected by the semantics of program execution.

This approach to symbolic evaluation thus improves upon other symbolic methods by the fact that during symbolic execution, each variable is associated with a set of conditional symbolic values rather than just one symbolic value. We now outline the idea behind the approach.

During symbolic execution, the symbolic state at any moment is the pair $\langle State, PC\rangle$, where $State = \{(X, \{\langle\beta_1, Q_1\rangle, \ldots, \langle\beta_n, Q_n\rangle\}), (A, \{\langle\alpha_1, P_1(i)\rangle, \ldots, \langle\alpha_m, P_m(i)\rangle\}), \ldots\}$ describes the (variable,value) bindings, and PC represents the path constraint. First, the evaluation of an expression Exp with the current $State$ of the variables —denoted by $\texttt{eval}(Exp, State)$— is realized as follows.

- If X is a simple variable, then $\texttt{eval}(X, State) = \{\langle\beta_1, Q_1\rangle, \ldots, \langle\beta_n, Q_n\rangle\}$
- If A is an array variable, and X is a simple variable, then $\texttt{eval}(A[X], State) = \{\langle\alpha_1, P_1(\beta_1) \wedge Q_1 \vee \ldots \vee P_1(\beta_n) \wedge Q_n\rangle, \ldots, \langle\alpha_m, P_m(\beta_1) \wedge Q_1 \vee \ldots \vee P_m(\beta_n) \wedge Q_n\rangle\}$
- For an (boolean) expression $Exp(V_1, \ldots, V_n)$, assuming the value-sets of the V_i (either simple or array) are:

$$\begin{aligned}&\mathcal{V}_1 : \{\langle\alpha_{11}, P_{11}\rangle, \ldots, \langle\alpha_{1m}, P_{1m}\rangle\}\\ &\ldots\\ &\mathcal{V}_n : \{\langle\alpha_{n1}, P_{n1}\rangle, \ldots, \langle\alpha_{nr}, P_{nr}\rangle\},\end{aligned}$$

  $\texttt{eval}(Exp(V_1, \ldots, V_n), State) =$
  $\{\langle Exp(\alpha_{11}, \ldots, \alpha_{n1}), P_{11} \wedge \ldots \wedge P_{n1}\rangle, \ldots, \langle Exp(\alpha_{1m}, \ldots, \alpha_{nr}), P_{1m} \wedge \ldots \wedge P_{nr}\rangle\}$.

Note that the cardinality of the value-set associated with an expression equals the product of the cardinalities of the value-sets (of the variables involved in the expression): $|\texttt{eval}(Exp(V_1, \ldots, V_n), State)| = |\mathcal{V}_1| \times \ldots \times |\mathcal{V}_n|$.
However, since some *val_constraint*'s of the resulting pairs may be false, the corresponding values are thus infeasible, and hence such pairs can be removed from the resulting value-set.

Given a statement S and a symbolic state $\langle State, PC\rangle$, symbolic execution of S from the symbolic state —denoted by $\texttt{exec}(S, \langle State, PC\rangle)$— returns the new symbolic state $\langle State', PC'\rangle$.
(1) The initial symbolic state at the beginning of a symbolic execution is:
$State = \{\ (X, \{\langle undef, true\rangle\}),\ (A, \{\langle undef, true\rangle\}), \ldots\}$, and $PC = true$.
That is, simple variable X and all array elements of A are undefined.

(2) For an assignment to a simple variable $X := Exp(\ldots)$ —assuming $State$ is $\{(X, \mathcal{X}),\ (A, \mathcal{A}), \ldots\}$— $\texttt{exec}(X := Exp(\ldots),\ \langle State, PC\rangle) = \langle State', PC\rangle$, where $State'$ is $\{(\ X, \texttt{eval}(Exp(\ldots), State)\ ),\ (A, \mathcal{A}), \ldots\}$.

(3) For an assignment to an array element $A[X] := Exp(\ldots)$, assuming $State$ is $\{(X, \mathcal{X}), (A, \mathcal{A}), \ldots\}$, $\mathcal{X} = \{\langle\beta_1, Q_1\rangle, \ldots, \langle\beta_n, Q_n\rangle\}$, $\mathcal{A} = \{\langle\alpha_1, P_1(i)\rangle, \ldots, \langle\alpha_m, P_m(i)\rangle\}$, and $\texttt{eval}(Exp(\ldots), State) = \{\langle\gamma_1, T_1\rangle, \ldots, \langle\gamma_s, T_s\rangle\}$,

$$\texttt{exec}(A[X] := Exp(\ldots),\ \langle State, PC\rangle) = \langle State', PC\rangle$$

where $State'$ is $\{(X, \mathcal{X}),\ (A, \mathcal{A}' \oplus \mathcal{E}'), \ldots\}$,
$\mathcal{A}' = \{\langle\alpha_1, P_1(i) \wedge i \neq \beta_1 \wedge \ldots \wedge i \neq \beta_n\rangle, \ldots, \langle\alpha_m, P_m(i) \wedge i \neq \beta_1 \wedge \ldots \wedge i \neq \beta_n\rangle\}$,
$\mathcal{E}' = \{\langle\gamma_1, T_1 \wedge (i = \beta_1 \vee \ldots \vee i = \beta_n)\rangle, \ldots, \langle\gamma_s, T_s \wedge (i = \beta_1 \vee \ldots \vee i = \beta_n)\rangle\}$,
and the operator $\oplus$ is defined as follows, assuming $\mathcal{A} = \{\langle\alpha_1, P_1\rangle, \ldots, \langle\alpha_m, P_m\rangle\}$,

$\mathcal{B} = \{\langle\beta_1, Q_1\rangle, \ldots, \langle\beta_n, Q_n\rangle\}$,

$$\begin{aligned}
&\mathcal{A} \oplus \{\} = \mathcal{A}\\
&\mathcal{A} \oplus (\mathcal{B} \bigcup \{\langle\beta_{n+1}, Q_{n+1}\rangle\}) =\\
&\quad\begin{cases}
\{\langle\alpha_1, P_1\rangle, \ldots, \langle\alpha_j, P_j \vee Q_{n+1}\rangle, \ldots, \langle\alpha_m, P_m\rangle\} \oplus \mathcal{B} & \text{if } \exists j \in [1..m] \mid \alpha_j = \beta_{n+1},\\
\{\langle\alpha_1, P_1\rangle, \ldots, \langle\alpha_m, P_m\rangle, \langle\beta_{n+1}, Q_{n+1}\rangle\} \oplus \mathcal{B} & \text{otherwise.}
\end{cases}
\end{aligned}$$

Note that $\mathcal{A}'$ contains the pairs associated with the array elements that are not affected by the assignment while $\mathcal{E}'$ contains the new pairs created by the assignment.

(4) For a sequence of statements $S_1; S_2; \ldots; S_n$,
$\texttt{exec}(S_1; S_2; \ldots; S_n, \langle \mathit{State}, \mathit{PC}\rangle) = \texttt{exec}(S_2; \ldots; S_n, \texttt{exec}(S_1, \langle \mathit{State}, \mathit{PC}\rangle))$

(5) When a decision point (conditional or loop) is met, the path constraint PC is used by a theorem prover to check whether the encountered condition is true, false, or undetermined. If the condition is true or false, symbolic execution continues down the appropriate branch. When undetermined, one of the branches is selected by the user, and the corresponding condition is added to the path constraint. Hereafter, only the treatment for conditional statements is given, because treating loop statements follows the same principle. Consider a conditional statement: `if` C `then` S_1 `else` S_2. Assuming $\texttt{eval}(C, \mathit{State}) = \{\langle\gamma_1, Q_1\rangle, \ldots, \langle\gamma_n, Q_n\rangle\}$, then
$\mathit{PC} \Rightarrow \texttt{eval}(C, \mathit{State}) \;=\; \mathit{PC} \Rightarrow \bigvee_{1\le i\le n} \gamma_i \wedge Q_i$
because the Q_i are mutually exclusive. In the same way,
$\mathit{PC} \Rightarrow \texttt{eval}(\neg C, \mathit{State}) \;=\; \mathit{PC} \Rightarrow \bigvee_{1\le i\le n} \neg\gamma_i \wedge Q_i$
Symbolic execution for the conditional statement is formally defined as:
$\texttt{exec}(\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2,\ \langle \mathit{State}, \mathit{PC}\rangle) =$

$$\begin{cases}
\texttt{exec}(S_1,\ \langle \mathit{State}, \mathit{PC}\rangle) & \text{if } \mathit{PC} \Rightarrow \bigvee_{1\le i\le n} \gamma_i \wedge Q_i\\
\texttt{exec}(S_2,\ \langle \mathit{State}, \mathit{PC}\rangle) & \text{if } \mathit{PC} \Rightarrow \bigvee_{1\le i\le n} \neg\gamma_i \wedge Q_i\\
\texttt{exec}(S_1,\ \langle \mathit{State}, \mathit{PC} \wedge \bigvee_{1\le i\le n} \gamma_i \wedge Q_i\rangle) \text{ or} & \\
\texttt{exec}(S_2,\ \langle \mathit{State}, \mathit{PC} \wedge \bigvee_{1\le i\le n} \neg\gamma_i \wedge Q_i\rangle) & \text{otherwise}
\end{cases}$$

Experiments with some simple programs were presented. It is not clear as to whether the approach has been experimented on more complex programs, since the approach may create an explosion in the number of pairs associated with each variable. Furthermore, a major weakness of the related SYMBAD system is that test cases are not generated.

2.1.2 Program execution based approaches

Program execution based (or *dynamic*) approaches start by executing the program with an arbitrary test input(s). This input is then iteratively refined, by execution of the program, to obtain a final input(s) executing the path. The refinement is done by applying *function minimization search algorithms* [46], an *iterative relaxation method* [30], etc. We will make an overview of some of these approaches below.

Function-minimization-algorithms method In [46], the refinement of test input is carried out at a branch of the path, where its corresponding condition (or

predicate) evaluates to an undesired value on the current test input $\mathbf{x}^0$. The branch condition is supposed to be of the following form: E_1 *op* E_2, where E_1 and E_2 are arithmetic expressions, and *op* is one of $\{<, \leq, >, \geq, =, \neq\}$. The condition E_1 *op* E_2 is first transformed into an equivalent condition of the form, $F(\mathbf{x})$ *rel* 0, where *rel* is one of $\{<, \leq, =\}$, such that E_1 *op* E_2 is satisfied iff $F(\mathbf{x})$ is negative or zero (depending on *rel*). For example,
$E_1 > E_2$ is transformed into $E_2 - E_1 < 0$,
$E_1 \leq E_2$ into $E_1 - E_2 \leq 0$,
$E_1 = E_2$ into $abs(E_1 - E_2) = 0$,
$E_1 \neq E_2$ into $-abs(E_1 - E_2) < 0$.
Our refinement problem is therefore reduced to a minimization problem with constraints, where $F(\mathbf{x})$ is minimized using an *alternating-variable method* for constrained minimization, stopping when $F(\mathbf{x})$ becomes negative or zero. The alternating-variable method (a *local search method*) consists in minimizing $F(\mathbf{x})$ with respect to each input variable in turn (the other variables becoming constants), from the starting point $\mathbf{x}^0$, until a solution is found or no progress can be made for any variable. The latter case indicates that the search process fails to find solution. Once a test input $\mathbf{x}^1$ is found, the program is executed on that input. If the path is traversed, $\mathbf{x}^1$ is a solution to the test data generation problem. Otherwise, a branch violation occurs at some other node, and the whole refinement process is repeated on input $\mathbf{x}^1$.

Iterative relaxation method In contrast with the above function-minimization search approach, [30] considers all the branch conditions of the path for refinement with the current test input $\mathbf{x}^0$. In this method, each branch condition E_1 *op* E_2 is also transformed into the equivalent $F_i(\mathbf{x})$ *op* 0. All the $F_i(\mathbf{x})$ are first approximated by the tangent plane of $F_i(\mathbf{x})$ at $\mathbf{x}^0$ (denoted by $\widetilde{F_i}(\mathbf{x})$). By this approximation, we have that $F_i(\mathbf{x}^0) = \widetilde{F_i}(\mathbf{x}^0)$, and that $\forall \mathbf{x} \neq \mathbf{x}^0$, $F_i(\mathbf{x})$ is approximated by $\widetilde{F_i}(\mathbf{x})$. Of course, $\widetilde{F_i}(\mathbf{x})$ is the exact approximation, if $F_i(\mathbf{x})$ is a linear function of the input. For example,

- a function $F(X, Y, Z)$ is approximated by $\tilde{F}(X, Y, Z) = aX + bY + cZ + d$, where a, b and c are respectively the slopes (or derivatives) of F —at the point (X_0, Y_0, Z_0)— with respect to variables X, Y and Z, and d is the constant.
- In this method, the slopes of F are however approximated by its divided differences. For instance,
  $a = \frac{F(X_0+\Delta X, Y_0, Z_0) - F(X_0, Y_0, Z_0)}{\Delta X}$, where they choose $\Delta X = 1$ for a unit increment in variable X.
- Once a, b and c are computed, d is computed from
  $aX_0 + bY_0 + cZ_0 + d = F(X_0, Y_0, Z_0)$.

One then finds $\triangle\mathbf{x}^0$ (increments to $\mathbf{x}^0$) such that all the $\widetilde{F_i}(\mathbf{x}^0 + \triangle\mathbf{x}^0)$ are negative or zero by using the Gaussian elimination method. Note that Gaussian elimination is a widely implemented method [55] for solving a system of linear equations. If the $F_i(\mathbf{x})$ are linear functions of the input, the refinement is done in one iteration. For non-linear functions, it may take more than one iteration to find the new input forcing the path executed. If the path is traversed on the refined input, the refinement terminates. Otherwise, $\mathbf{x}^1 = \mathbf{x}^0 + \triangle\mathbf{x}^0$ is further refined to obtain a desired input.

Discussion

These program-execution based approaches exploit its dynamic nature to overcome some limitations (the handling of arrays, pointers, and dynamic data structures) of the approaches based on symbolic evaluation. However, the number of iterations (program executions) required before the finding of a final input depends much on the complexity of the constraints on the path. Moreover, if the path is infeasible and/or the associated constraints nonlinear, these approaches may become difficult to apply. Indeed,
(1) since the alternating-variable method, employed by [46], is a local search method, it suffers from a well-known problem in local search as also illustrated in [25]: a local minimum is obtained without being able to reach a global minimum.
(2) The refinement of test inputs in [30] is not carried out on original branch functions, but on their approximations that are in a linear form. Hence, if the original branch functions are nonlinear and complicated, the refinement may be done in many iterations without assurance of convergence to a solution.

2.1.3 A mixed approach

An approach, incorporating ideas from symbolic evaluation, constraint-based testing and dynamic test data generation, is proposed in [53]. It takes an initial set of values for each input variable, and dynamically moves the values through the control flow graph of the program. The sets of values are reduced as the branches of the path are taken. This allows branch conditions of the path to be solved immediately at each branch. For example, suppose x and y are two integer variables with their domains being $[0, 100]$, and the constraint $x > y$ is encountered. The new domain for x will be $[51, 100]$, and for y, $[0, 50]$. The new domains are chosen such that their sizes are balanced, and that the constraint is satisfied for all pairs of values in the new domains. By this choice, the domain reduction can remove values that are solutions, such as $x = 40$ and $y = 30$. Moreover, it has difficulty in handling general expressions (e.g. non-linear expressions), as reducing the domains of the variables involved in a constraint such as $exp_1 > exp_2$, where exp_1 and exp_2 are expressions, is hard in the general case. Once the path is traversed, the domains of the variables represent thus test cases that will cause execution of the path. If the domain of any variable is empty, two cases are possible. First, the path is infeasible, and hence no values can be found. Second, the constraints were complicated, making the approach inefficient, as discussed above.

2.2 Related Work on Statement Coverage

Statement (or branch) coverage aims to generate test data exercising a node (or branch) of the control flow graph. Various approaches can be found in the literature.

They are generally classified [21] as *random*, *path-oriented*, or *goal-oriented*. Path-oriented means that one needs to select a path(s) to reach the specified statement, and then generates test input for the path; while with goal-oriented, the generation of test data to execute the statement is carried out irrespectively of the path taken, i.e. the path selection is not needed. We can also classify them following the underlying technique used by each approach as follows.

2.2.1 Random test data generation

Random test data generation [19] consists in trying test data generated randomly until the statement is executed. Here are the advantages of a random test data generator.

- It is easy to implement, because no program analysis is needed. The only tool required is a random number generator. Then, we need to verify if the selected program element is exercised on execution of a randomly generated test input. This verification can be realized by instrumentation of the program, i.e. by insertion of code to show that the selected program element is actually traversed.
- A big number of test inputs can be generated in a short time. However, the checking of the execution results with such test inputs would require a considerable human effort.

Many experiences [21] have shown however that it can be very inefficient to generate test data for complex programs. Such experimental observations confirm the following theoretical ones. The program elements corresponding to a bigger sub-domain (a test input drawn from such a sub-domain will execute such program elements) will have more chances to be exercised. This goes however in a contrary direction with the program elements corresponding to a smaller sub-domain. They are more difficult to be exercised. Moreover, infeasible program elements (corresponding to an empty sub-domain) cannot be detected automatically. Let us illustrate a difficulty with random testing by the following example. Suppose that we want to construct a program to check the type of a triangle. A triangle is equilateral if $a = b = c$, where a, b and c denote the lengths of its three sides. Assuming the input domain for (a, b, c) is $([1, 1000], [1, 1000], [1, 1000])$, and a, b and c are of an integer type, then the probability such that $a = b = c$ is $(1/1000) * (1/1000) = 10^{-6}$. Of course, a much more smaller probability can be envisioned with floating-point domains.

2.2.2 Statistical testing

In standard random methods, all elements in the input domain of the program have the same probability to be selected. This contrasts with *statistical testing* [67], a probabilistic method, which consists in selecting test data randomly according to some distribution over the input domain of the program. The idea is thus to associate a probability to each sub-domain of the input domain, based on some functional or structural criteria, e.g. a distribution such that all statements have the same probability to be traversed.(Note that statistical testing based on structural criteria is usually referred to as *statistical structural testing.*) A code analysis is thus required to determine a probability distribution over the input domain. Therefore, its implementation is far more complicated than a random test data generator. However, as pointed out by [67], if statistical testing is constructed based on the all-statements criteria, for instance, then the generated test cases can have a more or less equivalent power in fault detection, compared with an usual deterministic method for the all-statements criteria.

[27] proposes a new way to deal with statistical structural testing, based on the uniform random generation of execution paths, but not on the construction of a probability distribution on the input domain as the above statistical method. Statistical testing based on a structural criterion weaker than the all-the-paths criterion is defined as: Given an integer n, we have to generate randomly one or several paths of length $\leq n$ such that all possible paths have the same probability to be generated. Once such paths are generated, each of them is transformed into path constraints, that are solved by a constraint solver (for constraints over boolean and integer variables). Statistical testing based on the all-the-statements or all-the-branches criteria is then proposed. The objective is to generate random execution paths such that all statements (or branches) have balanced probabilities to be covered.

2.2.3 Program execution based approaches

In *program execution based* (or *dynamic*) approaches, as discussed above, a first test data(s) is initiated with a (randomly) chosen input(s). If an undesirable execution flow is observed at some branch in the program, then a refinement process is used to find a new input(s) that will change the execution flow at this branch. The refinement is realized by applying function minimization algorithms [21] (goal-oriented), an iterative relaxation method [32] (path-oriented), *genetic algorithms* [54] (goal-oriented), *simulated annealing* [65] (goal-oriented), etc. Although dynamic approaches are powerful in handling arrays and dynamic data structures, they may require a great number of executions when the program involves many nonlinear conditions.

Developments The chaining approach proposed in [21] is actually an extension of [46]. The approach uses data dependencies to identify statements that affect the execution of the given statement. By requiring that these statements be executed before the given statement, the chances of executing the given statement may be increased.

The results of [21] are then extended in [47] to programs with procedures by considering the possible effect of statements in the called procedures on execution of the selected element.

[32] is also an extension of [30] to branch coverage. A list of basic paths [56] reaching the given branch is first calculated. The approach selects a path among these paths to generate test data. The path selection is guided by a path resistance measure, that is a function of the following parameters:

- The complexity of the branch conditions on the path: the path resistance increases much faster with the increase in the number of nonlinear branch conditions.
- The execution behavior of the branch conditions on the path: a path offers a larger resistance, if the algorithm has switched from the path to another path, a larger number of times.

When the path offering the least resistance is found at each step, test data generation for path coverage as proposed in [30] is applied. A major problem that can be faced by the method lies in the selection of infeasible paths [14]: after identifying a set of basic paths covering a branch(s), some of the selected paths may be found to be infeasible.

We will summarize hereafter the approach [54], based on genetic algorithms.

A genetic-algorithms based approach [54] Test data generation for a testing requirement *target* (a node or branch) starts with an initial set (or *population*, in genetic terms) of random test inputs. If *target* is satisfied by any test input from the current population, the search process terminates. Otherwise, a new population is constructed from the current population such that it is hoped to come closer to covering *target*. The first step of the construction consists in selecting a part of the current population as parents in the next generation of test inputs. All test inputs of the current population are evaluated with a *fitness* function, and thereby test inputs with the highest fitness values are selected first. In this work, the fitness function is derived from the control-dependence information [22] of the program. Control dependencies capture the conditions required for execution of the *target*. Therefore, a test input satisfying a higher number of conditions on execution, will have a higher fitness value. Once a part of the current population is selected, it is then passed through *recombination* and *mutation* operations to generate a new population.

- A recombination operation takes as input, two test inputs $(a_0, a_1, \dots, a_k)$ and $(b_0, b_1, \dots, b_k)$, and produces two new test inputs $(a_0, a_1, \dots, a_{i-1}, b_i, \dots, b_k)$ and $(b_0, b_1, \dots, b_{i-1}, a_i, \dots, a_k)$.
- A mutation operation takes as input, a test input $(a_0, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_k)$, and produces a new test input $(a_0, \dots, a_{i-1}, b, a_{i+1}, \dots, a_k)$, where b is a randomly generated value.

The whole cycle is thus repeated with the resulting population, until either the *target* is satisfied, or a maximum number of attempts (each attempt corresponds to a newly generated population) or the time limit is exceeded.

2.2.4 A consistency-based approach

A goal-oriented approach, closely related to our work and based on *Constraint Logic Programming* (CLP) techniques, is given in [26]. The test data generation problem for a given statement is translated into constraints, solved by an instance of the CLP scheme: $CLP(\mathcal{FD})$ (the variables can only take values in finite domains). The general idea is to transform an imperative program (a program under test) into a $CLP(\mathcal{FD})$ program.

- Each variable of the initial program is transformed into a set of logic variables.
- Each instruction is transformed into a constraint or an operator of the $CLP(\mathcal{FD})$. Note that two specific operators have been introduced in the underlying $CLP(\mathcal{FD})$ to model the instructions of the control flow such as `if`, `while`, etc.

The choice of a point (statement) in the program results in a system of constraints. A test case reaching the selected point is then a solution of the constraint system. If there is no solution for the constraint system, it is a proof that the selected point is non-executable, i.e. no test case can make its execution.

Note that the solving of the $CLP(\mathcal{FD})$ relies on *consistency techniques* and a search process. Consistency techniques are designed to reduce the search space (of test data) by reducing the domains of the variables. The search process combines an enumeration process with an inference process of new constraints.

The approach offers advantages such as the handling of arrays and a restricted class of pointers. However, only integer inputs are treated. Note that a constraint solver over float numbers has later been proposed in [50]. It was shown how such a solver could be used in test data generation. The solver is however limited to floating-point data. In [26], procedure calls are handled, but only intraprocedural control dependences of the test program are used in the search process, even with the presence of procedure calls. Therefore, this is not precise for certain classes of programs as will be shown later.

Note that in [26], a state of the art on the use of CLP in software testing was given. For example, in an approach [48] to functional testing based on algebraic specifications, test cases (input data and expected output) are generated by using CLP. In such work, the operations of a program are specified by axioms (formulas of the first order logic). A test is therefore defined by an instance of one of these axioms. A further step is needed to transform such axioms into Horn clauses, allowing the use of *Prolog* to generate test cases.

2.3 Conclusion

In this chapter, we presented a state of the art of existing methods for structural test data generation. Such methods fall into two classes: path coverage vs statement (branch) coverage. By path coverage, we mean that the method is primarily intended to generate test data for a path of the program; by statement (branch) coverage, test data generation is oriented towards a statement (or a branch).

It should be emphasized that our classification of methods may face the following confusion: a path coverage method may be finally used to deal with statement coverage. A naive approach consists in selecting a set of paths covering a given statement. Test data generation for path coverage can be applied to each of these paths until a path is generated test data successfully. Unfortunately, such an approach faces a big problem: many paths going through a statement may be infeasible [14]. Therefore, an excessive effort may be wasted on such paths. To lift the confusion, a path-oriented method for statement coverage must be accompanied by an "intelligent" strategy to select more likely feasible paths among a (infinite) set of paths going through a statement. [32] is an example of that case. To deal with branch coverage, a set of paths traversing a given branch is first selected. At each step (iteration), a strategy is used to switch to the most likely feasible path among the paths, based on execution behavior of all previous iterations.

An important point is worth discussion again: the problem of test data generation for a node, a branch or a path is unsolvable in the general case [68], thereby every approach to such problem cannot thus be complete by nature. Despite that fact, many "partial" methods and systems as presented in this chapter, and many others never mentioned in this thesis, have been developed over the years to help the tester to a maximum extent possible. In some restricted cases (e.g. linear path constraints, programs without loops, ...), solvability can however be established to a certain degree.

Our classification of methods can also take another dimension: static vs dynamic. For a static method, test data generation is carried out without any execution of the test program in a real environment while execution of the test program or a simulation of such an execution is an indication of a dynamic method. Among static methods are symbolic execution, the consistency or *CLP* based method [26] as presented above, etc. Among dynamic methods are random test data generation, program execution based methods, ...

A possible strong point of static methods is that a related test data generator can be written in a fixed language independent of the languages used by the test program. The purpose is thus to make it possible to test programs written in many different languages with common features. To build such a generator, an internal representation (e.g. by the control flow graph) common to such different languages can be used. And whenever a new language is used for the test program, an extension to the generator can be built by only transforming the test program into the internal representation. A weak point of static methods is however that the generated test data are not guaranteed to execute the selected program elements, because no program execution or run-time verifications (unsoundness of floating-point arithmetic, underflow, overflow, ...) on an actual environment are realized. On the other hand, the reliability of test data generated by a dynamic method is assured. Therefore, ideally, one should build a test data generator in using the same language as used by the test program, as the same environments are used both for running the test program and for generating test data. However, extensions of the generator to other languages for the test program may be far more difficult because of such dependence on real environments. In the worst case, a new test data generator would even be constructed from scratch. It should be noted that in the testing literature, a test data generation method is, to our knowledge, either a totally static or a totally dynamic method.

One can, of course, apply a hybrid approach: test data are first generated statically, and then run by the user or automatically, on a real environment, to assure that they actually execute the chosen program elements. Such an approach is actually used in our work:

- The test program is written in *C*.
- Our test data generator is written in *Java*.

- Test data resulted from the solving of path constraints in Java, are verified by running a corresponding instrumented program in C with that test data.

A possible drawback of such approach is however that the solution space in C and that in Java may be different, because in Java, only one rounding mode is used for floating-point data (a rounding mode specifies how the result of an operation is rounded to make it a floating-point number) while in C, four rounding modes can be used, and each rounding mode corresponds to a different solution space. This also support the above ideal that a test data generator and a test program should be written in the same language. Hence, the use of a hybrid approach in our work, although not ideal, guarantees at least the reliability of test data, but also the facility of extension to other languages for the test program such as *Pascal*, ...

A summary of existing approaches with functionalities close to our method is also given in Table 11.4 (Chapter 11).

Chapter 3

Background and Notations

As suggested by its title, this chapter aims at presenting the background and notations used in this work. A background on test data generation is first given. We then study a background on constraint programming and consistency. Basic notions of the classical interval programming used in constraint solving are next presented. We finally discuss on how interval arithmetic is defined and actually implemented on machine.

3.1 Background on Test Data Generation

3.1.1 Transforming a Test Program into an Equivalent one

The purpose of this transformation is to isolate all embedded function calls from their enclosing expressions. For each embedded function call, a new variable is added to hold its return value into the test program [1]. The transformed program is equivalent to the original one, assuming that, in an expression, all embedded function calls are evaluated. This might not be the case for non-strict operators such as the conditional AND (&&) in Java. In an expression like `x>1 && f(x)`, if `x>1` evaluates to `false`, the value of the expression is `false`, and `f(x)` is not evaluated. This restriction can easily be lifted by a more elaborated transformation, e.g. conditional AND are transformed into conditional statements.

For example, the C program (Program-1) in Figure 3.1 contains the function `B` with two embedded function calls. Figure 3.2 shows the transformed function `B` without embedded function calls. In the sequel, when we refer to a program, we mean an equivalent one without embedded function calls.

3.1.2 Control Flow Graph

The control flow of a program is usually represented by a *Control Flow Graph* (CFG) [61]. Formally, the CFG for a procedure P is a directed graph, where the nodes represent statements and the edges represent possible flow of control between nodes. The CFG contains two distinguished nodes, $Entry_P$ and $Exit_P$, representing respectively a unique entry node and a unique exit node of P. A node, representing a

```
void M(double a[10], int c) {
  int i = 1;
  while (i <= c) {
    B(a);
    i = i+1;
  }
}


void C(double *x, double *y) {
  double t;
  if (*x > *y) {
    t = *x;
    *x = *y;
    *y = t;
  }
}
```

```
void B(double a[10]) {
  int i,j;
  scanf("%d %d", &i, &j);
  if (F(i) < F(j))
    C(&a[i], &a[j]);
  else C(&a[j], &a[i]);
}


int F(int i) {
  if (i >= 0 && i <= 9)
    return i;
  else exit(1);
}
```

Figure 3.1: Program-1

```
B(double a[10]) {
  int i,j,fi,fj;
  scanf("%d %d", &i, &j);
  fi = F(i);
  fj = F(j);
  if (fi < fj)
    C(&a[i], &a[j]);
  else C(&a[j], &a[i]);
}
```

Figure 3.2: An equivalent of Program-1's function B

(conditional or loop) statement, is called a *decision node* (a point where control flow can split in several branches). A list of assignments without decisions is grouped in a *basic block node*. Each procedure call is represented by two nodes, a *call node* and a *return node*. An outgoing edge from a decision node is called a *branch*. Each branch of the CFG is associated with a *condition*.

Control-flow interactions among a procedure and its related called procedures are usually represented by an *Interprocedural Control Flow Graph* (ICFG) [61, 49]. Formally, the ICFG for a procedure P is a directed graph, which consists of a unique global entry node $Entry_{global}$, a unique global exit node $Exit_{global}$, and the CFGs (for P and all procedures called directly or indirectly by P). Apart from the edges of the individual CFGs, the ICFG also contains the following kinds of edges:

- the edges $(Entry_{global}, Entry_P)$ and $(Exit_P, Exit_{global})$;
- each procedure call (represented by a call node c and a return node r) to procedure M corresponds to a *call edge* $(c, Entry_M)$ and a *return edge* $(Exit_M, r)$;
- the edges that connect the nodes (representing a `halt` statement) to node $Exit_{global}$.

Note that a `halt` statement represents an unconditional program halt such as the `exit()` system call in `C`. Each statement such as `x:=f(...)`, where `f(...)` is a function call, is represented by a pair of call and return nodes as in a procedure call. However, these nodes are now associated with `x:=f(...)`. Informally, an ICFG is constructed by connecting the individual CFGs at call sites.

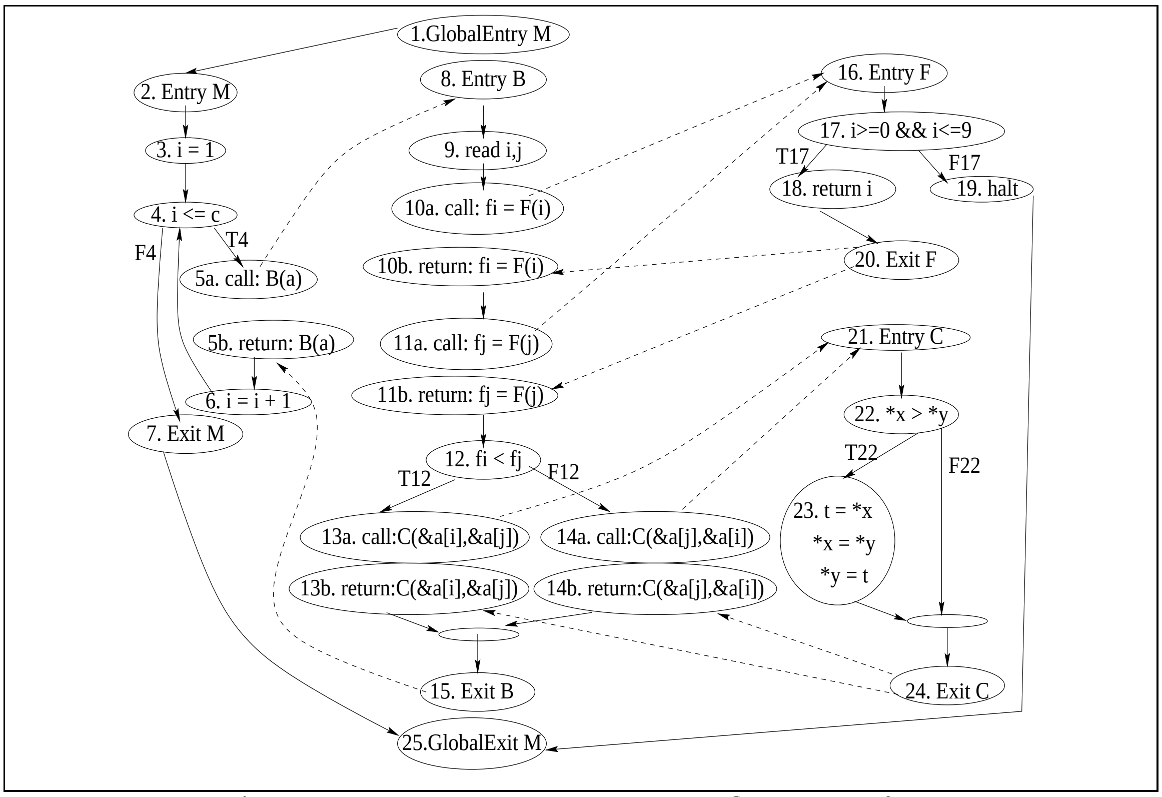


Figure 3.3: Interprocedural control flow graph for `M`

Figure 3.3 shows the ICFG for procedure `M` of Program-1 in Figure 3.1. The individual CFGs are connected by edges shown in dashed lines. If node i is a decision node, its true branch is labeled with a condition Ti, while its false branch is labeled with a Fi, that is the negation of Ti ($Fi = \neg Ti$). In this ICFG, the conditions $T4$, $T12$, $T17$, and $T22$ are respectively $i \leq c$, $fi < fj$, $i \geq 0 \,\&\&\, i \leq 9$, and $*x > *y$.

3.1.3 Path

A *Path* is a sequence of nodes from the global entry node $Entry_{global}$ to a node of the ICFG. Note that a (partial) execution of a procedure P corresponds to an *execution path* in the ICFG for P. Paths, where a return edge does not match the corresponding call edge, are obviously infeasible execution paths. We thus restrict paths to feasible execution paths, where every return edge is properly matched with its corresponding call edge. Note that a path can be an *unbalanced-left path* [49], representing an execution in which not all of the procedure calls have been completed, i.e. there are more call edges than return ones in the path.

3.2 Notations and Definitions

The following notations and definitions are borrowed directly from, or based on, those in [36]. They are rather standard in interval programming.

- $\mathcal{R}$ denotes the set of real numbers (reals).
- $\mathcal{F}$ denotes the set of floating-point numbers (float numbers) represented on a computer. $\mathcal{F}$ is thus a finite subset of $\mathcal{R}$. The elements of $\mathcal{F}$ are called $\mathcal{F}$-numbers.
- $\mathcal{B}ool$ denotes the set $\{false, true\}$.
- Capital letters denote intervals.
- The set of intervals is denoted by $\mathcal{I}$.
- The set of boolean intervals is denoted by $\mathcal{BI}$, where $\mathcal{BI} = \{[0,0],[0,1],[1,1]\}$ (0 and 1 represent respectively $false$ and $true$). $\mathcal{BI}$ is thus a subset of $\mathcal{I}$.
- If a is a real, a^+ denotes the smallest $\mathcal{F}$-number greater than or equal to a, and a^- the largest $\mathcal{F}$-number smaller than or equal to a. This means that if a is a real equal to an $\mathcal{F}$-number, then $a^- = a = a^+$; otherwise, a^- and a^+ are two successive $\mathcal{F}$-numbers.
- If a is an $\mathcal{F}$-number, a^+ denotes the smallest $\mathcal{F}$-number strictly greater than a, and a^- the largest $\mathcal{F}$-number strictly smaller than a. This means that a^-, a and a^+ are three successive $\mathcal{F}$-numbers.
- If x is a real, $\lfloor x \rfloor$ denotes the largest integer that is not larger than x and $\lceil x \rceil$ the smallest integer that is not smaller than x.
- The lower and upper bounds of an interval X are $\mathcal{F}$-numbers and denoted respectively by *left*(X) and *right*(X).
- Boldface letters denote vectors of objects.
- The domain of a simple variable x is denoted by *dom*(x). If a is an array variable, *dom*(a) denotes the domain for its array elements and *length*(a) its length (i.e. the number of elements).
- Let O be any object, $O[x/v]$ denotes the substitution of v for x in O.

Definition 3.1 (canonical interval). A *canonical interval* is an interval of the form $[a, a]$ or $[a, a^+]$, where a is a $\mathcal{F}$-number.

Definition 3.2 (ϵ_*interval*). An interval X is an ϵ_*interval* ($\epsilon > 0$) if X is canonical or *right*$(X)-$*left*$(X) \leq \epsilon$.

Definition 3.3 (ϵ_*box*). A *box* $(X_1, \ldots, X_n)$ is an ϵ_*box* if X_i $(1 \leq i \leq n)$ is an ϵ_*interval*.

3.3 Background on Consistency

3.3.1 Path Constraint

A *basic constraint* is a simple relational expression of the form E_1 *op* E_2, where E_1 and E_2 are arithmetic expressions and *op* is one of the following relational operators $\{<, \leq, >, \geq, =, \neq\}$. A *constraint* is a basic constraint or a logical combination of basic constraints using the following logical operators $\{NOT, AND, OR\}$. We assume that the logical operators of the programming language of the program under analysis correspond to those constraints. Otherwise, the constraints can easily be extended. A path of the ICFG can be represented by a list of constraints with one constraint for each condition on the path. This list of constraints is called a *path constraint* where the constraints of the list are connected by the logical AND.

3.3.2 CSP, Consistency, and Constraint Solving

Many important problems in areas like artificial intelligence and operations research can be viewed as *Constraint Satisfaction Problems* (CSP). A CSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ is defined by a finite set of variables $\mathcal{V}$ taking values from finite or continuous domains $\mathcal{D}$ and a set of constraints $\mathcal{C}$ between these variables. A solution to a CSP is an assignment of values to variables satisfying all constraints and the problem amounts to finding one or all solutions. Most problems in this class are $\mathcal{NP}$-complete, which means that backtracking search is an important technique in their solution.

Consistency techniques are constraint algorithms that reduce the search space by removing, from the domains and constraints, values that cannot appear in a solution. Consistency algorithms play an important role in the resolution of CSP [66], and have been used extensively in many constraint softwares such as Numerica [36], Prolog IV [3], CLP(BNR) [2], etc.

3.3.3 Test cases

An (integer, boolean or float) *input variable* is either an input parameter or a variable in an input statement of program P. The domain of a boolean variable is an element of BI. The domain of an integer variable is an interval, representing a set of consecutive integers. The domain of a float variable is an interval of $\mathcal{F}$-numbers. Note that since our work relies on interval programming as will be presented in the next section, the domain of a variable is represented by an interval. Let $x_1, \ldots, x_n$ be n input variables of P, and D_k be the domain of variable x_k $(1 \leq k \leq n)$. Then a *test input* is a vector of values $(i_1, \ldots, i_n)$, where $i_k \in D_k \quad (1 \leq k \leq n)$.

The execution of the program (on the specified path) uses operators defined on $\mathcal{F}$-numbers, integers, and booleans. We assume here that the test program is written in some fixed imperative language $\mathcal{L}$.

Definition 3.4 (eval). Let c be a constraint, and $\mathbf{v}$ be a test input. The predicate $eval(c, \mathbf{v})$ holds if execution of c with $\mathbf{v}$ using the operators of the programming language $\mathcal{L}$ yields true. The test input $\mathbf{v}$ is said to be a float $\mathcal{L}$ solution.

Definition 3.5 (Path Constraint). A constraint c is said to be a *path constraint* for a path p if for all test input $\mathbf{v}$, $eval(c, \mathbf{v})$ holds iff the execution of the program traverses the path p.

Definition 3.6. Given a path p (of an ICFG), a test input $\mathbf{v}$ is a *test case* for p if $eval(c, \mathbf{v})$ holds, where c is a path constraint for p.

Definition 3.7. Given a node n (of an ICFG), a test input $\mathbf{v}$ is a *test case* for n if there exists a path p traversing n such that $\mathbf{v}$ is a test case for p.

A test case is thus a test input traversing the specified path or reaching the specified statement. When no test case exists, the path is said to be *infeasible.*

The predicate $eval(c, \mathbf{v})$ can be realized in different ways by either executing the program under analysis, or by simulating such an execution (when the real environment is not available).

It is important to distinguish the real (or mathematical) solutions of a path constraint from its test cases (float $\mathcal{L}$ solutions). First, a mathematical solution may not be a float number. Second, a float (mathematical) solution $\mathbf{v}$ of a path constraint may not traverse the specified path, i.e. $c(\mathbf{v}) \not\Rightarrow eval(c, \mathbf{v})$. For example, the constraint, $c(x) \triangleq x = \frac{x}{3} + \frac{x}{3} + \frac{x}{3}$, is mathematically true for all $\mathcal{F}$-number in $\mathcal{F}$. However $eval(c, 1)$ may evaluate to false in some programming languages. Likewise, constraints may have float $\mathcal{L}$ solutions, while having no mathematical solution, i.e. $eval(c, \mathbf{v}) \not\Rightarrow c(\mathbf{v})$. [50] illustrates that the constraint, $16.0 + x = 16.0 \ \wedge \ x > 0$, actually possesses many float $\mathcal{L}$ solutions.

3.4 Classical Interval Programming

Computation with the reals is actually difficult, since only a finite subset of reals can be represented on a computer. This means that the computer can only work with $\mathcal{F}$-numbers, and all real operations are actually operations on $\mathcal{F}$-numbers, which are commonly known as non-sound. Because the result of an operation may not be computed exactly (due to round-off errors with sub-operations), or may not be representable in $\mathcal{F}$. One must approximate a real r by r^+ (upward rounding), or by r^- (downward rounding). To solve continuous constraints (over the reals) with traditional numerical methods, one thus obtains $\mathcal{F}$-numbers that are approximations of the mathematical solutions.

Interval methods solve the constraints by a different approach, which returns small intervals enclosing the mathematical solutions. They automatically bound numerical errors, and so ensure the reliability of the results. The basic idea consists in associating with each variable an interval representing its domain. The original problem is then pruned (by some consistency techniques) before divided into sub-problems (by splitting the interval associated with a variable), until all solutions are obtained. Such consistency techniques (on intervals) are designed to reduce the size of the intervals without removing solutions of the constraints [36].

We now give an example to illustrate the difference between traditional numerical methods and interval methods. Given the numerical equation $x^2 = 2$, one thus obtains $x = 1.41421356$ —an approximation of the mathematical solution— by a numerical method. On the other hand, with an interval method, one obtains the interval $X = [1.41421356, 1.41422357]$, containing the mathematical solution. Since the resulting interval is larger than the exact solution, interval methods ensure thus reliable solutions.

We give here some important definitions of interval programming [52], borrowed directly from, or based on, those in [36]. Since the goal is to work with intervals, all objects of the real space —such as reals, real sets, real functions, and real relations— should have an *interval extension* in the interval space. Note that interval extensions are not unique.

Definition 3.8 (Interval). An interval $I = [a, b]$, with $a, b \in \mathcal{F}$, denotes the set

- $\{x \in \mathcal{R} \mid a \le x \le b\}$ — if we work with the real space (i.e. our concern is real solutions),
- or $\{x \in \mathcal{F} \mid a \le x \le b\}$ — if we work with the float space (i.e. our concern is float solutions).

Also, given an interval I, $left(I)$ and $right(I)$ denote respectively a and b. The set of intervals is denoted by $\mathcal{I}$.

Definition 3.9 (Interval Extension). Let S be a subset of $\mathcal{R}$. The interval extension of S, denoted by $\Box S$, is the smallest interval I such that $S \subseteq I$. When $S = \{r\}$, we denote its interval extension by $\Box r$, and its value is the interval $[r, r]$ if r is an $\mathcal{F}$-number, and $[r^-, r^+]$ otherwise.

An interval function $F : \mathcal{I}^n \to \mathcal{I}$ is an interval extension of $f : \mathcal{R}^n \to \mathcal{R}$ if $\forall \mathbf{I} \in \mathcal{I}^n : f(\mathbf{I}) \subseteq F(\mathbf{I})$, where $f(\mathbf{I}) = \{f(\mathbf{r}) \mid \mathbf{r} \in \mathbf{I}\}$.

Figure 3.4 gives an example of interval extension for a function.

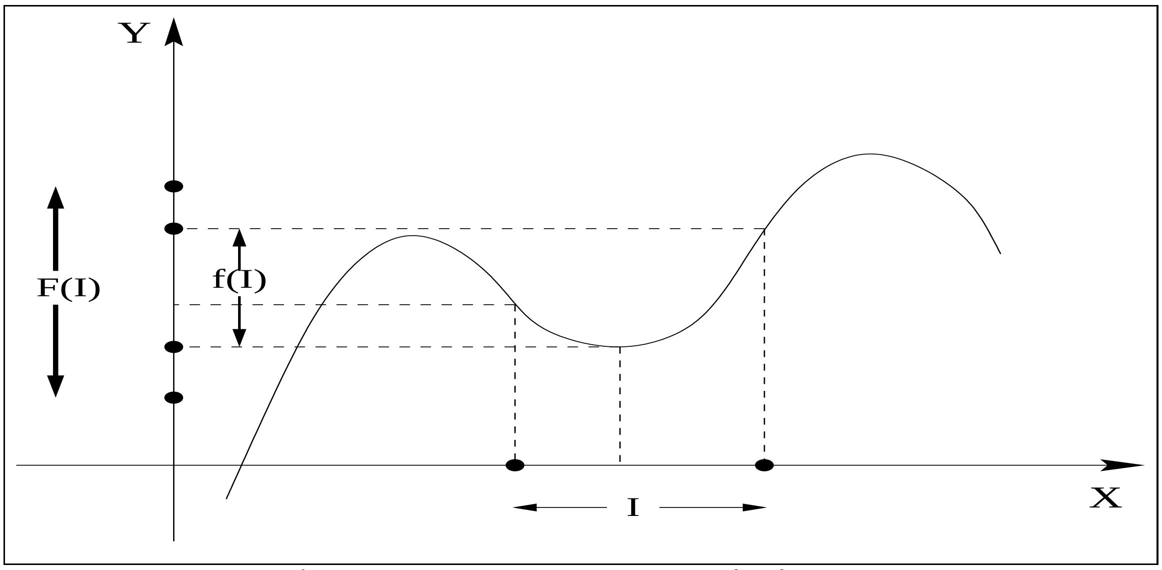


Figure 3.4: Interval extension of a function

The objective of an interval extension is to preserve the mathematical solutions. Given a function f, the *optimal interval extension* is the interval function returning $\Box(f(\mathbf{I}))$. Many usual functions possess an optimal interval extension. For example, the interval function $\oplus$, $[a_1, b_1] \oplus [a_2, b_2] = [(a_1 + a_2)^-, (b_1 + b_2)^+]$, is an optimal interval extension of the addition of two reals. Note that $a_1 + a_2$ and $b_1 + b_2$ are real operations, i.e. their results are reals. [15] reported however that there exist functions, for which one do not know any optimal interval extension due to practical reasons, e.g. $f(x) = x * sin(x)$.

Given an arbitrary function, one can obtain several (non-optimal) interval extensions [15], based on interval extensions for its primitive operations. The *natural interval extension* is however used in our work so as to also conserve float solutions, as illustrated later.

Definition 3.10 (Natural Interval Extension). Given an expression f, the *natural interval extension* of f is obtained by replacing in f, each constant k by its approximation $(\square k)$, each real variable x by the interval variable X, each real operation g by any interval extension G (not always optimal).

Proposition 3.1 ([52]). If $F : \mathcal{I}^n \to \mathcal{I}$ is the *natural interval extension* of $f : \mathcal{R}^n \to \mathcal{R}$, then F is an interval extension of f, i.e. $\forall \mathbf{I} \in \mathcal{I}^n : f(\mathbf{I}) \subseteq F(\mathbf{I})$.

By Proposition 3.1, $F(\mathbf{I})$ contains thus at least all real solutions of $f(\mathbf{I})$.

For instance, the natural interval extension of, $f(x) = x^{2.3} - 2x + sin(x)$, is the interval function, $F(X) = (X \odot \square 2.3) \ominus \square 2 \otimes X \oplus SIN(X)$, where $\ominus$, $\otimes$, $\odot$, and SIN are the interval extensions of subtraction, multiplication, exponentiation, and the trigonometric sin function.

The natural interval extension is often an over-estimation of the optimal interval extension. Consider, for example, the following function [36]:

$$f(x) = x - x$$

which always returns 0. The optimal interval extension is the following:

$$F(X) = [0, 0].$$

However, the natural interval extension, $F(X) = X \ominus X$, is over-estimated, since

$$F([0,1]) \;=\; [0,1] \ominus [0,1] \;=\; [-1,1]$$

contains the exact result with much noise. The reason for this problem is that although the two occurrences of X mean the same variable, the natural-extension evaluation considers each occurrence of X as independent of others. This over-estimation problem is due to the fact that several occurrences of the same variable appear in an expression. Also illustrated in [36] is the following interesting example. Consider the function

$$f_1(x) = x^2 - x$$

which is equivalent to

$$f_2(x) = x(x-1).$$

Unfortunately, the natural extensions of these functions do not yield the same results. For example, $F_1([0,5]) = [-5,25]$ while $F_2([0,5]) = [-5,20]$. Note that neither of these natural extensions is optimal, since an optimal interval extension will yield $[-0.25, 20]$. To reduce the over-estimation with the natural interval extension, some other forms of interval extension have been proposed in interval programming, namely *mean-value* interval extension, *Taylor* interval extension, ... An overview of these forms can be found in [15]. Although these drawbacks, using the natural extension in our work is however a reliable way (as shown later) to preserve float solutions.

Definition 3.11 (Interval extension for a relation). An interval relation $C : \mathcal{I}^n \to \mathcal{B}ool$ is an interval extension of the relation $c : \mathcal{R}^n \to \mathcal{B}ool$ if
$\forall \mathbf{I} \in \mathcal{I}^n \ : \ (\exists \mathbf{r} \in \mathbf{I} : c(\mathbf{r})) \Rightarrow C(\mathbf{I})$

For example, the interval relation $\approx$, defined in [36] as

$$I_1 \approx I_2 \ \triangleq \ I_1 \cap I_2 \neq \emptyset$$

is an interval extension of the equality relation on real numbers.

3.5 Interval Arithmetic

We will first show in this section how interval arithmetic is mathematically defined (exact interval arithmetic). We then show how exact interval arithmetic is actually implemented on a finite-precision (floating-point) machine.

3.5.1 Exact Interval Arithmetic

Interval arithmetic was first introduced in [52]. From the interval-programming background given in the previous section, exact (or mathematical) interval extensions for the real arithmetic operators $\{+, -, *, /\}$ are defined as follows.

Definition 3.12. Let $I_1 = [a, b]$ and $I_2 = [c, d]$ be two intervals such that a, b, c and d are reals, then

$$\begin{array}{l}
I_1 \oplus I_2 = [a + c, b + d] \\
I_1 \ominus I_2 = [a - d, b - c] \\
I_1 \otimes I_2 = [min\{a * c, a * d, b * c, b * d\}, max\{a * c, a * d, b * c, b * d\}] \\
I_1 \oslash I_2 = [min\{a/c, a/d, b/c, b/d\}, max\{a/c, a/d, b/c, b/d\}] \ \text{ if } \ 0 \notin I_2
\end{array}$$

Note that for the sake of simplicity, the real operators $\{+, -, *, /\}$ will sometimes be overloaded, denoting also their corresponding interval operators.

In interval programming, interval division is often defined under the condition that 0 is not contained in I_2. [38] remedies however such a restriction by providing explicit formulas for this quotient when I_1 and I_2 are any real intervals. Their formulas are actually an extension of the following *Ratz* theorem (also given in [38] for comparison with their formulas), which is an original corner-stone in dealing with the above problem of interval division.

Theorem 3.1 (Ratz). Let $[a, b]$ and $[c, d]$ be two non-empty bounded real intervals. Then

$$[a,b] \oslash [c,d] = \begin{cases} [a,b] \otimes [1/d, 1/c] & \text{if } 0 \notin [c,d] \\ [-\infty, +\infty] & \text{if } 0 \in [a,b] \wedge 0 \in [c,d] \\ [b/c, +\infty] & \text{if } b < 0 \wedge c < d = 0 \\ [-\infty, b/d] \cup [b/c, +\infty] & \text{if } b < 0 \wedge c < 0 < d \\ [-\infty, b/d] & \text{if } b < 0 \wedge 0 = c < d \\ [-\infty, a/c] & \text{if } 0 < a \wedge c < d = 0 \\ [-\infty, a/c] \cup [a/d, +\infty] & \text{if } 0 < a \wedge c < 0 < d \\ [a/d, +\infty] & \text{if } 0 < a \wedge 0 = c < d \\ \emptyset & \text{if } 0 \notin [a,b] \wedge c = d = 0 \end{cases}$$

The formulas for interval division as proposed in [38] will not however be presented here; They are somewhat different and more complicated than the above theorem. Note however that such formulas are more general than Ratz's, because they can also be used for unbounded intervals (connected intervals). Moreover, they are more efficient than Ratz's, because Ratz's formulas rely on the multiplication formulas by converting many quotients into $[a, b] * [1/d, 1/c]$. And this can unfortunately introduce additional round-off errors when evaluated in floating point arithmetic, e.g. a/d will generally be more precise than $a * (1/d)$.

Note that in the above presentation of exact interval arithmetic, we have only considered the properties of the reals ($\mathcal{R}$) and the extended reals ($\mathcal{R} \cup \{-\infty, +\infty\}$) in constructing interval extensions for the arithmetic operators. Intervals are represented by pairs of extended reals. That is, the endpoints of any interval (as the result of an interval operation) are calculated according to the arithmetic of extended reals (when infinity is involved).

In the rest of this section, we will first give an overview of the IEEE 754 standard for floating-point arithmetic. We then show how to use it to ensure *correct* (in the sense of not losing any mathematical solutions) and *optimal* (the resulting IEEE-754 floating-point interval is the narrowest interval containing the exact real interval) computations of interval extensions.

3.5.2 Overview of the IEEE-754 standard

The IEEE-754 floating-point standard [41] is the most common representation today for real numbers on computers, and most of the floating-point number systems as implemented on Intel-based PC's and most Unix platforms conform to it. We give here a brief overview of the IEEE 754 standard and its representation. A lot of this material was drawn from [39], [50], [38] and [23]. Note that [23] gives the most comprehensive presentation on floating point standards, including also the IEEE 854 standard for floating-point numbers. IEEE 754 uses binary base for representing numbers while IEEE 854, by extending IEEE 754, allows the use of both binary and decimal bases. It is however sufficient to restrict our presentation to IEEE 754.

Table 3.1: Storage layout of floating-point formats

Formats	Sign	Exponent	Mantissa	Bias
Single	1 [31]	8 [30-23]	23 [22-00]	127
Double	1 [63]	11 [62-52]	52 [51-00]	1023

Floating-point representation The standard (i.e. IEEE 754) defines several formats —such as *single* (single precision), *double* (double precision), *single extended* and *double extended*— which are different only in the sizes of certain fields. While the first two formats are well defined and implemented by most floating-point units, the two latter are not really well specified and therefore not sure to be available in a floating-point unit.

For any format, the floating-point numbers (also called floats) have three basic fields: the *sign*, the *exponent*, and the *mantissa* (also known as the *significand*). Table 3.1 shows the layout for single (32-bit) and double (64-bit) floating-point values. The number of bits for each field are shown (bit ranges are in square brackets).

- The *sign* field (denoted by s) is 0 for positive numbers, and 1 for negative.
- The *exponent* field (denoted by e) follows the following convention so as to represent both positive and negative exponents: A *bias* is added to the actual exponent in order to get the stored exponent. For single floats, the bias value is 127 (as shown in Table 3.1). Thus, an exponent of zero means that 127 is actually stored in the exponent field. A stored value of 100 indicates an exponent of -27 (100-127).
- The *mantissa* field (denoted by f) represents, in fact, the *fraction* bits (the precision bits) of the actual mantissa. That is, the actual mantissa is composed of an implicit leading digit and the fraction: The actual mantissa is assumed to be $1.f$ for *normalized* floats, or $0.f$ for *denormalized* floats (as will be discussed below), where $1.f$ and $0.f$ are binary values.

For any format, we assume that e_{max} is the maximum value that can be stored in the exponent field (an exponent of all 1s). For instance, $e_{max} = 255$ for the single format. Then, the set of possible bit patterns for $< s, e, f >$ is divided into the following classes of floats.

- *Normalized* numbers are defined when $0 < e < e_{max}$ and a leading 1 is assumed before the binary point of the actual mantissa. This represents thus the number $(-1)^s \times 1.f \times 2^{e-bias}$. Hence, for single format, the range of *positive* normalized numbers is $[2^{-126}, (2 - 2^{-23}) \times 2^{127}]$. And the range for *negative* normalized numbers is given by the negation of the above range.
- *Zero* is not directly representable in the normalized form above, because given the actual mantissa being $1.f$, we cannot specify a true zero mantissa to yield a value of zero. Hence, *signed zeros* (-0 and $+0$) are defined as special values when $e = 0$ and $f = 0$. And the sign field decides -0 and $+0$ as two distinct values, though by the standard, $-0 = +0$. Note that signed zeros are used in specific cases (e.g. $-10/-\infty = +0$) to get a more precise result. Furthermore, signed zeros can be returned in underflow situations (*underflow* means that values grow

too small –near zero– for representation). It is then easy to distinguish negative underflow from positive underflow.

- *Denormalized* numbers are defined when $e = 0$, $f \neq 0$, and a leading 0 is assumed before the binary point of the actual mantissa. This represents thus the number $(-1)^s \times 0.f \times 2^{-bias+1}$. Note that we can consider signed zeros as a special type of denormalized numbers, if we lift the condition $f \neq 0$ above. For single format, the range of *positive* denormalized numbers is $[2^{-149}, (1 - 2^{-23}) \times 2^{-126}]$. Therefore, denormalized numbers are closer to zero than normalized ones, and they continue to fill the region near zero, left uncovered by normalized numbers.
- *Infinities* ($-\infty$ and $+\infty$) are denoted with $e = e_{max}$ and $f = 0$, and the sign bit distinguishes between $-\infty$ and $+\infty$. In practice, infinities provide a way allowing computations to continue past overflow situations. *Overflow* means that values grow too large for representation. Note that returning infinities in such situations is much safer than simply returning the largest representable number.
- *NaN*'s (*Not a Number*) are represented by a bit pattern with $e = e_{max}$ and $f \neq 0$. They are used for an operation where the result is not mathematically defined, or for signaling an exception in executing operations. For example, a good way to handle exceptional situations like taking the square root of a negative number (traditionally, such situation would cause the computation to halt), is to return a *NaN*.

We now give an example to sum up the above presentation of floating-point numbers. Assume that the sign field contains 0, the exponent field 130. And the mantissa field contains the following bits:
1100000 00000000 00000000
This gives then the following number (in decimal):
$+1.11 \times 2^{130-127} = (2 - 2^{-2}) \times 2^3 = 14$

Floating-point operations and exceptions As discussed earlier, floats (denoted by $\mathcal{F}$-numbers) are a finite and discrete version of the real numbers on a computer. And all real operations are thus replaced by operations over $\mathcal{F}$-numbers. Since the result of an operation over $\mathcal{F}$-numbers may not be an $\mathcal{F}$-number, rounding is necessary to close the operations over $\mathcal{F}$. The IEEE 754 standard proposes the following four rounding modes:

- $+\infty$: which maps x to the smallest $\mathcal{F}$-number x_k such that $x \leq x_k$.
- $-\infty$: which maps x to the greatest $\mathcal{F}$-number x_k such that $x_k \leq x$.
- 0 : which is equivalent to the rounding mode $+\infty$ if $x < 0$ and to $-\infty$ if $x \geq 0$.
- *nearest even* (or *near*, for short) : which maps x to the nearest $\mathcal{F}$-number. When x has the same distance to two $\mathcal{F}$-numbers, it is then mapped to the one having 0 in the least significant bit of its mantissa field.

Definition 3.13. Let $f : \mathcal{R}^n \to \mathcal{R}$ be a real expression. Then we denote by f_r, a corresponding float expression of f, where r represents one of the following rounding modes $\{+\infty, -\infty, 0, near\}$.

Note that for all $r \in \{+\infty, -\infty, 0, near\}$:

$$f_{-\infty}(\mathbf{v}) \leq f_r(\mathbf{v}) \leq f_{+\infty}(\mathbf{v})$$
$$f_{-\infty}(\mathbf{v}) \leq f(\mathbf{v}) \leq f_{+\infty}(\mathbf{v}).$$

For many reasons (better accuracy, more precise reasoning with floating-point proofs, portability of floating-point softwares, ...), the standard requires that the result of arithmetic operations (addition, subtraction, multiplication and division) be exactly rounded. That is, the result must be computed exactly and then rounded to the nearest float (following the used rounding mode). For example, with Definition 3.13 above, the result of $f_r(\mathbf{v})$ must be the nearest float to $f(\mathbf{v})$ according to rounding mode r. In other words, the rounding of an operation $f(\mathbf{v})$ is exact if the rounding towards $-\infty$, $f_{-\infty}(\mathbf{v})$, will return the biggest $\mathcal{F}$-number smaller than or equal to the exact operation $f(\mathbf{v})$, and the rounding towards $+\infty$, $f_{+\infty}(\mathbf{v})$, will return the smallest $\mathcal{F}$-number greater than or equal to $f(\mathbf{v})$. In addition to the basic operations ($+$, $-$, $\times$ and $/$), IEEE 754 also specifies that square root, remainder, and conversion between integer and floating-point be correctly rounded. The standard does not require transcendental functions (e.g. *exp*) to be exactly rounded because of many practical problems as illustrated in [23].

When an exceptional condition –such as division by zero, underflow, overflow or invalid operation– occurs in IEEE 754 arithmetic, the default is to deliver a special result and continue. Typical default results are NaN for $0/0$ and $\sqrt{-1}$ (invalid operation), and ∞ for $1/0$ and overflow, etc. When an exception occurs, a status flag is also set. And testing the flags is the only way to distinguish $1/0$ (an actual infinity) from an overflow. The standard also requires each floating-point operation to raise an *inexact exception* when the result is not mathematically exact.

Given the effect of rounding on floating-point arithmetic —causing the so-called *round-off errors*— many properties on the reals are not preserved by the floats. For example, assuming a, b and c are floats, we cannot ensure that
$(a + b) + c = a + (b + c)$
in floating-point arithmetic. As a consequence already illustrated in Sub-section 3.3.3, the mathematical-solutions space of a path constraint can be totally different from its float-solutions space. Moreover, each rounding mode also produces a different float-solutions space.

Operations with special numbers Operations involving special numbers (-0, $+0$, $-\infty$, $+\infty$, and NaN's) are well defined by the standard. In the simplest case, any operation with a NaN yields a NaN result. Assuming NF stands for negative (normalized or denormalized) float and PF for positive float, Table 3.2 gives some arithmetic operations (in all rounding modes) with special numbers for the purpose of illustration. More detailed information of the arithmetic operations on floating-point numbers, can be found in [38]. For example, $(-0) +_r (+0) = +0$ for all rounding modes $r \neq -\infty$, and $(-0) +_{-\infty} (+0) = -0$.

Table 3.2: Some operations with special numbers

Result	Operations					
NaN	$(+\infty)+(-\infty)$	$(-\infty)-(-\infty)$	$(+\infty)-(+\infty)$	$\pm 0\times\pm\infty$	$\pm 0/\pm 0$	$\pm\infty/\pm\infty$
$-\infty$	$-\infty+(-\infty)$	$-\infty\times+\infty$	$-\infty/+0$	$NF/+0$	$PF/-0$	$+\infty/-0$
$+\infty$	$+\infty+(+\infty)$	$-\infty\times-\infty$	$-\infty/-0$	$NF/-0$	$PF/+0$	$+\infty/+0$
$+0$	$NF/-\infty$	$-0/-\infty$				

3.5.3 IEEE-754 Interval Arithmetic

[38] proposes an interval arithmetic framework based on the IEEE-754 standard, that we will refer to as the *IEEE-754 interval arithmetic*. Given the purpose that interval arithmetic must preserve the mathematical solutions, interval extensions for the addition and subtraction are defined as follows.

Definition 3.14. Let $I_1=[a,b]$ and $I_2=[c,d]$ be two intervals such that a, b, c and d are $\mathcal{F}$-numbers, then

$$\begin{aligned} I_1+I_2 &= [a +_{-\infty} c, b +_{+\infty} d] \\ I_1-I_2 &= [a -_{-\infty} d, b -_{+\infty} c], \end{aligned}$$

where the floating-point addition and subtraction with rounding towards $-\infty$ (or $+\infty$) are denoted $+_{-\infty}$ and $-_{-\infty}$ (or $+_{+\infty}$ and $-_{+\infty}$), according to Definition 3.13.

Definition 3.14 gives thus an idea of how interval arithmetic should be implemented on a machine using floating-point arithmetic. By outward rounding, one claims thus that the computed interval contains the exact interval. For example, with the addition, one informally claims (both in [38] and [23])

$$[a+c,b+d]\subseteq[a +_{-\infty} c, b +_{+\infty} d] \tag{3.1}$$

by the possible fact that $a +_{-\infty} c \le a+c\le b+d\le b +_{+\infty} d$.

We find however an exception to the above claim (Property 3.1). Suppose that $a=b=c=d=MAX\_F$, where MAX_F denotes the largest representable float in a given format. Then, we possibly obtain
$a +_{-\infty} c = +\infty$ and $b +_{+\infty} d = +\infty$
by the handling of the floating-point standard for overflow situations. In that case, Property 3.1 is violated, because
$[a+c,b+d]=[2*MAX\_F, 2*MAX\_F] \;\not\subseteq [a +_{-\infty} c, b +_{+\infty} d]=[+\infty,+\infty]$.

To overcome the above problem, we propose the following definition for interval addition.

Definition 3.15. Let $I_1=[a,b]$ and $I_2=[c,d]$ be two intervals such that a, b, c and d are $\mathcal{F}$-numbers. Let MIN_F and MAX_F be respectively the smallest and the largest representable $\mathcal{F}$-numbers. Then

$$I_1+I_2=\begin{cases}[-\infty, MIN\_F] & \text{if } b +_{+\infty} d = -\infty \\ [MAX\_F,+\infty] & \text{if } a +_{-\infty} c = +\infty \\ [a +_{-\infty} c, b +_{+\infty} d] & \text{otherwise}\end{cases}$$

It is easy to show that by Definition 3.15, Property 3.1 still holds. Note first that if $b +_{+\infty} d = -\infty$, then $b + d$ must be infinite or a negative overflow value, and that if $a +_{-\infty} c = +\infty$, then $a + c$ must be infinite or a positive overflow value. However, it is possible that $b + d$ is a negative overflow value while $b +_{+\infty} d = MIN\_F$, and that $a + c$ is a positive overflow value while $a +_{-\infty} c = MAX\_F$. Second, a possible application of such definition should be investigated as well.

It is not very helpful if the computed interval turns out to be large, since the mathematical solution could be anywhere in that interval. When a floating-point operation is assumed to be exactly rounded (like the basic operations), this is a necessary condition to construct an *optimal* interval extension (the computed interval is the smallest IEEE-754 interval containing the exact interval). Otherwise, optimal interval extensions are not assured. For the basic operations above, it is a simple matter to show that their interval extensions are optimal.

IEEE-754 interval extensions for the operations of multiplication and division, follow the same principle (outward rounding) in their definitions. See [38] for more details. In that work, any IEEE-754 interval is required to satisfy the following: -0 can only appear as a right endpoint, and $+0$ can only appear as a left endpoint. This is a nice property to facilitate dealing with interval division.

In the next, we will discuss about how interval arithmetic has been implemented in Java, a programming language that uses only the rounding mode *near*.

3.5.4 An Interval Arithmetic in Java

In Java, to our knowledge, only the rounding mode *near* is used. Interval arithmetic was implemented in Java in an interval library [37] as in Definition 3.16 below. Note that Definition 3.16 was derived from our reading of the code in [37]. And following us, no other facts, except some references such as [38], have ever been published concerning detailed implementation information for such Java code. Unfortunately, [38] can only be applied to construct interval arithmetic when the rounding modes toward $-\infty$ and $+\infty$ are available, while with only one rounding mode *near*, Java cannot satisfy such condition.

Definition 3.16. Let $I_1 = [a, b]$ and $I_2 = [c, d]$ be two intervals such that a, b, c and d are $\mathcal{F}$-numbers, then

$$\begin{aligned} I_1 + I_2 &= [(a +_{near} c)^-, (b +_{near} d)^+] \\ I_1 - I_2 &= [(a -_{near} d)^-, (b -_{near} c)^+]. \end{aligned}$$

We omit here interval extensions for the multiplication and division.

If the floating-point arithmetic in Java was implemented to satisfy a recommendation of IEEE 754 —the rounding of all arithmetic operations should be exact— then, the above Java interval extensions will ensure preserving all mathematical solutions. We have to show, for instance, for the addition that

$[a + c, b + d] \subseteq [(a +_{near} c)^-, (b +_{near} d)^+]$.

This is equivalent to proving

(1) $(a +_{near} c)^- \leq a + c$ and (2) $b + d \leq (b +_{near} d)^+$.

Proof. To prove (1), we consider the following two cases.

- If $a +_{near} c \leq a + c$, then we obtain (1) by default. Note that by our notations in Section 3.2, given a float f, f^- is the preceding float of f.
- If $a +_{near} c > a + c$, then we must have $(a +_{near} c)^- \leq a + c$. Suppose otherwise that $(a +_{near} c)^- > a + c$. Then, $a +_{near} c > (a +_{near} c)^- > a + c$. This is contradictory, because $+_{near}$ is exactly rounded. And hence, the above supposition is thus invalid.

The proof for (2) can be done by applying the same principle with that for (1). □

Note that the above proof is only valid if no overflow situations actually take place during the calculation of such interval extensions, as was illustrated in the previous subsection.

Of course, if the Java floating-point arithmetic is not exactly rounded, such Java interval extensions are thus unreliable in preserving all mathematical solutions. Let us take an example with the interval addition in Definition 3.16 to illustrate this claim. Suppose that f_1, f_2 and f_3 are three consecutive floats such that $f_1 < a + c < f_2 < f_3$, where $a + c$ is the left endpoint of the mathematical interval addition. If $+_{near}$ is not exactly rounded, then it is possible that $a +_{near} c = f_3$, from which $(a +_{near} c)^- = f_2$ (the computed left endpoint). Since $a + c < f_2$, we cannot preserve all mathematical solutions.

We can conclude that if only one rounding mode is used, it seems difficult to construct a reliable interval library, where one handles not only the arithmetic operations, but also analytic ones such as the *sin* function, etc. Because by the IEEE 754 standard, exact rounding is not assured, for example, for transcendental functions.

3.6 Conclusion

We presented in this chapter the background of our work. In Section 3.1, we gave the background related to test data generation, e.g. the control flow graph. Section 3.3 introduced some important notions such as CSP (constraint satisfaction problem), constraint solving (the resolution of CSP), as well as describing consistency techniques as underlying techniques in the solving of CSP. General notations were also given. The formal definitions of path constraints and test cases were then presented. We made a distinction between mathematical solutions of a path constraint and its test cases (float solutions of a path constraint with the boolean, integer and float operators of the programming language $\mathcal{L}$, used by the test program). In Section 3.4, we described a basis of the classical interval programming. We showed how to extend into intervals (obtaining thus interval extensions), the objects of the real space such as real numbers, real sets, real functions, and real relations. We then introduced the natural interval extension, which is used in our work for conserving float solutions in our constraint solving algorithms, as will be shown later. In Section 3.5, we were finally concerned with how to construct interval arithmetic in practice.

In the next chapter, we will present our new framework on interval logic developed so as to deal with our problem of test data generation. We show how to extend the classical definition of interval extension in order to build this framework.

Chapter 4

Extended Framework on Interval Logic

We here extend the classical definition of interval extension, and build a new interval logic framework to handle interval constraints involving at the same time, integer, float and boolean variables, as well as the logical operators such as AND, OR, NOT. Such a framework is needed for the following reasons. First, Definition 3.11 means that C is a mapping from I^n to the set $\{false, true\}$. Assuming C is an interval extension of relation c, let us denote $c(\mathbf{I}) = \{c(\mathbf{r}) \mid \mathbf{r} \in \mathbf{I}\}$ with $\mathbf{I} \in \mathcal{I}^n$. If $(\exists \mathbf{r} \in \mathbf{I})$ $c(\mathbf{r})$, then we have $\{C(\mathbf{I})\} \subseteq c(\mathbf{I})$. This leads to the following consequence: $[2,4] < [1,3]$ evaluates to $true$, from which one can deduce that $not([2,4] < [1,3])$ evaluates to $false$, whereas the negation can evaluate to $true$ if one treats $not([2,4] < [1,3])$ as $[2,4] \geq [1,3]$. This consequence is due to the fact that relations on intervals do not have a strict order, i.e. $[2,4] < [1,3]$ and $[1,3] < [2,4]$ are both true on intervals. Second, we need a framework in which we can define interval extensions for constraints involving boolean variables as well as the logical operators, such as $c(b,x,y) \triangleq not(b) \; and \; (x > 1) \; or \; (y < 2)$, where b is a boolean variable, while x, y are (integer or float) variables. We must then be able to evaluate $C(b : [0,1], x : [2,3], y : [3,5])$, for instance.

Definition 4.1 (Interval Extension of a Relation (constraint)). An interval relation $C : \mathcal{I}^n \to \mathcal{BI}$ is an interval extension of the relation $c : \mathcal{R}^n \to \mathcal{B}ool$ if $\forall \mathbf{I} \in \mathcal{I}^n : c(\mathbf{I}) \subseteq C(\mathbf{I})$, where $c(\mathbf{I}) = \{c(\mathbf{r}) \mid \mathbf{r} \in \mathbf{I}\}$, and the convention that $\{false\} = [0,0]$, $\{true\} = [1,1]$, and $\{false, true\} = [0,1]$.

Interval extensions for the relational operators $\{<, \leq, >, \geq, =, \neq\}$ are developed, based on the following definition.

Definition 4.2. Given a relation $c : \mathcal{R}^n \to \mathcal{B}ool$, the corresponding interval relation $C : \mathcal{I}^n \to \mathcal{BI}$ is constructed such that for all $\mathbf{I} \in I^n$

if $\nexists \mathbf{x} \in \mathbf{I} : c(\mathbf{x})$ then $C(\mathbf{I}) = [0,0]$
if $\exists \mathbf{x} \in \mathbf{I} : c(\mathbf{x}) \bigwedge \exists \mathbf{y} \in \mathbf{I} : \neg c(\mathbf{y})$ then $C(\mathbf{I}) = [0,1]$
if $\forall \mathbf{x} \in \mathbf{I} : c(\mathbf{x})$ then $C(\mathbf{I}) = [1,1]$

Proposition 4.1. The interval relation C, as defined in Definition 4.2, is an interval extension of the relation c.

Proof. We must prove the proposition with reference to Definition 4.1, namely for $\forall \mathbf{I} \in \mathcal{I}^n : c(\mathbf{I}) \subseteq C(\mathbf{I})$. Because we have to consider all the elements of any $\mathbf{I} \in \mathcal{I}^n$ in the definition of $C(\mathbf{I})$, and because of the convention that $\{false\} = [0,0]$, $\{true\} = [1,1]$ and $\{false, true\} = [0,1]$, it is easy to check that $c(\mathbf{I}) = C(\mathbf{I})$, from which $c(\mathbf{I}) \subseteq C(\mathbf{I})$ is proved. □

For instance, an interval extension of the relational operator $\leq$ is the following: $[a_1, a_2] \leq [b_1, b_2]$ is $[0,0]$ if $a_1 > b_2$, $[1,1]$ if $a_2 \leq b_1$, and $[0,1]$ otherwise. We now define interval extensions for the logical operators *not*, *and*, and *or*.

Definition 4.3. Let a_1, b_1, a_2, and b_2 be values taken in $\{0,1\}$ such that $a_1 \leq b_1$ and $a_2 \leq b_2$, then the interval logical operators NOT, AND, and OR are defined as follows

- $NOT([a_1, b_1]) = [1 - b_1, 1 - a_1]$,
- $[a_1, b_1]\ AND\ [a_2, b_2] = [min(a_1, a_2), min(b_1, b_2)]$,
- $[a_1, b_1]\ OR\ [a_2, b_2] = [max(a_1, a_2), max(b_1, b_2)]$.

Proposition 4.2. The interval logical operators NOT, AND, and OR, as defined in Definition 4.3, are respectively interval extensions of the logical operators *not*, *and*, and *or*.

Proof. The interval operator NOT can be seen as a mapping $NOT : \mathcal{BI} \to \mathcal{BI}$,
the interval operator AND as a mapping $AND : \mathcal{BI}^2 \to \mathcal{BI}$,
and the interval operator OR as a mapping $OR : \mathcal{BI}^2 \to \mathcal{BI}$.
We must prove the proposition with reference to Definition 4.1, namely for $\forall \mathbf{I} \in \mathcal{BI}^n$ (with $n = 1$ for NOT, $n = 2$ for AND and OR)

$$\begin{aligned} &not(\mathbf{I}) \subseteq NOT(\mathbf{I}) \\ &and(\mathbf{I}) \subseteq AND(\mathbf{I}) \\ &or(\mathbf{I}) \subseteq OR(\mathbf{I}) \end{aligned}$$

where $not(\mathbf{I}) = \{not(\mathbf{r}) \mid \mathbf{r} \in \mathbf{I}\}$ as usual, and the same for $and(\mathbf{I})$ and $or(\mathbf{I})$.

With the operator NOT, we have the following.
$NOT([0,0]) = [1-0, 1-0] = [1,1]$ by Def. 4.3, while $not([0,0]) = \{true\} = [1,1]$;
$NOT([0,1]) = [1-1, 1-0] = [0,1]$, while $not([0,1]) = \{false, true\} = [0,1]$;
$NOT([1,1]) = [1-1, 1-1] = [0,0]$, while $not([1,1]) = \{false\} = [0,0]$.
This means that for $\forall \mathbf{I} \in \mathcal{BI} : not(\mathbf{I}) = NOT(\mathbf{I})$, thereby $not(\mathbf{I}) \subseteq NOT(\mathbf{I})$ is assured.

With the operator AND, it should be noted first that if a_1 and a_2 are values taken in $\{0,1\}$, then $a_1\ and\ a_2 = min(a_1, a_2)$. Given the fact that

- $\forall v_1 \in [a_1, b_1], \forall v_2 \in [a_2, b_2] : min(a_1, a_2) \leq min(v_1, v_2) \leq min(b_1, b_2)$,
- $[a_1, b_1]\ and\ [a_2, b_2] = \{v_1\ and\ v_2 \mid v_1 \in [a_1, b_1], v_2 \in [a_2, b_2]\}$
  $= \{min(v_1, v_2) \mid v_1 \in [a_1, b_1], v_2 \in [a_2, b_2]\}$,

we can conclude that for $\forall \mathbf{I} \in \mathcal{BI}^2 : and(\mathbf{I}) \subseteq AND(\mathbf{I})$.

A proof for the operator OR follows the same principle as for the operator AND. □

An *interval solution* of a set of constraints is a box containing solutions of the different constraints. It is defined as follows.

Definition 4.4 (Interval Solution). Let $S = \{c_1, \ldots, c_m\}$ be a set of constraints. A box $\mathbf{X} \in \mathcal{I}^n$ is an interval solution of S if $right(C_i(\mathbf{X})) = 1$, for all i $(1 \leq i \leq m)$, where the C_i are respectively the natural interval extension of the c_i.

For simplicity, $C(\mathbf{X})$ will denote $right(C(\mathbf{X})) = 1$ (i.e. $C(\mathbf{X})$ is $[0, 1]$ or $[1, 1]$) and $\neg C(\mathbf{X})$ for $right(C(\mathbf{X})) = 0$ (i.e. $C(\mathbf{X})$ is $[0, 0]$).

Part II

The Consistency Approach

Chapter 5

Overview of the consistency approach

We are now able to precisely state our test data generation problem.

Problem statement *Given a node n, a branch b or a path p of the ICFG associated with a tested procedure P (possibly with procedure calls), generate a test input i such that P when executed on i will cause n, b or p to be traversed.*

This chapter describes the consistency approach for test data generation with path and statement coverage. It is a constraint solving approach based on a consistency notion, e-box consistency, generalizing box-consistency [36] to integer, boolean, and float variables.

5.1 Path coverage

Path coverage is the core of our approach. It includes the following steps.

1. A path constraint is derived from the specified path of the ICFG. Such a constraint involves integer, boolean and float variables, as well as operations with arrays.
2. The path constraint is solved by an interval-based constraint solving algorithm. The idea of such a solving algorithm is as follows.
   - An initial box is provided.
   - Consistency techniques are used to prune the box.
   - The box is split into some parts, which are then explored recursively until obtaining epsilon boxes —very small boxes— containing float solutions of the path constraint. These epsilon boxes are called interval solutions.
3. A test case is finally extracted from the interval solutions.

We now illustrate the operation of path coverage on the `nThRootBisect` program, given in Figure 5.1. The associated control flow graph of the `nThRootBisect` program is depicted in Figure 5.2, as well as the path 1-2-3-4-6-7-2-3-4-6-8 shown in dashed lines.

A path constraint is first generated step by step as follows:

1:	$l_0 = 1 \land h_0 = a_0$
2(True):	$\land\ (h_0 - l_0)^2 \geq e_0$
3:	$\land\ c_0 = (l_0 + h_0)/2$

```
float nThRootBisect(float a, int n, float e)
POST Let r > 0 such that r^n = a
     Return r* with (r* − r)^2  <  e

    float l, h, c;
    function float  f(float x) = (x**n - a) ;
1.  l = 1; h = a;
2.  while ((h − l)^2  >=  e) do
3.    c = (l + h)/2;
4.    if (f(c) = 0)
5.      return c;
6.    if (f(l)*f(c) < 0)
7.      h = c;
8.    else l = c;
9.  return h;
```

Figure 5.1: Program nThRootBisect (bisection method)

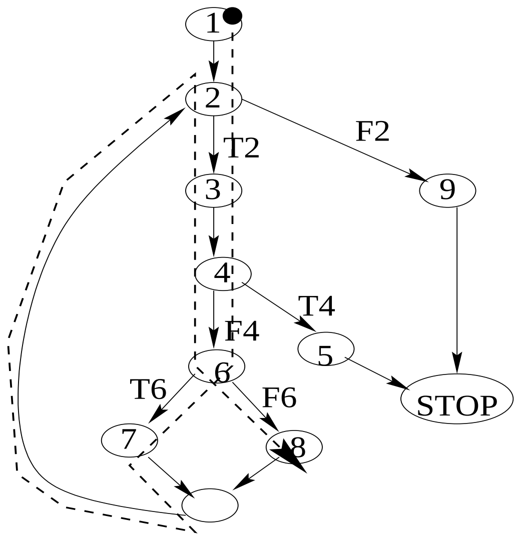


Figure 5.2: A path in the CFG of program nThRootBisect

4(False): $\wedge\ not(c_0^{n_0} - a_0 = 0)$
6(True): $\wedge\ (l_0^{n_0} - a_0) * (c_0^{n_0} - a_0) < 0$
7: $\wedge\ h_1 = c_0$
2(True): $\wedge\ (h_1 - l_0)^2 \geq e_0$
3: $\wedge\ c_1 = (l_0 + h_1)/2$
4(False): $\wedge\ not(c_1^{n_0} - a_0 = 0)$
6(False): $\wedge\ (l_0^{n_0} - a_0) * (c_1^{n_0} - a_0) < 0$
8: $\wedge\ l_1 = c_1$

The path constraint with an initial box ($a_0 \in [5, 20], n_0 \in [2, 20], e_0 \in [1e - 4, 1e - 2]$) forms a CSP, solved by our constraint solver to generate the test case ($a_0 = 7.000000010011718, n_0 = 2, e_0 = 0.00505$). Note that the input variables a_0, n_0 and e_0 represent here the input parameters of program `nThRootBisect`.

5.2 Statement coverage

For statement coverage, paths reaching the specified statement are dynamically constructed. The search for such paths is guided by the interprocedural control dependences of the program, as well as pruned by our e-box consistency filter to avoid exploring infeasible paths. Our algorithm for path coverage is then applied on these paths to generate test data.

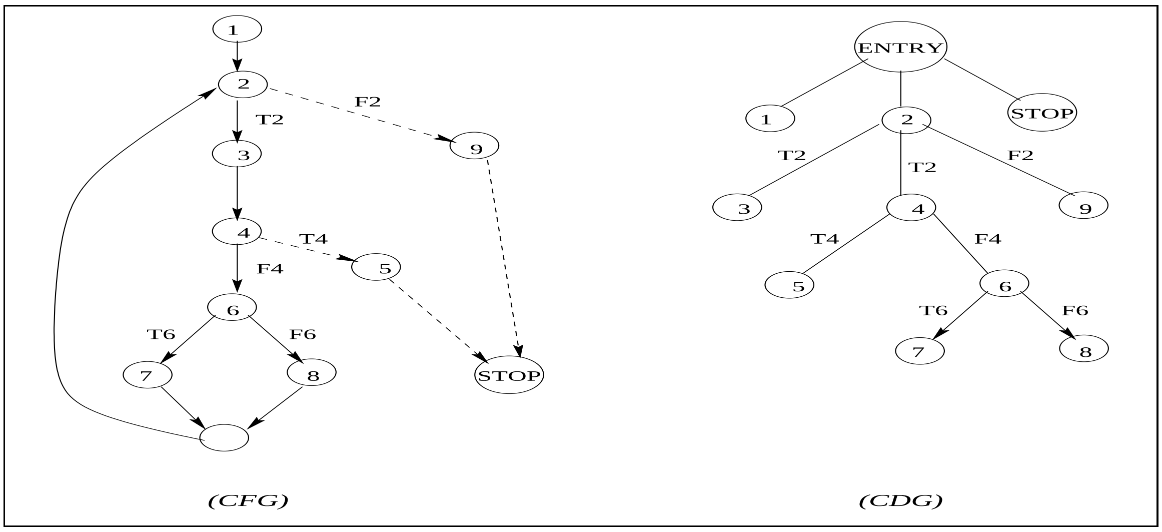


Figure 5.3: CFG and CDG of the nThRootBisect program

In the CFG of program `nThRootBisect`, shown in the left hand side of Figure 5.3, let us choose node 8. The reachability graph for node 8, depicted with solid edges, is first constructed. It is the smallest subgraph of the CFG, containing all the paths from the start node to node 8. The search is therefore carried out on the reachability graph. The search is guided by the control dependence graph (CDG) of program `nThRootBisect`, shown in the right hand side of Figure 5.3. The CDG captures the control dependences between nodes (see Definition 9.1 for *control dependence*). The idea is that nodes which must be reached in order to reach the specified node (the specified node is control-dependent on such nodes) will be considered first. For example, the control dependences for node 8 are <2-T2, 4-F4, 6-F6>. First, the path 1-2-3-4-6-8 will be constructed by the algorithm. Assuming the corresponding path constraint is inconsistent, the path 1-2-3-4-6-7-2-3-4-6-8 is next constructed. For node 5, the control dependences are <2-T2, 4-T4>. The path 1-2-3-4-5 will be first constructed, then path 1-2-3-4-6-7-2-3-4-5.

It should be noted that as a branch is dual to a statement in the control flow graph, all the above results with statement coverage can easily be extended to branch coverage. For example, to generate test data for a branch (a, b) where a and b are nodes, we must generate test data for paths reaching node a first and followed immediately by node b.

5.3 Specificities of our approach

It is important to precise the specificities of solving a path constraint compared to classical interval-based constraint solving.

- A path constraint is usually under-constrained; there usually exist many test inputs traversing the specified path (except for an infeasible path) while we are interested by finding one of them. Existing constraint systems, such as Numerica [36], are not always appropriate for under-constrained systems as they try to generate all the solutions.
- Existing solvers —Numerica, Prolog IV [3], and CLP(BNR) [2]— will produce (small) intervals containing the mathematical solutions of the path constraint. A mathematical solution can be a real which is not a float number. Moreover, even if a mathematical solution is a float number, this mathematical solution as test input is not guaranteed to traverse the specified path as the path constraint is executed using the programming language float operators, which are not mathematically sound.
- The goal of existing consistency techniques is to preserve all mathematical solutions in pruning the search space, and therefore may not ensure preserving all float solutions (solutions with the programming language operators) [50]. In contrast, the goal of our consistency techniques is to preserve float solutions. Our constraint solver in turn returns float solutions as test cases, and therefore ensure traversing the path. These differences make that existing constraint solving approaches cannot be used solely to generate test data for programs with integer, boolean, and float variables.
- It should be noticed that any constraints solving system may produce an interval containing neither float solutions nor mathematical solutions.

In the next chapter, we will present an algorithm for the construction of a path constraint from a specified path of the ICFG.

Chapter 6

Generation of Path Constraints

6.1 Algorithm

Given a path of an ICFG, we propose an algorithm (Algorithm 6.1) to construct a path constraint. Indexed variables are used to hold the definitions of the original variables in the path. Note that assignments to a variable are referred to as its *definitions*. For example, for variable x, its first definition in the path is assigned to x_0, its second to x_1, and so on. All uses of this variable are renamed accordingly and refer to its last definition. Since indexed variables have a unique definition, we will refer to them as *value instances* of the original variables.

Algorithm 6.1 Path constraint generation for a path in the ICFG

```
function PathConstraintGeneration($P$:$Procedure$,$G$:$ICFG$,$p$:$Path$) : $CSP$;
PRE $G$ is the ICFG for test procedure $P$
    $p$ is a path $p_1,\ldots,p_n$ in $G$
POST return a path constraint for path $p$
declare
  $PC$ : path constraint for path $p$
begin
  $PC$ := $\emptyset$; {$PC$ is initially empty}
  for each $i$ from 1 to $n$ do
    $PC$ := $PC$ $\wedge$ ConstraintsForNode($p_i$);
    if ($p_i$ is a decision node) and ($i$ $<$ $n$) then
      $PC$ := $PC$ $\wedge$ ConstraintsForBranch($< p_i, p_{i+1} >$);
  return $PC$;
end
```

The algorithm `PathConstraintGeneration` (Algorithm 6.1) takes as input a path in the ICFG. It makes a traversal of the path to generate constraints for its nodes and branches. The generated path constraint is the conjunction of all these constraints.

`ConstraintsForNode` and `ConstraintsForBranch` respectively generate constraints for a node and a branch of the ICFG. They will be described by the following definition.

Definition 6.1. Let p be a path of the ICFG, p_i be a node of p, and $< p_i, p_{i+1} >$ be a branch of p. Then $\mathcal{PC}(p, p_i)$ and $\mathcal{PC}(p, < p_i, p_{i+1} >)$ respectively denote constraints generated for p_i and $< p_i, p_{i+1} >$.

Let $\mathcal{PC}(p)$ denote the path constraint generated for the path p, then from Algorithm 6.1 and Definition 6.1, we have

$$\mathcal{PC}(p) = \bigwedge_{p_i\ :\ a\ node\ of\ p} \mathcal{PC}(p, p_i) \bigwedge_{<p_i,p_{i+1}>\ :\ a\ branch\ of\ p} \mathcal{PC}(p, < p_i, p_{i+1} >)$$

Depending on the type of p_i (which can be a global entry, an assignment, an input statement, a call node, etc.), $\mathcal{PC}(p, p_i)$ and $\mathcal{PC}(p, < p_i, p_{i+1} >)$ are constructed accordingly as follows.

6.1.1 Global Entry Node

p_i is the global entry node of the ICFG associated with a test procedure P. Suppose that P has the following parameters: x (a simple variable), a (an array variable). Then, $\mathcal{PC}(p, p_i)$ is

$$\mathrm{x}_0 \in dom(x) \wedge \bigwedge_{0 \le i < length(a)} \mathrm{a}_0[i] \in dom(a)$$

where $dom(x)$ denotes the domain for x, and $dom(a)$ the domain for all array elements of a. These constraints thus aim to define input variables from the formal parameters of procedure P.

Note that:

- The initial domain (interval) may depend on the programming language $\mathcal{L}$, but can also be fixed by the user.
- We focus our presentation to one-dimensional arrays, but the approach itself can be easily generalized to multi-dimensional arrays.
- We suppose that the size of array variable a is specified. If it is not the case, i.e. $length(a)$ is unknown, we note only that a_0 is an input variable, and hence no $\mathrm{a}_0[i]$ are created at this node. Later, when we deal with an $\mathrm{a}_0[i]$ (i is a number), and if no input variable representing $\mathrm{a}_0[i]$ exists, then an input variable $\mathrm{a}_0[i]$ is created once for all.

Definition 6.2. Let exp be an expression. Then $\overline{exp}$ denotes a version of exp in which each variable is substituted by its last value instance.

6.1.2 Assignment

- If p_i is an assignment to a simple variable, `x := exp`, then $\mathcal{PC}(p, p_i)$ is

$$x_k = \overline{exp}$$

where k is the smallest integer not yet used for identifier x. Note that this sort of equality constraints, associated with assignments, will be denoted as $x_k := \overline{exp}$.

- If p_i is an assignment to an array element, `a[j]:=exp`, then $\mathcal{PC}(p, p_i)$ is

$$\mathsf{na4}(a_{k'}, a_k, \overline{j}, \overline{exp})$$

where a_k is the last value instance of a; k' is the smallest integer not yet used for identifier a. The constraint na4 (na represents New Array) is defined hereafter.

Definition 6.3. The constraint $\mathsf{na4}(b, a, j, v)$ states that b is an array which is of the same size as a and has the same component values, except for v as the value of its j-th component. It can be defined more formally as follows.

$$\mathsf{na4}(b, a, j, v) \triangleq b[j] = v \bigwedge_{i \neq j} b[i] = a[i]$$

Definition 6.4. By convention, when all the elements of array a are *null* (non-initialized), we will denote this as $a = null$.

Example 6.1. Here is an example of constraint generation involving arrays:
Initial code: `int a[5]; a[0] = 8; a[1] = 7;`
Generated constraint: $\mathsf{na4}(\mathrm{a}_0, null, 0, 8) \wedge \mathsf{na4}(\mathrm{a}_1, \mathrm{a}_0, 1, 7)$.

6.1.3 Input Statement

- If p_i is an input statement to a simple variable, `read x` (in C, this is realized by `scanf`), then $\mathcal{PC}(p, p_i)$ is

$$x_k \in dom(x)$$

where k is the smallest integer not yet used for identifier x. This constraint defines x_k as an input variable.

- If p_i is an input statement to an array element, `read a[j]`, then $\mathcal{PC}(p, p_i)$ is

$$\mathsf{na3}(a_{k'}, a_k, \overline{j})$$

where a_k is the last value instance of a; k' is the smallest integer not yet used for identifier a. The constraint na3 is defined as follows.

Definition 6.5. The constraint $\mathsf{na3}(b, a, j)$ is formally defined as

$$\mathsf{na3}(b, a, j) \triangleq b[j] \in dom(b) \bigwedge_{i \neq j} b[i] = a[i]$$

The goal of this constraint is to define $b[j]$ as an input variable.

6.1.4 Decision Node

If p_i is a decision node, then $\mathcal{PC}(p, p_i)$ is empty.
However $\mathcal{PC}(p, < p_i, p_{i+1} >)$ is $\overline{c}$, where c is the condition associated with the branch $< p_i, p_{i+1} >$.

6.1.5 Procedure Call

If p_i is a call node to a procedure P (the control is going to pass to P), then for each pair (x', x) —where x' is an actual parameter of the call, and x is the corresponding formal parameter of P (that is a by-value or by-reference parameter)— an assignment $x := x'$ is generated. These assignments are converted into constraints, that are then affected to $\mathcal{PC}(p, p_i)$.

If p_i is the return node of a call to procedure P (the control just quits P), an assignment $x' := x$ is generated if x is a by-reference parameter. Moreover, if the return node is associated with $z := P(\dots)$, an assignment is also generated. All these assignments are converted into constraints, that are then affected to $\mathcal{PC}(p, p_i)$.

This way of handling parameters is commonly known as the *call by value-result* mode of parameter passing [18], where the parameters of the procedure are not directly bound to the variable's address. Rather, they have their own space within their scope, and the new values of the parameters are copied back into the caller's variables only when the procedure is terminated.

To illustrate the difference between the call by reference and the call by value-result modes, consider the following C code [18]:

```
void a(int *x, int *y) {
  *x = 1;
  *y = 2;
}

int t;
a(&t,&t);
```

Then with the call by value-result mode, the value of t after the call depends on the order of parameter copies when the call is finished; while with the call by reference mode, the value of t will always be 2. This problem is due to aliasing (i.e. if x and y refer to the same variable or address, then x and y are *aliased*). Since in this work, we rather focus our attention to the feasibility of applying our consistency approach for test data generation, the aliasing problem is left for future work.

6.2 Example

Example 6.2. We illustrate the operation of the algorithm on the path 1-2-3-4-5a-8-9-10a-16-17-18-20-10b-11a-16-17-18-20-11b-12-13a-21-22-23-24-13b-15-5b-6-4-7-25 (in Figure 3.3). Constraints are generated for the nodes as follows.

Node 1: $\bigwedge_{0\leq i<10} \mathrm{a}_0[i] \in dom(a) \wedge \mathrm{c}_0 \in dom(c)$; ($a$ and c are the parameters of procedure `M` in Figure 3.1. The constraint defines thus input variables.)
Node 2: no constraints are generated;
Node 3: $\mathrm{i}_0 := 1$;
Node 4-$T$4: $\mathrm{i}_0 \leq \mathrm{c}_0$;
Node 5a: $\mathrm{a}_1 := \mathrm{a}_0$;
Node 8: no constraints;
Node 9: $\mathrm{i}_1 \in dom(i) \wedge \mathrm{j}_0 \in dom(j)$;
Node 10a: $\mathrm{i}_2 := \mathrm{i}_1$;
Nodes 16, 17-$T$17, 18, 20: $\mathrm{i}_2 \geq 0 \wedge \mathrm{i}_2 \leq 9$;
Node 10b: $\mathrm{fi}_0 := \mathrm{i}_2$;
Node 11a: $\mathrm{i}_3 := \mathrm{j}_0$;
Nodes 16, 17-$T$17, 18, 20: $\mathrm{i}_3 \geq 0 \wedge \mathrm{i}_3 \leq 9$;
Node 11b: $\mathrm{fj}_0 := \mathrm{i}_3$; Node 12-$T$12: $\mathrm{fi}_0 < \mathrm{fj}_0$;
Node 13a: $\mathrm{x}_0 := \mathrm{a}_1[\mathrm{i}_1] \wedge \mathrm{y}_0 := \mathrm{a}_1[\mathrm{j}_0]$;
Nodes 21, 22-$T$22, 23, 24: $\mathrm{x}_0 > \mathrm{y}_0 \wedge \mathrm{t}_0 := \mathrm{x}_0 \wedge \mathrm{x}_1 := \mathrm{y}_0 \wedge \mathrm{y}_1 := \mathrm{t}_0$;
Node 13b: $\mathsf{na4}(\mathrm{a}_2, \mathrm{a}_1, \mathrm{i}_1, \mathrm{x}_1) \wedge \mathsf{na4}(\mathrm{a}_3, \mathrm{a}_2, \mathrm{j}_0, \mathrm{y}_1)$;
Nodes 15, 5b: $\mathrm{a}_4 := \mathrm{a}_3$;
Nodes 6, 4-$F$4, 7, 25: $\mathrm{i}_4 := \mathrm{i}_0 + 1 \wedge \neg(\mathrm{i}_4 \leq \mathrm{c}_0)$.

6.3 Analysis

A path constraint is composed of:

1. constraints defining input variables: simple input variable and input array element (`na3` constraints),
2. assignment constraints: equality constraints with ":=" notation for simple variables, and `na4` constraints for array elements,
3. branch constraints (constraints for the branches of the path).

However, only the branch constraints (3) represent the conditions which must be satisfied so that the path is traversed. The other types of constraints (1) and (2), as will be shown later, are used in the simplification of the branch constraints in terms of input variables. The solving of the path constraint is the solving of its branch constraints. In the CSP associated with a path constraint, only the input variables will have a domain. There is no need to define a domain for the other variables as they are defined in terms of input variables or constraints. If it is not the case, the program is referring to non-initialized variables, and is thus incorrect.

Proposition 6.1. The constraint generated by Algorithm 6.1 is a path constraint.

Proof. We only present here a sketch of the proof.

The path can be seen as a program (called *path program*) if we replace every condition c_i of the path by an assignment $b_i := c_i$, where b_i is a boolean. A test input $\mathbf{v}$ will traverse the path if after executing the path program, all the b_i will become true. To prove that the constraint generated by Algorithm 6.1 —as denoted above by $\mathcal{PC}(p)$— is a path constraint, we must show that $\bigwedge b_i$ is equivalent to $\mathcal{PC}(p)$.

As the above abstraction of the algorithm (namely, the idea of using indexed variables) follows the same principle as the Static Single Assignment (*SSA*) [8] (which is an equivalent representation of a program), we can conclude that the generated constraint $\mathcal{PC}(p)$ is actually equivalent to the path program. In other words, $\mathcal{PC}(p)$ is a path constraint. This means that if a test input **v** traverses the path p, then the constraint $\mathcal{PC}(p)$ (evaluated by using the operators of the programming language $\mathcal{L}$) will be satisfied by that input. $\square$

Note that in an SSA form, there is only one assignment to each variable in the entire program, and each use of a variable refers to only one assignment. Thanks to this form, one can reason easily about variables because if two variables have the same name, then they contain the same value wherever they occur in the program. Here is an example of a simple sequence of assignments and its corresponding SSA form :

Original form : `x = 0; y = x+1; x = x+y; y = x+y;`
SSA form : $\texttt{x}_1$ `= 0;` $\texttt{y}_1$ `=` $\texttt{x}_1$`+1;` $\texttt{x}_2$ `=` $\texttt{x}_1$`+`$\texttt{y}_1$`;` $\texttt{y}_2$ `=` $\texttt{x}_2$`+`$\texttt{y}_1$`;`

Our constraints dealing with arrays such as na3 and na4 constraints, are also inspired from SSA form. Indeed, SSA form provides a special expression, among others, to handle arrays: $update(a, j, w)$, which evaluates to an array that has the same size and the same elements as a, except for the j-th element where the value is w.

In another work [26], also based on SSA form, a definition statement $a_1 = update(a_0, j, w)$ is translated into

$$\mathsf{element}(J, A_1, W) \bigwedge_{I \neq J} (\mathsf{element}(I, A_0, V) \wedge \mathsf{element}(I, A_1, V))$$

where the constraint $\mathsf{element}(I, L, V)$ expresses that V is the I-th element in the list L. Note that in [26], the initial program is first transformed into an SSA form, and constraints reaching a node (statement) are then constructed from this form; while in our work, given a specified path, we rather make a traversal of the path to construct directly an SSA-form-like path constraint.

6.4 Conclusion

In this chapter, we first described an algorithm for constructing a path constraint from a given path of the ICFG. The output path constraint is actually the conjunction of the constraints generated for the nodes and branches of the path. For each type of nodes (global entry, assignment, ...), we showed in detail how specific constraints are generated. Special constraints dealing with arrays, na3 and na4, were introduced. How these constraints are actually treated will be discussed in a following chapter. We then illustrated the working of the algorithm on an example. Finally, we informally analyzed its correctness.

In the next chapter, we will talk about our consistency technique and the basis for its conservation of float solutions.

Chapter 7

A New Consistency Technique

This chapter presents a new consistency technique, the core of our constraint solver developed for the test data generation problem. The ideas underlying the conservation of float solutions in our filtering algorithms will also be highlighted. We first define a consistency notion, called *e-box consistency.*

7.1 Consistency

We introduced e-box consistency in [63] as an extension of the classical box-consistency [36] to integer, boolean, and float variables. The objective is to reduce the domains of the variables (i.e. their intervals) without removing solutions.

Definition 7.1 (e-box consistency). Let $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ be a *CSP* where $\mathcal{V} = (x_1, \dots, x_n)$, a set of (float and integer) variables; $\mathcal{D} = (X_1, \dots, X_n)$ with $X_i = [l_i, r_i]$ the domain of x_i $(1 \leq i \leq n)$; $\mathcal{C} = (c_1, \dots, c_m)$, a set of constraints defined on $x_1, \dots, x_n$ and $c \in \mathcal{C}$ be a k-ary constraint on the variables $(x_1, \dots, x_k)$. The constraint c is *e-box consistent* in $\mathcal{D}$ if for all x_i $(1 \leq i \leq k)$

- $C(X_1, \dots, X_{i-1}, [l_i, l_i], X_{i+1}, \dots, X_k) \bigwedge$
  $C(X_1, \dots, X_{i-1}, [r_i, r_i], X_{i+1}, \dots, X_k)$ when $l_i \neq r_i$
- $C(X_1, \dots, X_{i-1}, [l_i, r_i], X_{i+1}, \dots, X_k)$ when $l_i = r_i$

where C is the natural interval extension of constraint c.

The CSP P is *e-box consistent* in $\mathcal{D}$ if for all $c \in \mathcal{C}$, c is e-box consistent in $\mathcal{D}$.

Note that in the definition of the classical box-consistency, the above two bullets are replaced by the following:

- $C(X_1, \dots, X_{i-1}, [l_i, l_i^+], X_{i+1}, \dots, X_k) \bigwedge$
  $C(X_1, \dots, X_{i-1}, [r_i^-, r_i], X_{i+1}, \dots, X_k)$ when $l_i \neq r_i$
- $C(X_1, \dots, X_{i-1}, [l_i, r_i], X_{i+1}, \dots, X_k)$ when $l_i = r_i$

where C is *any* interval extension of constraint c.

The latter definition states the potential existence of *real solutions* in small boxes ($[l_i, l_i^+]$ and $[r_i^-, r_i]$) on all sides of the domains, whereas Figure 7.1 describes the intuition of our e-box consistency where small boxes on all sides of the domains may contain *float solutions.*

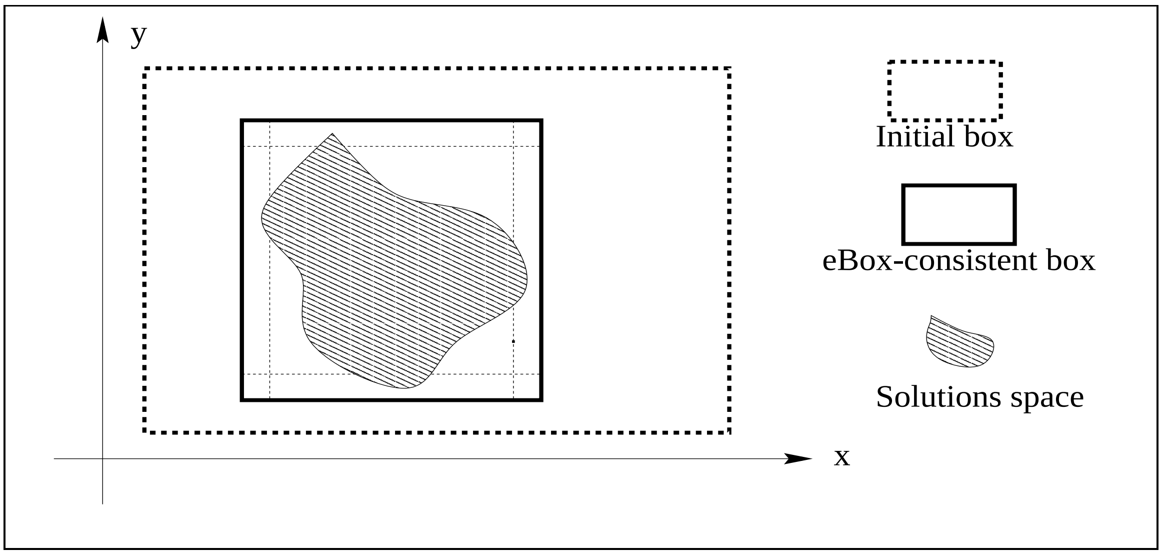


Figure 7.1: Example of e-box consistency

Given the initial domains of the variables (the initial box), the purpose of filtering is to obtain the smallest box, satisfying the e-box consistency, and without removing any solutions from the initial box. In constraint programming, one finds a lot of sophisticated consistencies dealing with real solutions, such as used in Prolog IV, CLP(BNR), or Numerica. A consistency dealing with float solutions was also proposed in [50], where it concentrates only on float variables.

Definition 7.2 (Filtering by e-box consistency). Filtering by e-box consistency of a CSP $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is a CSP $P' = (\mathcal{V}, \mathcal{D}', \mathcal{C})$ such that (1) $\mathcal{D}' \subseteq \mathcal{D}$, (2) P and P' have the same float solutions, and (3) P' is e-box consistent in $\mathcal{D}'$.

Our filtering algorithm is based on the property that if $C(\mathbf{X})$ does not hold, i.e. $right(C(\mathbf{X})) = 0$, then no solution of c lies in box $\mathbf{X}$, that can then be pruned. We denote by $\Phi_{e-box}(P)$, the filtering by e-box consistency of P. Note that the filtering by e-box consistency of a CSP, by its definition, always exists and is unique. An implementation of $\Phi_{e-box}(CSP)$, called `PhiEBox`, is presented in Algorithm 7.1, that uses standard mechanisms of pruning such as in [36, 15]. Therefore, the use of an open solver may facilitate the implementation of this algorithm.

We now detail the operation of `PhiEBox`. A *queue* is first initialized (line 1). It contains all the constraints of the input CSP. The objective of the main loop in line 2 is to make a filtering operation on each constraint until the e-box-consistent CSP is obtained (line 10). The filtering operation for each constraint c (extracted from *queue* in line 3) in turn is realized by the loop in line 4, which consists in a filtering operation on each pair $< c, x_i >$. For each variable x_i of c —*variables(c)*, in line 4, denotes all variables involved in c— an univariate interval constraint on X, C_X, is generated (line 5) by replacing all variables of c, except x_i, by the intervals corresponding to their domains. By convention, variable x_i of c is also replaced by interval variable X in interval constraint C_X. The leftmost and rightmost zero canonical intervals of X_i (the domain for variable x_i) are then computed respectively by the functions `LeftNarrow` and `RightNarrow` (line 6). The leftmost (rightmost) "zero" canonical interval is the leftmost (rightmost) canonical interval I such that $C_X(I)$.

Algorithm 7.1 Φ_{e-box}

```
function PhiEBox($V$ : $Variables$, $D$ : $Box$, $C$ : $Constraints$) : $CSP$;
PRE
  $V$ a set of variables
  $D$ a box of their corresponding domains
  $C$ a set of constraints over $V$
POST
  Return a CSP $(V,D',C)$ such that $(V,D',C) = \Phi_{e-box}(V,D,C)$
begin
1: queue := $C$;
2: while queue $\neq \emptyset$ do
3:    c := dequeue(queue); {Suppose c is a constraint over $x_1,\ldots,x_k$}
      updatedDomVars := $\emptyset$; {A set of variables with domain updated}
4:    for $x_i \in$ variables(c) do
5:       $C_X$ := C($X_1,\ldots,X_{i-1},X,X_{i+1},\ldots,X_k$);{univariate interval constraint}
6:       $left(X_i')$ := $left$(LeftNarrow($C_X$, $X_i$));
         $right(X_i')$ := $right$(RightNarrow($C_X, X_i$));
         if $X_i \neq X_i'$ then
7:          $X_i$ := $X_i'$;
8:          if $X_i = \emptyset$ then return $(V,\emptyset,C)$;
            updatedDomVars := updatedDomVars $\bigcup$ {$x_i$};
      endfor
9:    queue := queue $\bigcup$ {c' $\in$ C | updatedDomVars $\bigcap$ variables(c') $\neq \emptyset$};
   endwhile
10:return $(V,D,C)$;
end
```

Only function `LeftNarrow` is described in Algorithm 7.2. Function `RightNarrow` can be derived easily from `LeftNarrow` as a symmetric version.

The new domain for x_i will be X_i' (line 7) if $X_i \neq X_i'$. If the new domain for x_i is empty, an inconsistent CSP is returned (line 8). Following the pruning for constraint c, we add into *queue* constraints which can be pruned further (line 9). When we reach line 10, no constraint is left to be pruned. In other words, all the constraints from C have gone through filtering, and no further domain reductions can be realized — i.e., a fixpoint is reached. We obtain thus the e-box-consistent CSP.

We now go into details for Algorithm 7.2 (`LeftNarrow`). If C_X is satisfied on interval I, two sub-cases (corresponding to line 2 and line 3) are possible; otherwise an empty interval (denoted by $\emptyset$) is returned (line 9). If I is canonical (line 2), it will be returned as the leftmost zero canonical interval. If I is not canonical (line 3), it will be divided into two sub-intervals that are then explored recursively. The division of an interval into two parts is different following X is an integer variable (line 4) or float variable (line 5). If the exploration of I_{left} (line 6) is successful, the result is returned (line 7). Otherwise, the result of exploring I_{right} is returned (line 8).

Algorithm 7.2 Finding the leftmost zero canonical interval

```
function LeftNarrow($C_X$ : $IntervalConstraint$, $I$ : $Interval$) : $Interval$;
PRE
  $C_X$ an univariate interval constraint on $X$
  $I$ an interval
POST
  Return the leftmost zero canonical interval $L \in I$ such that $C_X(L)$
  if it exists; otherwise return $\emptyset$
begin
1: if $C_X(I)$ then
2:   if $I$ is canonical then return $I$
3:   else
       $m := (left(I) + right(I))/2$;
4:     if $X$ is an integer variable then
         $I_{left} := [left(I)\ ,\ \lfloor m \rfloor]$; $I_{right} := [\lceil m \rceil\ ,\ right(I)]$;
5:     else
         $I_{left} := [left(I)\ ,\ m]$; $I_{right} := [m\ ,\ right(I)]$;
6:     $L$ := LeftNarrow($C_X$, $I_{left}$);
7:     if $L \neq \emptyset$ then return $L$;
8:     else return LeftNarrow($C_X$, $I_{right}$);
9: else return $\emptyset$;
end
```

Analysis

Technically, our `LeftNarrow` algorithm (Algorithm 7.2) is simpler than a standard one [15] for the classical box-consistency [36], but safe for float solutions. Our algorithm consists in applying recursively a domain-splitting on the initial interval to obtain the leftmost zero canonical interval. On the other hand, the standard `LeftNarrow` recursively applies two operations, an iterator of Newton and a domain-splitting on the initial interval. Because of the use of the Taylor interval extension, and of the fact that the iterator of Newton aims to prune parts —which do not have mathematical solutions— to make the algorithm converge more quickly, it may not be safe for float solutions as shown in [50].

The correctness of `LeftNarrow` follows directly if all the primitive interval operations and relations are monotonic. For example, an interval relation C is *monotonic* if $\forall \mathbf{X}_1, \mathbf{X}_2 \in \mathcal{I}^n \ :\ \mathbf{X}_1 \subseteq \mathbf{X}_2 \Rightarrow C(\mathbf{X}_1) \subseteq C(\mathbf{X}_2)$. As natural interval extensions are used in our work, and they are known to be monotonic [36], the assumption is naturally satisfied. With `LeftNarrow`, we see that if a sub-interval $I' \subseteq I$ is refuted, i.e. $\neg C_X(I')$, then from the monotonicity of C_X, we can deduce that there exists no zero canonical interval in I'. Combined with the fact that `LeftNarrow` proceeds from the left first, the returned canonical interval, if it exists, must be the leftmost zero canonical interval.

Note that for efficiency reasons, it is common practice to work with epsilon intervals rather than with canonical intervals in the definition of the e-box consistency, as well as in the functions `LeftNarrow` and `RightNarrow`. The efficiency of the filtering algorithm is therefore influenced by the chosen epsilon. The choice of such a value is analyzed in the experimentation chapter.

The correctness of `PhiEBox` (Algorithm 7.1) must be verified in reference to Definition 7.2. Remark first that `PhiEBox` can be viewed as applying repeatedly a filtering operation on each pair $< constraint, variable >$ until a fixpoint is reached. Point (1) of Definition 7.2 is evident. Point (3) follows from the fact that a filtering operation on each pair $< constraint, variable >$ by `LeftNarrow` and `RightNarrow` makes that the pair $< constraint, variable >$ is e-box consistent following Definition 7.1. Point (2) concerns the conservation of float solutions by `LeftNarrow` and `RightNarrow`, which will be the subject of the next section. Since `PhiEBox` converges to a fixpoint, it returns either the e-box-consistent CSP (because all the pairs $< constraint, variable >$ are e-box consistent at the fixpoint), or an inconsistent CSP (if some inconsistency is detected).

7.2 Conservation of Float Solutions

Since interval libraries are traditionally constructed to preserve mathematical solutions, it may be that float solutions will not be preserved when using such libraries. As introduced in Chapter 3, our aim is to obtain float solutions. We describe here the core results related to the conservation of float solutions in our filtering algorithms. We first present the following proposition given in [50], which is the basis for the conservation of float solutions.

Proposition 7.1. Assuming every basic operation has an optimal interval extension, if $F : \mathcal{I}^n \rightarrow \mathcal{I}$ is the *natural interval extension* of a real expression $f : \mathcal{R}^n \rightarrow \mathcal{R}$, then for all rounding mode $r \in \{+\infty, -\infty, 0, near\}$ and $\forall \mathbf{I} \in \mathcal{I}^n$, we have $f_r(\mathbf{I}) \subseteq F(\mathbf{I})$,
where f_r is a corresponding float expression of f, and $f_r(\mathbf{I}) = \{f_r(\mathbf{v}) \mid \mathbf{v} \in \mathbf{I} \textit{ and } \mathbf{v} \in \mathcal{F}^n\}$.
Note that for all $r \in \{+\infty, -\infty, 0, near\}$, $f_{-\infty}(\mathbf{v}) \leq f_r(\mathbf{v}) \leq f_{+\infty}(\mathbf{v})$.

This proposition states that natural interval extensions conserve float solutions, when all the basic operations have an optimal interval extension. Conservation of float solutions here is reflected by the fact that for whatever rounding mode r being used, interval evaluation on interval $\mathbf{I}$, $F(\mathbf{I})$, contains at least all float solutions of $f_r(\mathbf{I})$.

A sketch of the proof for Proposition 7.1 is presented in [50], which is based on the following observations.

- If f is a basic operation with an optimal interval extension, we then have:

$$\forall \mathbf{I} \in \mathcal{I}^n, F(\mathbf{I}) = [min(f_{-\infty}(\mathbf{I})), max(f_{+\infty}(\mathbf{I}))] \tag{7.1}$$

  Property (7.1) is also stated in [38] for arithmetic operators. It means that $left(F(\mathbf{I})) \leq f_r(\mathbf{v}) \leq right(F(\mathbf{I}))$ for all float $\mathbf{v} \in \mathbf{I}$ and for any rounding mode r.
- If f is a composition of basic operations, the proposition still holds by induction. Indeed, since F is the natural interval extension of f, we can assume that $f_r(\mathbf{v})$ and $F(\mathbf{I})$ are computed by evaluating the same sequence of basic operations.

Proposition 7.1 requires the interval extension of the basic operations to be optimal, what may not be the case for some basic operations (e.g. *sin*, *ln*, etc.). The following property extends Property (7.1), when the interval extensions of some basic operations are not optimal.

$$\forall \mathbf{I} \in \mathcal{I}^n, [min(f_{-\infty}(\mathbf{I})), max(f_{+\infty}(\mathbf{I}))] \subseteq F(\mathbf{I}) \tag{7.2}$$

If Property (7.2) is satisfied by the non-optimal basic operations, then the interval extension F conserves float solutions, because $f_r(\mathbf{I}) \subseteq [min(f_{-\infty}(\mathbf{I})), max(f_{+\infty}(\mathbf{I}))]$, whatever rounding mode r being used.

Note that ideally we expect to have Property (7.1) for all basic operations. But since calculating $min(f_{-\infty}(\mathbf{I}))$ and $max(f_{+\infty}(\mathbf{I}))$ can be too difficult for certain basic operations, due to the reasons given later in Section 10.11.5 (e.g. the monotony of a real operation f cannot ensure that $f_{-\infty}$ is monotone), we have to guarantee at least Property (7.2) so that float solutions are preserved.

Conservation of float solutions in Java In practice, it is sometimes difficult to have Property (7.2) for basic operations when only the rounding mode *near* is used such as in Java. However, if Properties (7.3) and (7.4) below are satisfied, then one can show that float solutions are also preserved. Because for any $\mathbf{I} \in \mathcal{I}^n$, $[min(f_{-\infty}(\mathbf{I})), max(f_{+\infty}(\mathbf{I}))] \subseteq [min(f^-_{near}(\mathbf{I})), max(f^+_{near}(\mathbf{I}))] \subseteq F(\mathbf{I})$.

$$\forall \mathbf{I} \in \mathcal{I}^n, [min(f^-_{near}(\mathbf{I})), max(f^+_{near}(\mathbf{I}))] \subseteq F(\mathbf{I}) \tag{7.3}$$

where $f^-_{near}(\mathbf{I}) = \{(f_{near}(\mathbf{v}))^- \mid \mathbf{v} \in \mathbf{I} \text{ and } \mathbf{v} \in \mathcal{F}^n\}$ and $f^+_{near}(\mathbf{I}) = \{(f_{near}(\mathbf{v}))^+ \mid \mathbf{v} \in \mathbf{I} \text{ and } \mathbf{v} \in \mathcal{F}^n\}$.

$$\forall \mathbf{v} \in \mathcal{F}^n, (f_{near}(\mathbf{v}))^- \leq f_{-\infty}(\mathbf{v}) \leq f_{near}(\mathbf{v}) \leq f_{+\infty}(\mathbf{v}) \leq (f_{near}(\mathbf{v}))^+ \tag{7.4}$$

Remark If any interval library in Java is implemented such that Property (7.3) and Property (7.4) hold for all of its basic operations, then conservation of float solutions is ensured by that library. Unfortunately, Property (7.4) is only correct if the rounding of an operation is exact. We illustrate this claim by an example. Recall that if $f(x)$ is a real operation such that $f_1 < f(x) < f_2$ —where f_1 and f_2 are two consecutive floats— then the rounding of f is exact if $f_{-\infty}(x) = f_1$ and $f_{+\infty}(x) = f_2$. If the rounding of f is not exact, it is possible that $f_{-\infty}(x) = f_1$ and $f_{+\infty}(x) = f_3$, where f_1, f_2 and f_3 are three consecutive floats. Note that $f_{near}(x)$ is either $f_{-\infty}(x)$ or $f_{+\infty}(x)$, depending on which of these is closer to $f(x)$ than the other.

- If $f_{near}(x) = f_1$, then $(f_{near}(x))^+ = f_2 < f_{+\infty}(x)$.
- If $f_{near}(x) = f_3$, then $(f_{near}(x))^- = f_2 > f_{-\infty}(x)$.

In all of these cases, Property (7.4) does not hold.

It is known that arithmetic operations are exactly rounded [50]. However, also following [50], Intel-387 provides transcendental functions with up to 4.5 *ulps* error, where an *ulps* corresponds to the size of the gap between two consecutive float numbers. As a consequence, if transcendental functions provided by Java are not exactly rounded, we cannot state whether Property (7.4) still holds or not.

In practice, since interval libraries are rather constructed to preserve mathematical solutions, and these interval libraries are then used in constraint solving to find mathematical solutions, it can be that Property (7.2) will not always hold. Therefore, one must construct an interval library satisfying Property (7.2), if the aim is to preserve all float solutions. Using such an interval library, our filtering algorithms will ensure preserving all float solutions. Because the functions `LeftNarrow` (Algorithm 7.2) and `RightNarrow` —the heart of our filtering algorithms— are based on the property that if $C(I)$ does not hold, where C is the natural interval extension of a constraint c, then no float solution lies in I, that can then be safely pruned.

Note that Property (7.2) is only necessary for the general framework on the floating-point numbers such as proposed by the IEEE 754 standard. Because for whatever rounding mode being used, float solutions with that rounding mode are preserved by interval operations. It is actually the case, when the programming language under test $\mathcal{L}$ is the C language. However, if the programming language under test uses only one rounding mode such as in Java, and the purpose is to generate test data for Java, one can easily observe that Property (7.3) is already sufficient to preserve all float solutions in Java. Because for any $\mathbf{I} \in \mathcal{I}^n$,
$f_{near}(\mathbf{I}) \subseteq [min(f^-_{near}(\mathbf{I})), max(f^+_{near}(\mathbf{I}))] \subseteq F(\mathbf{I})$.

In summary, using an interval library —where the basic interval operations are conservative on the floats— and the natural interval extensions for all real expressions, provides a safe way to preserve all float solutions.

7.3 Conclusion

In this chapter, we presented a new consistency technique for test data generation, the e-box consistency, as well as the basis for its conservation of float solutions. We defined, in Section 7.1, the e-box consistency. Such consistency aims at reducing as much as possible the domains of the variables without removing any solution of a CSP. We then gave detailed algorithms to achieve consistency for the CSP. In Section 7.2, we presented an important proposition, which is the basis for the conservation of float solutions in [50], as well as in our filtering algorithms. We then proposed a general way to implement any interval library such that its basic operations are conservative on the floats. Using such an interval library, our filtering algorithms will ensure the conservation of all float solutions.

In the next chapter, we will talk about test data generation for path coverage, more particularly the solving of the path constraint. We will show how our consistency algorithms are integrated in a constraint solver to generate test data.

Chapter 8

Test Data Generation for Path Coverage

This chapter describes constraint-solving algorithms for test data generation under the path coverage criteria (searching for test data executing a path of the ICFG). Our algorithms are divided into two sections corresponding to two cases, one where arrays are not involved in the test program (simple case) and one with the presence of arrays (general case). Our purpose is that readers can read only the simple version, without any need to read the general version where we discuss, among others, how constraints with arrays are actually handled.

8.1 Simple algorithm

Our algorithm for path coverage (given in Algorithm 8.1) includes the following steps.

1. A path constraint is derived from the specified path of the ICFG (by Function `PathConstraintGeneration` given in Chapter 6). Such a constraint involves integer, boolean and float variables, as well as operations with arrays. Note that the branch constraints BC mean constraints generated for the branches of the path. They represent the conditions which must be satisfied so that the path is traversed. The other types of constraints (OC) are used in the simplification of the branch constraints in terms of input variables. In this simple case (no presence of arrays), the branch constraints on the path can always be expressed in terms of input variables, because they are initially the constraints on program variables and those variables depend on input variables with assignment constraints along the path. This means that the solving of the path constraint (in the next step) is in fact the solving of BC', a simplified version of BC where all the branch constraints are expressed in terms of input variables. Note that for every non-input variable x, there must exists an assignment constraint, $x := def(x)$, in OC. The replacement of a non-input variable by its definition is only carried out in a "symbolic" manner. We illustrate our idea hereafter. Assuming x appears in a data structure DS_1, and $def(x)$ is represented by a structure DS_2, then a

Algorithm 8.1 Generation of test data for path coverage (simple version)

```
function TestDataGenPC($P$:$Procedure$,$G$:$ICFG$,$p$:$Path$):$\mathcal{F}^n$;
PRE $G$ the ICFG for test procedure $P$
    $p$ a path in $G$
POST a test case on which the path $p$ is executed
begin
  $PC$:= PathConstraintGeneration($P$,$G$,$p$);
  $BC$ := the branch constraints of $PC$;
  $OC$ := $PC$ \ $BC$;
  $BC'$ := a version of $BC$, simplified from $OC$ and expressed in terms of input vars;
  $V$ := the input variables in $BC'$;
  $D$ := the domains of the variables in $V$;
  return SolvePathConstraints($V$, $V$, $D$, $BC'$);
end

function SolvePathConstraints($V, V'$:$Variables$,$D$:$Box$,$BC$:$BranchConstraints$):$F^n$;
PRE $V$ the input variables in $BC$
    $V'$ a subset of $V$ ($V' \subseteq V$)
    $D$ a box representing the domains of the variables in $V$
POST Return some float solution $\mathbf{v} \in D$ of $BC$
     Otherwise it returns $\emptyset$
begin
  $qs$ := QuickFindSolution($BC$, $D$);        // Appendix A will make reference
  if $qs \neq \emptyset$ then return $qs$;                // to these 2 lines
  ($V$, $D_t$, $BC$) := PhiEBox($V$, $D$, $BC$);
  if $D_t$ is $\emptyset$ then return $\emptyset$;
  else
    if $D_t$ is an $\epsilon\_box$ then return FindSolution($BC$,$D_t$);
    else
      if $V'$ is not empty then
        Choose arbitrarily a variable $x$ in $V'$;
        $m$ := $(left(X_t) + right(X_t))/2$;
        if $x$ is an integer variable then
          $ms$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_t[X_t/[\lfloor m \rfloor, \lfloor m \rfloor]]$, $BC$);
        else $ms$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_t[X_t/[m, m]]$, $BC$);
        if $ms \neq \emptyset$ then return $ms$;
        if $x$ is an integer variable then
          $ls$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_t[X_t/[left(X_t), \lfloor m \rfloor - 1]]$, $BC$);
        else $ls$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_t[X_t/[left(X_t), m^-]]$, $BC$);
        if $ls \neq \emptyset$ then return $ls$;
        if $x$ is an integer variable then
          $rs$ :=SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_t[X_t/[\lfloor m \rfloor + 1, right(X_t)]]$, $BC$);
        else $rs$ := SolvePathConstraints($V$, $V' \setminus \{x\}$, $D_t[X_t/[m^+, right(X_t)]]$, $BC$);
        if $rs \neq \emptyset$ then return $rs$ else return $\emptyset$;
      else return SolvePathConstraints($V, V, D_t, BC$);
end
```

link is established between DS_1 and DS_2. Therefore, we are not dealing with important-sized expressions as resulted from an actual simplification.

2. The path constraint (represented by BC') is solved by an interval-based constraint solving algorithm (Function `SolvePathConstraints`). If function `QuickFindSolution` (specified in Specification 8.1) gives a test case, that is then returned. This step is theoretically unnecessary, but it may quickly find a solution of an under-constrained path constraint. Otherwise, function `PhiEBox` (given earlier in Algorithm 7.1) prunes first the path constraint before it is explored further by a domain-splitting.
3. Given a path constraint, the output of the constraint solver is either a test case returned by function `QuickFindSolution`, a (set of) interval solutions of size epsilon (ϵ_box), or that the path constraint has no interval solution. When the path constraint has no interval solution, the path is actually infeasible. When an ϵ_box is returned, a test case is extracted by function `FindSolution`, as specified hereafter.

Specification 8.1 (QuickFindSolution). Let $\mathcal{C}$ be a path constraint, and b be a box. The function `QuickFindSolution`$(\mathcal{C}, b)$ returns the middle point $\mathbf{v} \in b$ if $eval(\mathcal{C}, \mathbf{v})$ holds. Otherwise it returns $\emptyset$.

Specification 8.2 (FindSolution). Let $\mathcal{C}$ be a path constraint, e be an ϵ_box and TS be a representative set of floating-point vectors in e. The function `FindSolution`$(\mathcal{C}, e)$ returns, if it exists, some vector $\mathbf{v} \in TS$ such that $eval(\mathcal{C}, \mathbf{v})$ holds. Otherwise it returns $\emptyset$.

Note that a more detailed discussion of the `FindSolution` function is given later in Section 10.6.

8.1.1 Incremental construction of the path constraint

In Algorithm 8.1, once a path constraint is constructed for a given path, it is then solved to obtain a test case. We can however improve the algorithm by incrementally filtering the path constraint during its construction. Figure 8.1 illustrates such

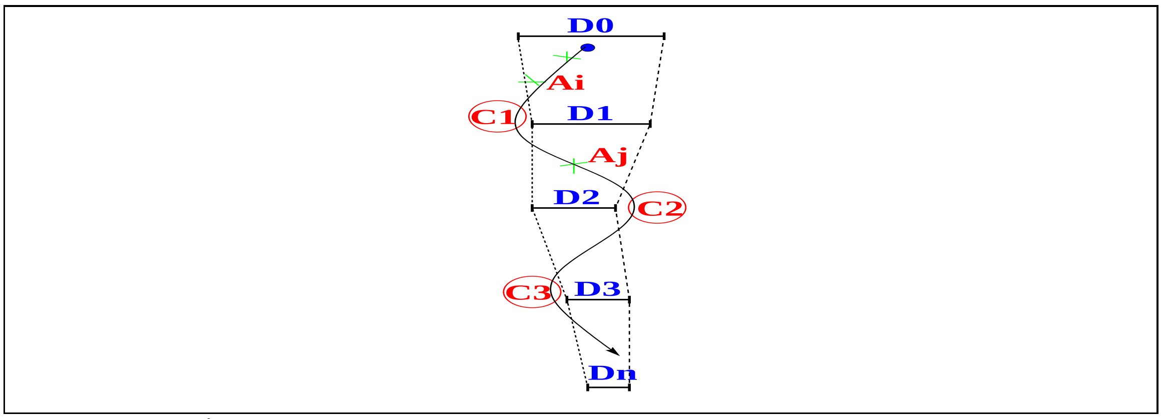


Figure 8.1: Incremental Construction of Path Constraint

incremental handling of the path constraint, where the path is represented by the

```
float nThRootBisect(float a, int n, float e))
POST Let r > 0 such that r^n = a
     Return r* with (r* − r)^2  <  e

    float l, h, c;
    function float  f(float x) = (x**n - a) ;
1.  l = 1; h = a;
2.  while ((h − l)^2  >=  e) do
3.    c = (l + h)/2;
4.    if (f(c) = 0)
5.      return c;
6.    if (f(l)*f(c) < 0)
7.      h = c;
8.    else l = c;
9.  return h;
```

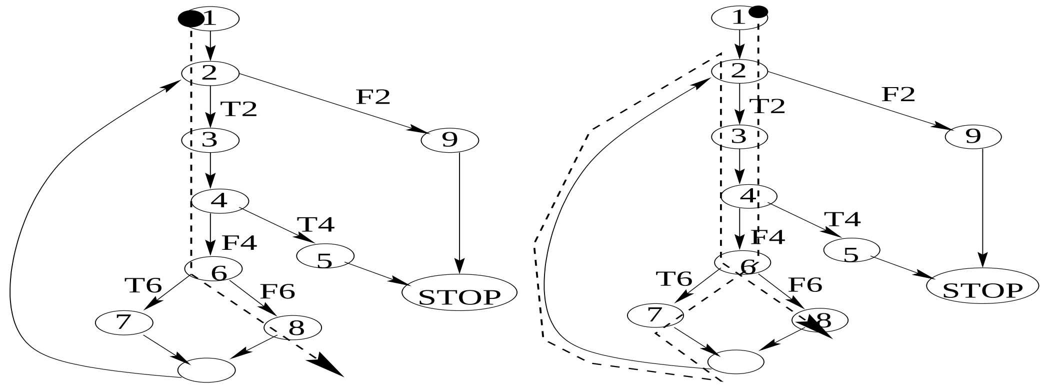


Figure 8.2: Two paths in Program nThRootBisect

arrow-headed curve; C_1, C_2, ... represent the branches on the path; A_i, A_j, ... the assignments on the path; and D_0, D_1, D_2, ..., D_n the successive domains of the input variables.

Our idea is that filtering is done at each branch C_i of the path. Therefore, if $D_i = \emptyset$, then the path is infeasible, and no more path construction is needed. As a consequence of the successive filtering steps, we have
$D_n \subseteq ... \subseteq D_1 \subseteq D_0$
Finally, if no inconsistency is detected at the end of the path-constraint's construction ($D_n \neq \emptyset$), then the solving of the path constraint is actually started.

8.1.2 Possible ways to perform path constraint solving

The objective of solving the path constraint is to get a solution for the input variables. Some approaches to path constraint solving are possible, as will be discussed below with an example of how to generate test data for the path 1-2-3-4-6-8 of the `nThRootBisect` program shown in Figure 8.2.

As presented in Chapter 6, the path constraint generated for the above path is
1: $l_0 = 1 \wedge h_0 = a_0$
2(True): $\wedge\ (h_0 - l_0)^2 \geq e_0$
3: $\wedge\ c_0 = (l_0 + h_0)/2$

4(False): $\wedge\ not(c_0^{n_0} - a_0 = 0)$
6(False): $\wedge\ not((l_0^{n_0} - a_0) * (c_0^{n_0} - a_0) < 0)$
8: $\wedge\ l_1 = c_0$

The branch constraints (denoted BC) generated for the branches of the path are: $(h_0 - l_0)^2 \geq e_0$, $not(c_0^{n_0} - a_0 = 0)$, $not((l_0^{n_0} - a_0) * (c_0^{n_0} - a_0) < 0)$.

The other constraints (denoted OC) generated for the assignments of the path are: $l_0 = 1$, $h_0 = a_0$, $c_0 = (l_0 + h_0)/2$, $l_1 = c_0$.
(Recall that these are equality constraints, where we can also denote "=" by ":=" to mean assignment constraints.)

We now discuss three possible ways to handle the path constraint, as well as their characteristics.

Substitution of BC in terms of input variables We substitute the branch constraints BC in terms of input variables by using the information from the assignment constraints OC. We then solve only these BC constraints. In other words, the constraints system includes only the branch constraints BC reexpressed in terms of input variables, as follows:
$(a_0-1)^2 \geq e_0$, $not(((1+a_0)/2)^{n_0} - a_0 = 0)$, $not((1^{n_0} - a_0) * (((1+a_0)/2)^{n_0} - a_0) < 0)$.
The constraints system together with an initial box representing the domains of the input variables a_0, n_0 and e_0, will finally be solved by Function `SolvePathConstraints` in Algorithm 8.1.

This way of handling the path constraint, although possible, can results in very large expressions and many redundant evaluations (e.g. a non-input variable, defined by an assignment constraint, may occurs many times in an expression before the substitution).

Classical constraint solving The constraints system is composed of the whole set of contraints involved in the path constraint, i.e. including both BC and OC. With the above path constraint, we obtain the following constraints system:

- $(h_0 - l_0)^2 \geq e_0$, $not(c_0^{n_0} - a_0 = 0)$, $not((l_0^{n_0} - a_0) * (c_0^{n_0} - a_0) < 0)$ (BC)
- $l_0 = 1$, $h_0 = a_0$, $c_0 = (l_0 + h_0)/2$, $l_1 = c_0$ (OC)

By contrast with the above substitution approach, the set of variables of the constraints system now includes also non-input variables. An initial box is provided only to the input variables. The other non-input variables (l_0, h_0, c_0, l_1) can be given the biggest domain, following their type. Because non-input variables are in fact dependent upon input variables by OC, and their domains can be actually calculated later during filtering.

This approach does not exploit the *unique definition property* of the variables. Moreover, the number of constraints increases considerably, if there are lots of non-input variables. The big question is whether in some cases, this approach could lead to more pruning, compared with the substitution approach.

Separating BC and OC The approach used in this work consists in separating the branch constraints and the assignment constraints. This means that we perform the solving on the branch constraints BC (i.e. the constraints system includes solely the BC constraints), and use the assignment constraints OC only for keeping intermediate results.

To achieve our aim, (1) the path constraint is represented by establishing *links* between BC and OC. For example, a link is established between $not(c_0^{n_0} - a_0 = 0)$ and $c_0 := (l_0 + h_0)/2$, meaning that the former is dependent on the latter during calculations. The other links are established by the following pairs of constraints: $< c_0 := (l_0 + h_0)/2\,, l_0 := 1 >$, $< c_0 := (l_0 + h_0)/2\,, h_0 := a_0 >$, $< (h_0 - l_0)^2 \geq e_0\,, h_0 := a_0 >$, ... (2) From these links, we know the input variables involved directly or indirectly in each branch constraint.

$$\begin{array}{rcl} (h_0 - l_0)^2 \geq e_0 & : & a_0\ ,\ e_0 \\ not(c_0^{n_0} - a_0 = 0) & : & a_0\ ,\ n_0 \\ not((l_0^{n_0} - a_0) * (c_0^{n_0} - a_0) < 0) & : & a_0\ ,\ n_0 \end{array}$$

This approach improves upon the substitution approach by working with BC as if they were directly expressed in terms of input variables, and therefore does not have to deal with important-sized expressions (following an actual substitution of BC in terms of input variables). This also means that pruning is performed only on the input variables. Furthermore, optimizations can be realized during the evaluation of BC as follows:

- If any branch constraints involve directly or indirectly many occurrences of an intermediate variable (defined by an OC constraint), this variable is evaluated only once.
- Intermediate variables are evaluated only if the domain of its dependent variables has changed.

Regarding the capacity of pruning, this approach is equivalent to the substitution approach. More research is however needed to determine if this approach is better or worse than the classical constraint solving approach; or rather, for some classes of path constraints, it is better, and for others, it is worse.

8.1.3 Example

Example 8.1. As an example for path coverage, let us look again at the path 1-2-3-4-6-8 shown in Figure 8.2. The path constraint generated is:

$l_0 := 1\ \wedge\ h_0 := a_0\ \wedge\ (h_0 - l_0)^2 \geq e_0\ \wedge\ c_0 := (l_0 + h_0)/2\ \wedge\ not(c_0^{n_0} - a_0 = 0)\ \wedge\ not((l_0^{n_0} - a_0) * (c_0^{n_0} - a_0) < 0)$.

The input variables, generated during the path constraint generation, are a_0, n_0 and e_0. With the initial box $(a_0 : [2, 1000], n_0 : [2, 20], e_0 : [1e - 4, 10])$, here are the results of the incremental filtering of the path constraint (as discussed in Subsection 8.1.1)

- At the branch $T2$: $(a_0 : [2, 1000], n_0 : [2, 20], e_0 : [1e - 4, 10])$
- At the branch $F4$: $(a_0 : [2, 1000], n_0 : [2, 20], e_0 : [1e - 4, 10])$
- At the branch $F6$: inconsistency is detected

The path is thus infeasible and no test case is found. Note that the path constraint together with the initial box is detected inconsistent before reaching Node 8.

Example 8.2. As another example for path coverage, let's look at the path 1-2-3-4-6-7-2-3-4-6-8 shown in Figure 8.2. The path constraint generated is:

$l_0 := 1 \ \wedge \ h_0 := a_0 \ \wedge \ (h_0 - l_0)^2 \geq e_0 \ \wedge \ c_0 := (l_0 + h_0)/2 \ \wedge \ not(c_0^{n_0} - a_0 = 0) \ \wedge \ (l_0^{n_0} - a_0) * (c_0^{n_0} - a_0) < 0 \ \wedge \ h_1 := c_0 \ \wedge \ (h_1 - l_0)^2 \geq e_0 \ \wedge \ c_1 := (l_0 + h_1)/2 \ \wedge \ not(c_1^{n_0} - a_0 = 0) \ \wedge \ (l_0^{n_0} - a_0) * (c_1^{n_0} - a_0) < 0 \ \wedge \ l_1 := c_1.$

With the initial box
$(a_0 : [2, 1000], n_0 : [2, 20], e_0 : [1e - 4, 10]),$
the results of incremental filtering when reaching Node 8 are
$(a_0 : [2, 9.000000000000046], n_0 : [2, 9], e0 : [1e - 4, 10]).$
We finally obtained the test case $(a_0 = 7.236079157811748, n_0 = 2, e_0 = 5.00005)$, with the path constraint solving.

8.2 General algorithm

Our general algorithm for path coverage is given in Algorithm 8.2. It is interesting to highlight the main difference between this algorithm (the path constraint with arrays) and the algorithm given in the previous section (the path constraint without arrays). Given a path constraint without arrays, its branch constraints are simplified once for all, in terms of input variables by recursively replacing non-input variables by their definitions in some assignment constraints of the path constraint. These simplified branch constraints together with an initial box (representing the domains of the input variables) are then solved to develop test cases executing the path. However when a path constraint involves arrays, it is not always possible to simplify all of its branch constraints in terms of input variables with the initial box. For example, suppose $a[i]$ (i is an expression involving input variables) is an array reference occurring in a branch constraint, then it is generally impossible to determine which array element $a[i]$ is. Therefore, the branch constraints will be simplified incrementally along with their resolution. The simplification is thus integrated in the filtering (function `Filtering` in Algorithm 8.3), which in turn is integrated in the path constraint solving (function `SolvePathConstraints`) as illustrated above. Note also that the number of (currently identified) input variables can change over the solving process. Indeed, input variables are defined by a constraint $x \in dom(x)$ (defining input variable x) or $\texttt{na3}(b, a, j)$ (defining input variable $b[j]$). If j is not a number, $b[j]$ can only be added to the set of input variables when j can be simplified into a number. The function `Filtering` realizes the filtering on the path constraint. The path constraint is represented by the branch constraints and the other constraints. As explained in Chapter 6, the pruning is only performed on the branch constraints. The function `Simplify` (Algorithm 8.4) simplifies the branch constraints by extracting information from the other constraints. The number of known input variables may increase after a simplification.

In Algorithm `Filtering`, the branch constraints are first simplified (line 1). The pruning of the branch constraints involving only input variables is performed in line 3. When the resulting box (D_t') is empty, the CSP is inconsistent. If there are branch constraints not involving input variables, these are simplified using the

Algorithm 8.2 Generation of test data for path coverage (general version)

```
function TestDataGenPC($P$:$Procedure$,$G$:$ICFG$,$p$:$Path$):$\mathcal{F}^n$;
PRE $G$ the ICFG for test procedure $P$
    $p$ a path in $G$
POST a test case on which the path $p$ is executed
begin
  $PC$:= PathConstraintGeneration($P$,$G$,$p$);
  $BC$ := the branch constraints of $PC$;
  $OC$ := $PC \setminus BC$;
  $V$ := the currently identified input variables in $BC$;
  $D$ := the domains of the variables in $V$;
  return SolvePathConstraints($V$, $V$, $D$, $BC$, $OC$);
end

function SolvePathConstraints($V,V'$:$Variables$,$D$:$Box$,
                              $BC$:$BranchConstraints$,$OC$:$OtherConstraints$):$F^n$;
PRE $V$ the currently identified input variables in $BC$
    $V'$ a subset of $V$ ($V' \subseteq V$)
    $D$ a box representing the domains of the variables in $V$
POST Return some float solution $\mathbf{v} \in D$ of $BC$
     Otherwise it returns $\emptyset$
begin
  if $BC$ involves only input variables then        // Appendix A
    $qs$ := QuickFindSolution($BC$, $D$);            // will make reference
    if $qs \neq \emptyset$ then return $qs$;         // to these 3 lines
  ($V_t$, $D_t$, $BC_t$, $OC_t$) := Filtering($V$, $D$, $BC$, $OC$);
  if $D_t$ is $\emptyset$ then return $\emptyset$;
  else
    if $D_t$ is an $\epsilon\_box$ then return FindSolution($BC_t$,$D_t$);
    else
      if $V'$ is not empty then
        Choose arbitrarily a variable $x$ in $V'$;
        $m$ := $(left(X_t) + right(X_t))/2$;
        if $x$ is an integer variable then
          $ms$ := SolvePathConstraints($V_t$, $V' \setminus \{x\}$, $D_t[X_t/[\lfloor m \rfloor, \lfloor m \rfloor]]$, $BC_t$, $OC_t$);
        else $ms$ := SolvePathConstraints($V_t$, $V' \setminus \{x\}$, $D_t[X_t/[m, m]]$, $BC_t$, $OC_t$);
        if $ms \neq \emptyset$ then return $ms$;
        if $x$ is an integer variable then
         $ls$ := SolvePathConstraints($V_t$, $V' \setminus \{x\}$, $D_t[X_t/[left(X_t), \lfloor m \rfloor - 1]]$, $BC_t$, $OC_t$);
        else $ls$ := SolvePathConstraints($V_t$, $V' \setminus \{x\}$, $D_t[X_t/[left(X_t), m^-]]$, $BC_t$, $OC_t$);
        if $ls \neq \emptyset$ then return $ls$;
        if $x$ is an integer variable then
         $rs$ :=SolvePathConstraints($V_t$, $V' \setminus \{x\}$, $D_t[X_t/[\lfloor m \rfloor + 1, right(X_t)]]$, $BC_t$, $OC_t$);
        else $rs$ := SolvePathConstraints($V_t$, $V' \setminus \{x\}$, $D_t[X_t/[m^+, right(X_t)]]$, $BC_t$, $OC_t$);
        if $rs \neq \emptyset$ then return $rs$ else return $\emptyset$;
      else return SolvePathConstraints($V_t, V_t, D_t, BC_t, OC_t$);
end
```

Algorithm 8.3 Filtering of path constraints

```
function Filtering($V$:$Variables$,$D$:$Box$,
                   $BC$:$BranchConstraints$,$OC$:$OtherConstraints$):$CSP$;
PRE
  ($V^*$,$D$,$BC \wedge OC$) is a CSP where $V^* = vars(BC \wedge OC)$
  $V$ the set of input vars currently identified in branch constraints $BC$ ($V \subseteq V^*$)
  $D$ a box representing the domains of the variables in $V$
POST
  Return a CSP ($V$,$\emptyset$,$BC \wedge OC$) if $BC$ is detected as inconsistent.
  Otherwise return ($V'$, $D'$, $BC' \wedge OC'$)
  with • $V \subseteq V' \subseteq V^*$
       • ($V^*$, $D'$, $BC' \wedge OC'$) CSP equivalent to ($V^*$,$D$,$BC \wedge OC$)
       • $BC'$ = $BC'_1 \wedge BC'_2$
       • $BC'_1$ e-box consistent
begin
1:($V_t$, $D_t$, $BC_t$, $OC_t$) := Simplify($V$, $D$, $BC$, $OC$);
  $C_1$ := branch constraints (involving only input variables) of $BC_t$;
  $C_2$ := $BC_t \setminus C_1$;
  $Store$ := $C_1$;
2:while $C_1 \neq \emptyset$ do
3:  ($V_t$, $D'_t$, $Store$) := PhiEBox($V_t$, $D_t$, $Store$);
    if $D'_t = \emptyset$ then return ($V$,$\emptyset$,$BC$,$OC$);
    if $D'_t = D_t$ then break;
    $D_t$ := $D'_t$;
    if $C_2 = \emptyset$ then break;
4:  ($V'_t$, $D'_t$, $C'_2$, $OC'_t$) := Simplify($V_t$, $D_t$, $C_2$, $OC_t$);
    $C_1$ := branch constraints (involving only input variables) of $C'_2$;
    $C_2$ := $C'_2 \setminus C_1$;
    $Store$ := $Store \wedge C_1$;
    $V_t$ := $V'_t$;  $D_t$ := $D'_t$; $OC_t$ := $OC'_t$;
  endwhile
5:return ($V_t$, $D_t$, $Store \wedge C_2$, $OC_t$);
end
```

reduced domains. This is performed until $C_1 = \emptyset$ (nothing to prune), or no pruning is achieved ($D'_t = D_t$), or all branch constraints only involve input variables ($C_2 = \emptyset$). Finally the function returns a new CSP (line 5), satisfying (1) *Store* (all branch constraints involving only input variables) is e-box consistent in box D_t, and (2) C_2 (the other branch constraints involving non-input variables) cannot be simplified further with box D_t.

We now analyze in detail the function `Simplify`. The function `Simplify` returns an equivalent but simplified CSP. The objective is to simplify the branch constraints BC in terms of the input variables in V with the box D. If BC involves only input variables (line 1), the function returns the input CSP without modifications. Otherwise, it enters in the main loop until no more simplification can be done. The following simplifications are performed. In line 2, every non-input simple variable x is replaced by its definition. A variable is simple if it is neither an array variable nor an array element. This simplification of simple variables is performed only once in the first call of Function 8.4. The simplification of array variables is however more complex and must be done incrementally during the solving process.

Algorithm 8.4 Simplification of path constraints

```
function Simplify($V$:$Variables$,$D$:$Box$,
                  $BC$:$BranchConstraints$,$OC$:$OtherConstraints$):$CSP$;
PRE
  ($V^*$,$D$,$BC \wedge OC$) is a CSP
  $V$ the set of input vars currently identified in branch constraints $BC$ ($V \subseteq V^*$)
  $D$ a box representing the domains of the variables in $V$
POST
  Return a CSP ($V$,$\emptyset$,$BC \wedge OC$) if $BC$ is detected as inconsistent.
  Otherwise, return ($V'$,$D'$,$BC' \wedge OC'$)
  with • $V \subseteq V' \subseteq V^*$
       • ($V^*$, $D'$, $BC' \wedge OC'$) CSP equivalent to ($V^*$,$D$,$BC \wedge OC$)
begin
1:if $BC$ involves only input variables then return ($V$,$D$,$BC$,$OC$);
  else
2:  while $\exists$ a simple and non-input variable $x$ in $BC \wedge OC$ do
      $BC[x/def(x)]$; {There must exists a constraint, $x := def(x)$, in $OC$}
      $OC[x/def(x)]$; {simplification for variable $x$ once for all}
    $simplify$ := $true$;
3:  while $simplify$ do
      $simplify$ := $false$;
4:    foreach constraint na3($b$, $a$, $j$) in $OC$ with $b[j]$ not in $V$ such that
      $j$ involves only input variables with their domains being point intervals do
          $j$ is simplified into a number $jval$;
          $OC$[na3($b$, $a$, $j$)/na3($b$, $a$, $jval$)]; $BC[b[j]/b[jval]]$;
          $V$ := $V \cup \{b[jval]\}$;
          $simplify$ := $true$;
5:    foreach na4($b,a,j,v$) in $OC$ such that
      $j$ involves only input variables with their domains being point intervals do
        $j$ is simplified into a number $jval$;
        $OC$[na4($b$, $a$, $j$, $v$)/na4($b$, $a$, $jval$, $v$)];
        $simplify$ := $true$;
6:    foreach $b[i]$ in $BC$ such that
      $i$ involves only input variables with their domains being point intervals do
        $i$ is simplified into a number $ival$;
        $BC[b[i]/b[ival]]$;
        $simplify$ := $true$;
7:    foreach $b[i]$ in $BC$ such that $i$ is a number and $b[i]$ is not an input variable do
7a:     case $\exists$ ($b$ := $a$) in $OC$ : $BC[b[i]/a[i]]$; $simplify$ := $true$;
7b:     case $\exists$ na3($b$, $a$, $j$) in $OC$ | $j$ is a number :
          if $a \neq null$ then $BC[b[i]/a[i]]$; $simplify$ := $true$; else return ($V$,$\emptyset$,$BC$,$OC$);
7c:     case $\exists$ na4($b$, $a$, $j$, $v$) in $OC$ | $j$ is a number :
          if $i = j$ then $BC[b[i]/v]$; $simplify$ := $true$;
          else if $a \neq null$ then $BC[b[i]/a[i]]$; $simplify$ := $true$;
          else return ($V$,$\emptyset$,$BC$,$OC$);
        endcase
    endwhile
8:  return ($V$, $D$, $BC$, $OC$);
  endif
end
```

Following the simplification of non-input simple variables, the next steps have the purpose of simplifying constraints involving arrays. Lines 4 and 5 simplify the constraints na3 and na4 in OC. Line 6 simplifies reference to array element $b[i]$, where the index is known. Finally, in line 7, every reference to such array element $b[i]$ is propagated in some constraint of OC, which can be one of the following constraints:

- $b := a$ (this kind of constraints is generated only during parameter passing of array variables), in line 7a;
- na3 (line 7b);
- and na4 (line 7c).

Note that see Definition 6.4 for the definition of *null* arrays. An inconsistency can be detected when an array element is used in an expression without being initialized.

Example

Example 8.3. As an example for path coverage, let us take the path 1-2-3-4-5a-8-9-10a-16-17-18-20-10b-11a-16-17-18-20-11b-12-13a-21-22-23-24-13b-15-5b-6-4-7-25 in the ICFG given in Figure 3.3. As illustrated in Example 6.2 (Chapter 6), the generated path constraint is:

$\bigwedge_{0 \leq i < 10} \mathbf{a_0}[i] \in dom(a) \wedge \mathbf{c_0} \in dom(c) \wedge \mathrm{i_0} := 1 \wedge \mathrm{i_0} \leq \mathrm{c_0} \wedge \mathrm{a_1} := \mathrm{a_0} \wedge \mathbf{i_1} \in dom(i) \wedge \mathbf{j_0} \in dom(j) \wedge \mathrm{i_2} := \mathrm{i_1} \wedge \mathrm{i_2} \geq 0 \wedge \mathrm{i_2} \leq 9 \wedge \mathrm{fi_0} := \mathrm{i_2} \wedge \mathrm{i_3} := \mathrm{j_0} \wedge \mathrm{i_3} \geq 0 \wedge \mathrm{i_3} \leq 9 \wedge \mathrm{fj_0} := \mathrm{i_3} \wedge \mathrm{fi_0} < \mathrm{fj_0} \wedge \mathrm{x_0} > \mathrm{y_0} \wedge \mathrm{t_0} := \mathrm{x_0} \wedge \mathrm{x_1} := \mathrm{y_0} \wedge \mathrm{y_1} := \mathrm{t_0} \wedge \mathsf{na4}(\mathrm{a_2}, \mathrm{a_1}, \mathrm{i_1}, \mathrm{x_1}) \wedge \mathsf{na4}(\mathrm{a_3}, \mathrm{a_2}, \mathrm{j_0}, \mathrm{y_1}) \wedge \mathrm{a_4} := \mathrm{a_3} \wedge \mathrm{i_4} := \mathrm{i_0} + 1 \wedge \neg(\mathrm{i_4} \leq \mathrm{c_0})$.

Note that a_0 (array variable), c_0, i_1, and j_0 are the input variables generated during the path constraint generation. Given the initial box ($a_0 : [5, 20], c_0 : [1, 10], i_1 : [-5, 20], j_0 : [-5, 20]$), we obtained the test case:
$a_0 = (12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 8.75, 12.5, 12.5)$, $c_0 = 1$, $i_1 = 4$, $j_0 = 7$.

8.3 Analysis

Our constraint solving algorithm for path coverage (Algorithm 8.2) is sound but not complete, because the `FindSolution` function is incomplete. If it does not find test data, it could be that the path is infeasible. Indeed, given an epsilon interval solution, we take only some points in it to check if they are test cases. Of course, we can make a complete labeling in interval solutions, and hence having a complete solver, but then the complexity may become too expensive. However, since path constraints are usually under-constrained (there are many test cases traversing the path if it is feasible), and the epsilon can be chosen very small (usually *1e-16* in our experiments), even a middle point in the interval solution turns out to be sufficient as will be illustrated by our experiments.

The constraint solving problem handled in this work, as well as some other constraint solving problems in constraint programming, are $\mathcal{NP}$-complete, because they can be reduced to the SAT problem in Computability Theory. This means that backtracking search is, in general, an important technique in solving them. As a consequence, our constraint solving algorithm can be considered as belonging to the class of algorithms based on backtracking search with propagation (in Constraint Programming). The propagation is realized here by our e-box consistency filter.

8.4 Conclusion

We presented, in Section 8.1, a simple version of our test data generation algorithm for path coverage. We particularly showed how our earlier consistency algorithms are integrated in a constraint solving algorithm to generate test data. The purpose of this section is thus to allow the reader to skip the general version (given in Section 8.2), without any problem to understand the rest of this thesis. Section 8.2 deals with how constraints involving arrays —na3 and na4 constraints— are actually handled. The notion of filtering is also reviewed in the presence of arrays in the test program. In Section 8.3, we gave an analysis of our test data generation algorithm for path coverage.

In the next chapter, we will present a test data generation algorithm for statement coverage. Such algorithm intensively uses consistency, as well as the results of our algorithm for path coverage, to generate test data exercising a node of the ICFG.

Chapter 9

Test Data Generation for Statement Coverage

This chapter proposes an algorithm of test data generation for statement coverage (searching for test data traversing a node of the ICFG). Note that as a branch is dual to a statement in the control flow graph, all the following algorithms can easily be adapted for branch coverage. Given a node, different paths reaching the node will be dynamically generated. The search will be guided by a Control Dependence Graph, as well as pruned by our e-box consistency filter. First, two different control dependences for programs with procedure calls are introduced: the intraprocedural and the interprocedural control dependences. We will show that the interprocedural control dependence is better for our purpose.

9.1 Control Dependence Graph

Control dependence captures the effects of predicate statements (`if`, `while`, ...) on the program's behavior. Intuitively, a node a is linked to a node b in the control dependence graph if any execution path reaching b contains also a. In other words, reaching statement a is a necessary condition to reach statement b. Technically, control dependence is defined in terms of a CFG and the post-dominance relation among the nodes in the CFG [22].

Definition 9.1. A node V is *post-dominated* by a node W in a CFG G, if every directed path from V to $Exit_G$ (not including V) contains W.

A node Y is *control dependent* on node X with condition C (where C is a condition associated with a branch from X) iff

1. there exists a directed path P from X to Y going through branch C such that all Z in P (excluding X and Y) are post-dominated by Y, and
2. X is not post-dominated by Y.

Note that if node Y is control dependent on (node X, condition C) then node X must have two branches. Following one of the branches corresponding to condition C results in Y being executed while taking the other results in Y not being executed.

Table 9.1: Intraprocedural control dependences of Program 1

Nodes	Control Dependent On
3,4,7	(2, true)
4,5a,5b,6	(4, T4)
9,10a,10b,11a,11b,12,15	(8, true)
13a,13b	(12, T12)
14a,14b	(12, F12)
17,20	(16, true)
18	(17, T17)
19	(17, F17)
22,24	(21, true)
23	(22, T22)

Table 9.2: Interprocedural control dependences of Program 1

Nodes	Control Dependent On
3,4	(2, true)
5a,8,9,10a,16,17	(4, T4)
7	(4, F4)
13a,13b	(12, T12)
14a,14b	(12, F12)
4,5b,6,10b,11a,11b,12	(17, T17)
15,16,17,18,20,21,22	(17, T17)
19	(17, F17)
23,24,25	(22, T22)

Intraprocedural control dependence analysis is carried out independently on individual procedures, calculating thus control dependences that exist within them. Concretely, given the CFG for each procedure, intraprocedural control dependences for the procedure are obtained by applying an existing algorithm for control dependence computation [22] to the CFG. Table 9.1 illustrates the intraprocedural control dependences for all procedures of Program 1, given in Figure 9.1. Note that (1) the CFGs for those procedures are extracted from the ICFG (in Figure 9.1) by ignoring, for each call site, its pair of call and return edges, and connecting directly its call node with its return node; (2) we view the entry node of the CFG associated with a procedure as a predicate node representing the conditions that cause the procedure to be executed, and therefore nodes in the CFG that are not control dependent on any condition nodes are control dependent on the entry node. In the table, for example, node 3 is control dependent on node $Entry\,M$ (node 2) with condition *true*, and node 5a on node 4 with condition $T4$.

Interprocedural control dependence analysis accounts for interactions between individual procedures. Those interactions are reflected by call and return edges,

```
void M(double a[10], int c) {
  int i = 1;
  while (i <= c) {
    B(a);
    i = i+1;
  }
}

void C(double *x, double *y) {
  double t;
  if (*x > *y) {
    t = *x;
    *x = *y;
    *y = t;
  }
}
```

```
void B(double a[10]) {
  int i,j,fi,fj;
  scanf("%d %d", &i, &j);
  fi = F(i); fj = F(j);
  if (fi < fj)
    C(&a[i], &a[j]);
  else C(&a[j], &a[i]);
}

int F(int i) {
  if (i >= 0 && i <= 9)
    return i;
  else exit(1);
}
```

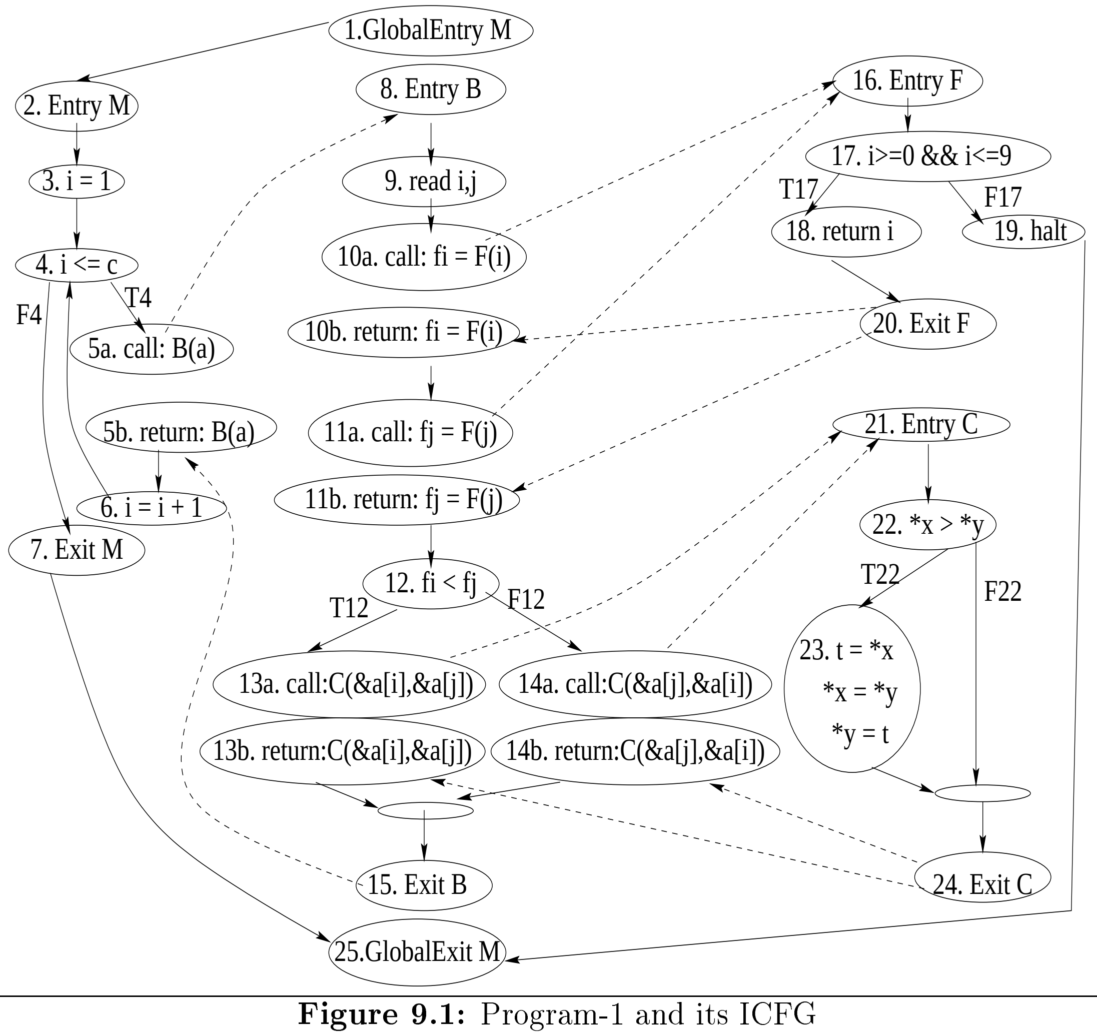


Figure 9.1: Program-1 and its ICFG

connecting the individual CFGs, in the ICFG. Interprocedural control dependence can be computed for the nodes of the ICFG by an existing technique [61]. Table 9.2 illustrates the interprocedural control dependences for Program 1.(Note that a graphical representation of Table 9.2 is also given in Figure 9.2.) A comparison between these dependences and those computed intraprocedurally (in Table 9.1) shows several differences.

1. There are intraprocedural dependences which are ignored in the interprocedural context, e.g. node 9 is intraprocedurally control dependent on node *EntryB* (node 8) while this dependence is not interprocedurally necessary.
2. There are interprocedural dependences between nodes in different procedures while these dependences cannot be computed intraprocedurally, e.g. node 6 is interprocedurally control dependent on node 17. Note that the presence of embedded `halt` statements in called procedures are not the only cause of such dependences [61].
3. There are interprocedural dependences between nodes in the same procedures, yet these dependences are not intraprocedurally established, e.g. node 7 is interprocedurally dependent on node 4 while this is not intraprocedurally detected.

All these differences show that intraprocedural control dependences can be imprecise to guide the search of test data for programs with procedure calls. We hence choose to use an interprocedural control dependence graph for this purpose.

Definition 9.2 (ICDG). An *interprocedural control dependence graph* (ICDG) for a procedure P is a directed graph where the nodes are the nodes of the ICFG associated with P. The edges represent the interprocedural control dependences between nodes. Edges are labeled with conditions. An edge (X, Y) labeled with a condition C in an ICDG means that Y is interprocedurally control dependent on (X, C). There will be however no edge for a node that is interprocedurally control dependent on itself.

Figure 9.2 depicts the ICDG for Program 1, which is actually a graphical representation of Table 9.2. Note that, for simplicity, additional nodes are introduced in the ICDG to group all nodes with the same control conditions together, e.g. nodes 5a,8,9, ... (interprocedurally control dependent on node 4 with condition $T4$) are grouped together under an additional node.

Definition 9.3 (Reachability graph). The *reachability graph* for a node n in a directed graph G (with a unique start node) is the smallest subgraph of G, containing all the paths from the start node to node n.

Definition 9.4 (Decision graph). The *decision graph* for a node n in an ICDG G is the reachability graph for n in G.

Note that Definition 9.3 is general for any directed graph while Definition 9.4 is specific to an ICDG.

The construction of the reachability graph and the decision graph for a node is straightforward. For example, the decision graph for node 7 is depicted in dashed lines in Figure 9.2. Given the decision graph for a node, a path from the root of the

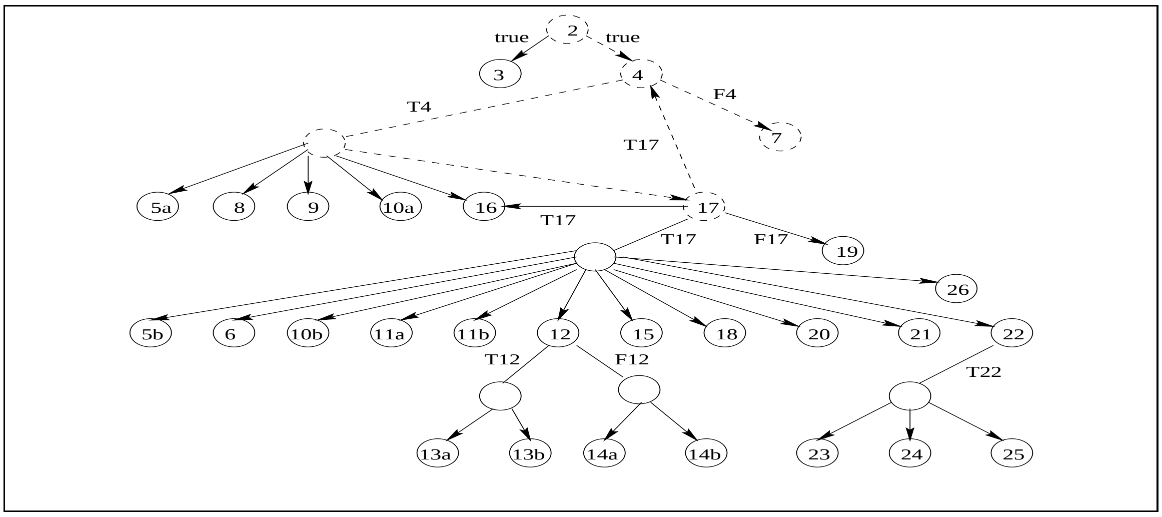


Figure 9.2: ICDG for Program 1 and the decision graph for node 7 in dashed lines

graph to the node contains a set of constraints that must be satisfied by a class of inputs causing the node to be executed. For example, the path 2-4-7 in the decision graph for node 7 corresponds to inputs executing node 7 with no passage in the loop (associated with condition node 4), while the path 2-4-17-4-7 corresponds to inputs executing node 7 with one passage in the loop. Therefore, the decision graph for a node captures all the possible constraints to satisfy to reach the node.

9.2 Algorithm

The generation of test data for statement coverage is described in Algorithm 9.1 (`TestDataGenerationSC`). Paths reaching the input node (node N) are dynamically constructed. When a path reaches this node, test data generation for path coverage is used to find a test case. Note that the search for such paths is carried out on the reachability graph for the input node in the ICFG. As the potential number of paths reaching the node can be large (or infinite), heuristics and pruning are used during the search. First, the search is guided by the ICDG, and more particularly by the decision graph. The algorithm always extends a path by first choosing nodes in the decision graph, as such nodes are required in the path. Second, the exit of the loop is also selected first to avoid infinite paths. Third, the search is pruned by our e-box filtering operator (function `Filtering` in Algorithm 8.3). A path (or prefix) is abandoned as soon as we detect that it is an infeasible path. Given such infeasible prefixes, we can memorize them in order to avoid choosing them uselessly later.

This algorithm can be optimized in many ways (incremental construction of path constraints, ...). We however prefer to present a simple and comprehensive version. Note also that when node b is control dependent on node a with condition c, then reaching node a and traversing the branch corresponding to condition c is already sufficient to reach node b. As a consequence, if the aim is to generate test data for node b, the construction of paths can be stopped as soon as <node a, condition c> is traversed.

Algorithm 9.1 Statement coverage

```
function TestDataGenerationSC($P$:$Procedure$, $G$:$ICFG$, $N$:$Node$) : $\mathcal{F}^n$;
PRE $G$ The ICFG for test procedure $P$
    $N$ a node in $G$
POST a test case traversing node $N$
begin
  $G1$ := reachability graph for node $N$ in $G$;
  $G2$ := $ICDG$ for $G$;
  $DG$ := decision graph for node $N$ in $G2$;
  return TestGen($P$, $G1$, <$START$>, $START$, $N$, $DG$);
  {START is the start node in $G1$}
end

function TestGen($P$:$Procedure$, $G$:$ReachabilityGraph$,
                 $path$:$Path$, $start$:$Node$, $end$:$Node$, $DG$:$DecisionGraph$) : $F^n$;
PRE $path$ a path in $G$
   $DG$ the decision graph for node $end$
POST a test case traversing node $end$
begin
  for each successor $s$ of $start$ in $G$ do
  {If $start$ in $DG$, the successors in $DG$ are selected first,}
  {if $start$ is a loop, the exit of the loop is selected first}
    $newPath$ = $path$ . $s$ ;
    $PC$ := PathConstraintGeneration($P$,$G$,$newPath$);
    $BC$ := the branch constraints of $PC$;
    $OC$ := $PC \setminus BC$;
    $V$ := the input variables currently identified in $BC$;
    $D$ := the domains of the variables in $V$;
    ($V'$, $D'$, $BC'$, $OC'$) := Filtering($V$, $D$, $BC$, $OC$); //this line will be referenced
    if ($D' \neq \emptyset$) then
      if ($s$ = $end$) then
        {test data generation for path coverage}
        $result$ = SolvePathConstraint($V'$,$V'$,$D'$,$BC'$,$OC'$);
        if $result \neq \emptyset$ then return $result$;
      else return TestGen($P$,$G$,$newPath$,$s$,$end$,$DG$);
  endfor
  return $\emptyset$;
end
```

As an example, consider node 7 in Figure 3.3, as well as its decision graph shown in dashed lines in Figure 9.2. First, the path 1-2-3-4-7 will be constructed by the algorithm. Assuming that the corresponding path constraint is inconsistent, the path 1-2-3-4-5a-8-9-10a-16-17-18-20-10b-11a-16-17-18-20-11b-12-13a-21-22-23-24-13b-15-5b-6-4-7 (entering in the main loop) is next constructed. Generation of test data for the latter path was already given in Example 8.3 (Chapter 8.2).

9.3 Analysis

Our algorithm of test data generation for statement coverage (Algorithm 9.1) is sound but not complete. It may loop or fail to find test data. This follows from the fact that determining whether a node of the control flow graph is executable, is undecidable in the general case (reduced to the halting problem in computability theory) [68]. Also, it was reported in [25] that there exist loops, for which the termination with some data is unknown up till now. So it is impossible to determine if an instruction placed after a loop is executable in the general case.

9.4 Towards a dynamic approach to statement coverage

This section aims to propose a dynamic approach to statement coverage, where random test data generation, program execution, and any method for path coverage such as ours, can be integrated.(The idea of this dynamic approach is closely related to [69].) There are some reasons to develop such a dynamic approach.

- Intuitively, our static consistency approach to statement coverage may be time-consuming or even impractical for programs of certain size. For example, we want to find test data for a statement placed after loops executed a great number of times. In that case, numerous constraints can be generated, and the time taken to solve them can be large, while random test data generation may be sufficient to apply.
- Given the fact that the selection of paths reaching a node relies only on our e-box consistency filter, a possible drawback is that a path accepted by such filter may not be executable. This is due to that a constraints system being e-box consistent does not guarantee having float solutions.

We below present the main steps of our dynamic approach.

Step 1 (Random test data generation): Test data are generated randomly (existing techniques for test data generation can also be applied) for a maximum number of times fixed by the user. Program execution is used to run each test data to verify if a given statement (*Node*) is executed. If so, the test data is returned, and the approach terminates. The next step takes place only if no test data thus generated are proved to execute the node. Note that during this step, when a test data is executed, the trace of its execution (i.e. the execution path) is memorized for later uses.

Step 2 (Selection of useful prefixes): Useful prefixes of all paths (either memorized in the previous step, or obtained from Step 5) are derived. Informally, a *prefix* of a path is defined as a subpath starting from the first node in the path.

Step 3 (Extension of prefixes, one step towards the specified Node): Construct extended prefixes from such prefixes, by adding a node to each prefix. Note that extended prefixes are possibly not paths attaining *Node*.

Step 4: Apply our path coverage algorithm to each extended prefix to generate candidate test data. Note that any method for path coverage such as ours, [30] or [46] can be used here. This step should be done in an incremental manner so that information can be reused from one iteration to the next to improve performance — e.g. prefixes that are detected as infeasible should be memorized so that we will not explore any further paths from such prefixes.

Step 5: Execute candidate test data to see if its execution path reaches *Node*. If not the case, goto Step 2.

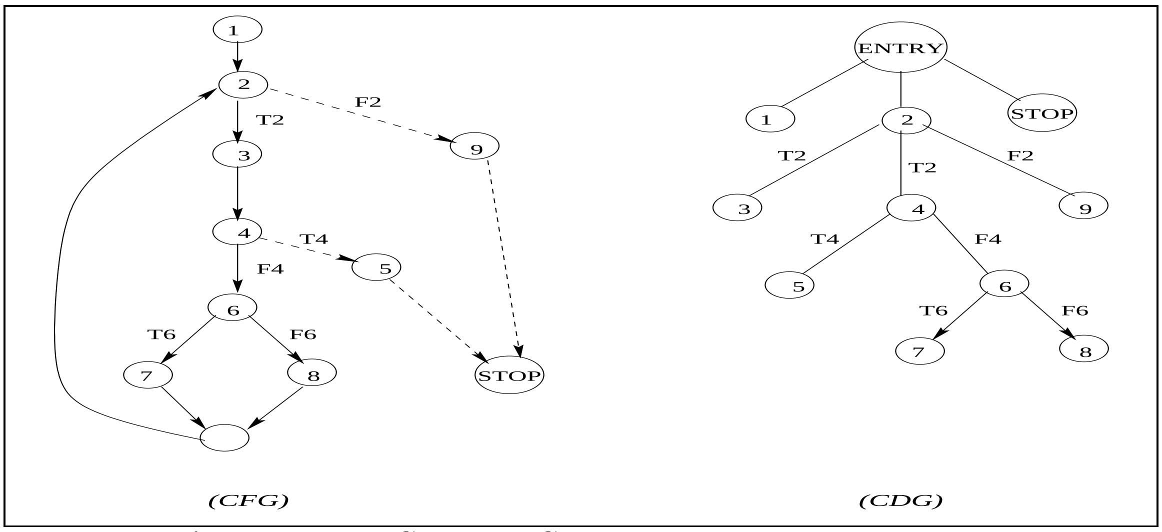


Figure 9.3: CFG and CDG of the nThRootBisect program

We now take an example to illustrate our idea. Suppose that we want to generate test data for node 8 in Figure 9.3 (the same as Figure 5.3), and that with a randomly generated test data, we obtain the path 1-2-3-4-6-7-2-3-4-5-*STOP*.

In Step 2, by using the control dependences in the right hand side of Figure 9.3, more exactly the decision graph for node 8, we obtain the following prefixes from the above path: 1-2-3-4-6 and 1-2-3-4-6-7-2-3-4. Note that each prefix is derived whenever we are at a decision node where the branch taken is not as proposed by the decision graph; that is, we should better take the other branch.

In Step 3,

- from prefix 1-2-3-4-6, we obtain extended prefix 1-2-3-4-6-8
- from prefix 1-2-3-4-6-7-2-3-4, we obtain extended prefix 1-2-3-4-6-7-2-3-4-6.

As illustrated by the above example, each path results in a set of prefixes, and each prefix in turn leads to the construction of an extended prefix. This set of extended prefixes will be used by our method for path coverage to generate candidate test data. To be efficient, at each time, we should select the most "promising" extended prefix among such extended prefixes by some of the following possible criteria (other criteria can be used as well):

- an extended prefix with the shortest length in terms of the number of *branches*
- an extended prefix with the shortest length in terms of the number of *nodes*
- an extended prefix ending in a node that is "nearest" to the specified $Node$ (i.e. taking such an extended prefix leads us closer to $Node$). For example, with node 8 above, extended prefixes ending in node 8 are hence in priority, compared with extended prefixes ending in node 6 and extended prefixes ending in node 2.

In Steps 4 and 5, when a test data is generated for an extended prefix by our path-coverage method, it will be run to verify if node 8 is executed. If so, it is returned as a test case. Otherwise, its execution path is exploited to derive new extended prefixes as above, and they are then added to the set of extended prefixes for generating test data. For example, assuming a test data is generated for the extended prefix 1-2-3-4-6-7-2-3-4-6 above, and its execution results in the path 1-2-3-4-6-7-2-3-4-6-7-2-9-$STOP$, we then obtain the following new prefixes and extended prefixes:

- prefixes: 1-2-3-4-6-7-2-3-4-6 and 1-2-3-4-6-7-2-3-4-6-7-2
- extended prefixes: 1-2-3-4-6-7-2-3-4-6-8 and 1-2-3-4-6-7-2-3-4-6-7-2-3.

The whole test data generation process is carried out until all the allowed resources (e.g. timeout, ...) are exhausted. Our above ideas are finally summarized in Algorithm 9.2.

9.5 Conclusion

We presented in this chapter an algorithm of test data generation for statement coverage. Our algorithm searches the ICFG for paths reaching a specified node. The search is guided by the interprocedural control dependences of the program, as well as pruned by our e-box consistency filter. When such a path if found, test data generation for path coverage is then applied to find test data.

In Section 9.1, we introduced the two different kinds of control dependences for programs with procedure calls: the intraprocedural and the interprocedural control dependences. We showed that interprocedural control dependence is more precise to use in the presence of procedure calls in the test program, e.g. when called procedures contain `halt` statements. It should be noted that existing methods of test data generation [54, 26], based on the control flow graph, use only intraprocedural control dependence to guide the search process. Therefore, they may be unable to determine the possible effect of the called procedures on execution of the selected statement. Note however that a method for programs with procedures [47] uses some sort of interprocedural data dependence analysis to guide the search. In Section 9.2 and Section 9.3, we dealt with our algorithm of test data generation for statement coverage and its analysis. Since the generation of test data executing a node — the problem— is undecidable in the general case, any approach to the problem —a solution— is thus incomplete. And so is our method. Therefore, in such case, one should consider using heuristics. In Section 9.4, we proposed a dynamic approach to

Algorithm 9.2 Dynamic algorithm for statement coverage

```
function DynamicTestDataGenSC($P$:$Procedure$, $G$:$ICFG$, $N$:$Node$, $Max$:$integer$) : $\mathcal{F}^n$;
PRE $G$ the ICFG for test procedure $P$
    $N$ a node in $G$
    $Max$ the maximum number of times for random test data generation
POST a test case traversing node $N$
begin
  {Step 1}
  $memorized\_paths$ := $\emptyset$;
  for $i$ from 1 to $Max$ do
    $rtd$ := a randomly generated test data for procedure $P$;
    if (execution of $P$ with $rtd$ reaches $N$) then return $rtd$;
    else
      $path$ := the execution path in $G$ for executing $P$ with $rtd$;
      $memorized\_paths$ := $memorized\_paths$ $\cup$ $path$;

  {Step 2,3,4,5}
  $G2$ := $ICDG$ for $G$;
  $DG$ := decision graph for node $N$ in $G2$;
  $set\_extendedPrefixes$ := $\emptyset$;
  for each $path$ of $memorized\_paths$ do
    derive prefixes and $extended\_prefixes$ from $path$, based on $DG$;
    $set\_extendedPrefixes$ := $set\_extendedPrefixes$ $\cup$ $extended\_prefixes$;
  while $set\_extendedPrefixes$ $\neq$ $\emptyset$ do
    $extendedPrefix$ := the most ``promising'' extended prefix in $set\_extendedPrefixes$;
    $PC$ := PathConstraintGeneration($P$,$G$,$extendedPrefix$);
    $BC$ := the branch constraints of $PC$;
    $OC$ := $PC$ $\setminus$ $BC$;
    $V$ := the input variables currently identified in $BC$;
    $D$ := the domains of the variables in $V$;
    {test data generation for path coverage}
    $td$ := SolvePathConstraint($V$,$V$,$D$,$BC$,$OC$);
    if ($td$ $\neq$ $\emptyset$ and execution of $P$ with $td$ reaches $N$) then return $td$;
    else
      $path$ := the execution path in $G$ for executing $P$ with $td$;
      derive prefixes and $extended\_prefixes$ from $path$, based on $DG$;
      $set\_extendedPrefixes$ := $set\_extendedPrefixes$ $\cup$ $extended\_prefixes$;
  endwhile
  return $\emptyset$;
end
```

statement coverage. However, it has not been implemented yet in our COTTAGE system (presented in Chapter 10).

Part III

Implementation and Experimental Results

Chapter 10

The COTTAGE System

This chapter describes the system (called COTTAGE) that has been developed to validate our consistency method, as well as the facilities provided by COTTAGE. We will first describe COTTAGE from a "black-box" viewpoint: the *development* of COTTAGE, the *subset of C* treated by COTTAGE, as well as its *input* and *output*, are discussed. The *parameters* of COTTAGE are however presented during our "white-box" presentation of the system, because the former is dependent on the latter. The white-box presentation includes the following issues: the *implementation* of COTTAGE (especially its architecture), how *function calls* are actually handled, *type analysis* and its role in our system, *soundness and completeness* of COTTAGE. Finally, possible *extensions* of the system are discussed along with their technical issues.

10.1 Development

The COTTAGE system —written in *Java*— is intended for test data generation of programs written in (a subset of) *C*. The system is an extension of our previous prototype [64]. Note that our previous prototype used only our internal representation of the test program as input. To build COTTAGE, we thus constructed a parser in Java (by using *JavaCC*, a well-known Java tool) to translate the *C* program under test into such internal representation. The system uses

- an interval library [37], relying on [38], for the implementation of the constraint solving algorithm,
- and algorithms from [61] to construct the interprocedural control dependences of the test program.

Without these libraries, the COTTAGE system is about 13,000 Java lines. The implementation is designed, however, to be independent from the programming language used by the program under test. This means that the source code, written in some imperative language $\mathcal{L}$, is first translated into an internal representation that is common to many languages, such as C, Pascal, etc.

10.2 The subset of C

The COTTAGE system is able to generate test data for programs written in a subset of C. The following features are not yet supported by our system:

1. non-numeric types such as string, two (and more) dimensional arrays (note however that our approach can easily be extended to multi-dimensional arrays), enum, struct, union, general pointers, ... ;
2. type-qualifiers: `const` and `volatile`;
3. storage-class-specifiers: `auto`, `register`, `static`, `extern` (since we handle only a single source file), `typedef`;
4. labeled-statements such as `case`, `default`, and `switch`;
5. jump-statement `goto` (note that `continue` and `break` statements are both handled);
6. operators: %, <<, >>, &, $\wedge$, |, ?, `sizeof`;
7. control-lines, except `#define` for numeric constants, and `#include`, are both handled.

10.3 Input

The input to the system is composed of two things:

- a single source program (possibly containing multiple procedures) that is written in a *subset* of C (described later),
- a file containing the initial data (such as the name of a test procedure in the source program, the domains for the input variables) and the values for the *parameters* of the system (described later).

10.4 Output

Given an input source program, it is transformed into an internal representation, namely the interprocedural control flow graph (ICFG) of the program. The primary output from the system is an indication of coverage for the criteria *all-the-statements* on the ICFG. Such a coverage indication is the percentage of the ICFG's nodes executed by the generated test data, and is calculated by:
(Total executed nodes / Total nodes of the ICFG) $\times$ 100.

For each generated test data, the system also automatically generates an instrumented C program (a file of test data ready for execution), allowing the user to verify the correctness of the test data.

Note that we have not handled the *all-the-paths* criteria yet in our system. Our current objective is to cover only all the statements in the test program.

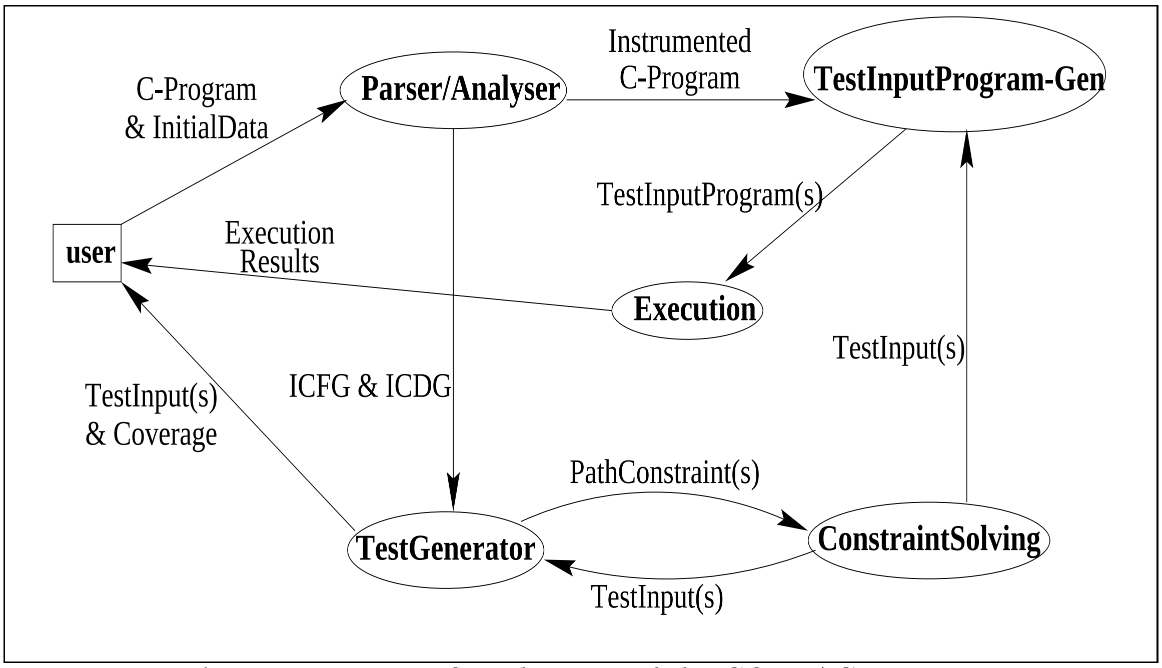


Figure 10.1: Dataflow diagram of the COTTAGE system

10.5 Implementation

A dataflow diagram of the COTTAGE system is given in Figure 10.1. A single source program and initial data —the name of a test procedure, the domains for the input variables, etc.— are input to the `Parser/Analyzer` component.

The `TestGenerator` component inputs an ICFG and its corresponding ICDG (Interprocedural Control Dependence Graph) from the `Parser/Analyzer` component. It tries to generate test cases for all nodes of the ICFG. This component implements thus our algorithm of test data generation for statement coverage. For each node of the ICFG, a path(s) reaching the node is dynamically constructed. A path constraint is then generated and passed to the `ConstraintSolving` component.

The `ConstraintSolving` component generates a test input following Algorithm 8.2, and passes it to the `TestInputProgram-Gen` component. Since our implementation (more specifically our constraint solving algorithm) is written in Java, the generated test input is thus a float solution, in Java, of the path constraint. Since the program under test is in C, we must also verify that the test input is a float solution, in C, of the path constraint. In other words, we must verify that the test input actually traverses the path on execution of the C program under test. This can be verified by running a corresponding instrumented C program with the test input. It should be noted however that a program can have different behaviors following the compilation options selected, or following the compiler selected. Therefore, we should rather instrument the executable (or binary) program to avoid such problems. An instrumentation at the level of the C program is acceptable in our case, because our focus is rather developing a prototype in order to validate the ideas developed in the thesis.

The `TestInputProgram-Gen` component therefore receives a test input and an instrumented C-program. It then generates a C program (referred to as a *TestInputProgram*) for running the instrumented C-program on the test input. Note that generating an instrumented C program for each generated test data is potentially costly when the program is big enough. In that case, it is better to generate only one instrumented C program with an execution for each test data. However, if the test program contains some `halt` statements, like `exit( ... )` in C, the sole instrumented program will halt immediately following an execution on a test data generated for a `halt` statement, making it impossible to execute tests for the other test data placed after this test data's execution. A possible solution is to generate (1) an instrumented program for a test data of a `halt` statement, and (2) a sole instrumented program for the other test data.

The `Execution` component first compiles, and then runs the TestInputProgram. Currently, the user carries out this component, and verifies the execution results for the *actual coverage* of the test inputs generated. However, the component should ideally be automated. And the execution results should be sent back to the `TestGenerator` component. Therefore, a test input not traversing the node will be rejected, and other test inputs will be generated for the node until obtaining a test input actually executing the node (a test case for the node).

Finally, the `TestGenerator` component reports all test inputs found, as well as the *predicted* statement coverage for all the nodes of the ICFG. The predicted coverage means the coverage, calculated without connection with the C language.

10.6 FindSolution function

In our implementation, given an epsilon interval-solution, we often simply select its middle point to check if it satisfies the path constraint. If so, it is actually a float solution (in Java), and it is returned as a predicted test case for the path. Experiments will show that this simple and efficient implementation turns out to be sufficient. Furthermore, a general *labeling* strategy, such as described in [50], can also be applied to the epsilon interval solution. The labeling is based on an uniform exploration of the domain. It is parameterized by the number of levels of exploration (labeling level, for short). Figure 10.2 illustrates this enumeration process on one variable. The numbers correspond to the levels. Using this labeling strategy, on our

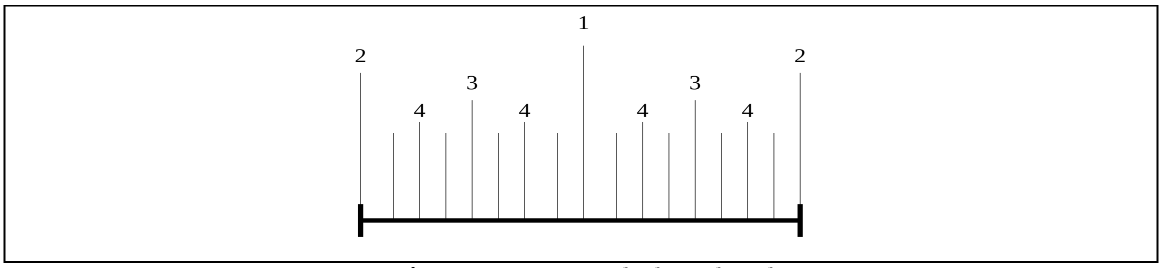


Figure 10.2: Labeling level

test cases (see Appendix A for details), we only observed little change (in time) to find a test case, compared with the default labeling level (one) where only a middle point is chosen. This is, without doubt, due to the following reasons:

- The size (epsilon) of interval solutions is usually chosen very small, such as *1e-16*, in our experiments.
- Path constraints are usually under-constrained, i.e. they have a lot of solutions. Therefore, when we obtain an epsilon interval solution, and one float solution is found in it, it is possible that such an interval solution also contain many other float solutions. This is a continuity argument that justifies that taking limited samples on a small interval as in `FindSolution`, will give a solution if there is one in the interval.

10.7 Parameters

Many parameters have been used by the COTTAGE system. Here are the main parameters:

- The size of *epsilon*: the size of interval solutions. The smaller the epsilon is, the more time is required to find an interval solution, but the resulting epsilon interval is more precise, and the `FindSolution` function has more chances to find a solution.
- *timeout*: a time limit for solving a path constraint. This allows the system to escape complex (and usually unsound) path constraints.
- *labeling-level*: the number of levels of labeling (as discussed above).

Other parameters allow the user to have more control over the system during test data generation such as:

- to fix a global timeout for the test data generation (covering the all-the-statements criteria).
- to print out all the generated paths reaching a node.
- to limit the number of paths reaching a node. Such a choice can be used when a lot of paths reach the timeout for solving their corresponding path constraint without generated test data.
- to make an exhaustive enumeration over the found interval solutions, i.e. an enumeration over all the float values in the interval, in order to find a test data. Such a choice can be used to examine the efficiency of the `FindSolution` function (Specification 8.2), because `FindSolution` selects only some points in the interval solution to verify if they are test data for a path. Note that for intervals near zero, such as $[0, 1e-16]$, it can take several hours (in our case, about 8 hours) without terminating even only an exhaustive enumeration (the time for path-feasibility verification is not included here). However, for intervals far from zero, an exhaustive enumeration is possible.
- to change the default branch taken at a node during the path generation (when the two branches have the same priority to be selected).
- to print out the generated test data for a node, together with its corresponding solved path constraint.

10.8 Function calls and Conservation of float solutions

Function calls to built-in functions such as *exp* (Euler number e raised to the power of a number), *log* (the natural logarithm of a number), *sin*, etc, are treated as basic operators, i.e. these function calls are not developed in the ICFG. The interval extensions of these functions were already available in [37] or constructed in our Java implementation. It is not clear whether the Java implementation in [37] of transcendental functions satisfies the property for conserving float solutions as stated in Section 7.2. We however checked this property experimentally. Note that to make our approach work, interval extensions for built-in functions, relations, and operators of the C programming language have been implemented in Java. There is thus a possibility that a float solution found by the constraint solving system is discarded as a solution by the TestInput part of the system (executing the instrumented C program).

Function calls to user-supplied functions where the code is not available (i.e. already separately compiled), are not currently handled by the system. However, a possible way to deal with user-supplied functions is to consider them as basic operators as is the above case with built-in functions. Interestingly, this way may also consolidate the objective of *modularity* in software development, especially in software testing. Once a functionality has been carefully tested, it can simply be replaced, during testing, by a "black box" accompanied by input/output specifications — that is, the code in the functionality is not required to be covered by the generated test data. As discussed above, treating user functions as basic operators, it is necessary to provide interval extensions for user functions during test data generation.

10.9 Type analysis and interval evaluation

Type analysis is important in our system to ensure the precision of interval evaluations. The purpose is that all sub-expressions involved in an expression are associated with a type. And for integer-typed sub-expressions, the result of an interval evaluation over such sub-expressions must be rounded to make it more precise. This is due to the fact that the used interval library makes no distinction between the interval evaluation of an integer-typed expression and that of a float-typed expression, and that all expressions are rather considered as float-typed. This can give rise to imprecise interval evaluations for integer-typed expressions.

Let us take an example to illustrate our above idea. Suppose an expression $(x + y) * 2.0$, where x and y are integer variables. Then the type of the whole expression is float, while sub-expression $x + y$ is of type integer. Suppose also that x is associated with the interval $[2, 4]$, and y with $[5, 6]$. Since we work with an interval library where the bounds of an interval are floats, $[2, 4]$ and $[5, 6]$ are in fact represented respectively by $[2.0, 4.0]$ and $[5.0, 6.0]$, and the bounds of their intervals are always rounded to integer values.

10.10 Soundness and Completeness of COTTAGE

Since the system verifies that the generated test inputs actually traverse the corresponding paths on execution of the C program under test, soundness of COTTAGE is ensured. However, three incompleteness cases may arise.

1. The constraint solver provides solution boxes covering all mathematical solutions and float Java solutions of a path constraint. But a float C solution could not be covered by the provided boxes.
2. An (efficient) implementation of `FindSolution` could fail to find a float C solution in an interval.
3. The system could fail to find a solution for a path constraint because of the time limit for solving a path constraint.

Case (1) is a limitation of our approach as we perform the search for a solution in a programming language independent from the program under test. This however provides more flexibility, and allows our system to handle testing with different programming languages. In practice, as illustrated by our experiments, the resulting theoretical limitation has little effect on the system. First, most solvable path constraints have many possible test cases. The objective is to find one test case per node (or branch), not to find all of them. Second, the implementation of the constraint solver is designed to limit this problem. For instance, the basic interval operations preserve the float solutions in Java.

Cases (2) and (3) are necessary limitations to ensure the efficiency of our system. However, as shown by the experiments, the choice of the epsilon value reduce this problem while increasing the overall efficiency of the system.

10.11 Extensions of COTTAGE

10.11.1 Handling other testing criteria

Currently, the COTTAGE system generates test data to reach a statement (statement coverage), or to traverse a path in the program (path coverage). An immediate extension of the system for branch coverage (test data generation for a branch) can be done easily based on our method for statement coverage.

Our method for path coverage can be used to generate test data for data-flow criteria in the same manner as the approach in [29]: Their path-coverage method [30] was exploited so as to deal with data-flow criteria. Note that in data-flow testing, a definition-use association is a basic bloc for any data-flow criteria. For instance, the *all-uses* criteria is satisfied if a (definition-clear) subpath from every definition to each of its uses, must be covered by the generated test data. The idea behind exercising a definition-use association is that we first select a program path(s) traversing such association; The next task consists of the generation of test data to execute the selected path.

Fault-based criteria such as proposed in [58], can also be integrated into our system. In that work, each type of faults is modeled by certain constraints. For example, assume that the possible range of indices for an array is between 1 and 10. Index-out-of-bound faults with that array can be modeled by two testing requirements such as: $I > 10$ and $I < 1$. Each of such requirements can be added to the path constraint during test data generation. Failure to generate test data for the augmented path constraint thus may be an indication of absence of such faults.

10.11.2 Test data generation for other imperative languages

Although our method of test data generation is suited for any imperative language $\mathcal{L}$, we currently handle only test programs written in *C*. To treat other imperative languages used for writing test programs, such as *Pascal*, we need to construct a parser to translate the Pascal program under test into our internal representation. This can be done in the same way as was constructed the system from our previous prototype [64]: a parser was constructed to translate the C program under test into the internal representation. A point not less important is that given a path constraint, since its solution space in Java and that in Pascal may be different, we should automatically construct an instrumented Pascal program for running each generated test data, as is the case with the current system (Java vs C). This way aims to ensure that the generated test data is actually a test case for a path when executing in a real environment.

10.11.3 Procedure calls

Procedure calls are currently handled by constructing the ICFG for the test procedure — the CFGs for all called procedures (except called built-in functions) are integrated into the CFG for the test procedure. We then try to generate test data for all the nodes of the ICFG. An advantage with such approach is that the set of generated test data covers not only all statements of the test procedure, but also all statements of the called procedures. Because the set of test data seems to be more prone to detect faults related to interactions among several procedures. For example, suppose that we have two procedures `A` and `B` such that

- in `A`, there are calls to `B`.
- In `B`, there is a branch point to check whether an exceptional case occurs. If so, a `halt` statement is raised, resulting in an unconditional halt of the main program.

Assume that we want to generate test data for `A`, and that a test data is found to reach the exceptional case. The tester is then informed that perhaps some faults may appear in `A`, or that at least, some treatments with such exceptional case would be integrated in `A` in order to avoid sudden halt during execution of `A`, caused by execution of `B`.

The above advantage comes however at the expense of the fact that we possibly have to deal with long paths during test data generation, when many procedure calls appear in the test procedure. Other disadvantages, as discussed in Section 10.8, involve the inability to deal with user procedures (the code is not available), as well as the possible loss of modularity in software testing. In the case where a called (user) procedure can be supplied with an interval extension, a possible extension of the system consists in leaving, to the tester, the choice between (1) treating the called procedure as an operator, and (2) extending it in the ICFG.

10.11.4 Handling multi-dimensional arrays

Although our current system is limited to one-dimensional arrays, our approach to dealing with one-dimensional arrays can naturally generalize to any-dimensional arrays. For example, an assignment like

```
  a[i][j] := exp;
```

can be transformed into the constraint

$\mathtt{na5}(a_{k'}, a_k, \overline{i}, \overline{j}, \overline{exp})$

where a_k is the last value instance of a, and k' the smallest integer not yet used for identifier a; $\overline{i}$, $\overline{j}$ and $\overline{exp}$ are defined by Definition 6.2. The constraint `na5` is defined and handled in the same manner with the constraint `na4` (Definition 6.3). Informally, the constraint $\mathtt{na5}(b, a, i, j, v)$ states that b is an array which is of the same size as a, and has the same component values, except that v is the value for its (i, j)-component.

10.11.5 Interval extensions

Interval extensions for some operators not yet handled by our system such as %, <<, >>, ... , should be constructed, thus allowing the handling of such operators.

One issue, that seems very important, is how to construct interval extensions for complex functions such as *sin*, so that they preserve *all float solutions.* The reasons for that issue are the following.

- The interval library [37] used by our implementation was aimed at preserving *all mathematical solutions.* Unfortunately, such aim was only achieved to a certain extent: interval extensions for arithmetic operations guarantee to preserve the mathematical solutions, but not for transcendental functions as was stated by the library.
- Even if the interval library is assured to preserve all mathematical solutions as intended, it is out of its scope to deal with float solutions, because the space of mathematical solutions for the path constraint is totally different from that of float solutions (as illustrated in Subsection 3.3.3).
- Since properties for real operations may not be preserved for their corresponding float operations, it appears to be difficult to construct interval extensions preserving all float solutions. We illustrate this idea by the example hereafter, inspired from [15].

The function $f(x) = log(1 - x)/x$ is monotone and continuous for $x < 1$. Its representation for $x \in [-1, 1]$ is given in Figure 10.3. Unfortunately, the shape of the same function for x near 0, when evaluated on the floats, is totally different with the former case, as illustrated in Figure 10.4. It is interesting to note that the “float” representation is no longer monotone, and seems unpredictable.

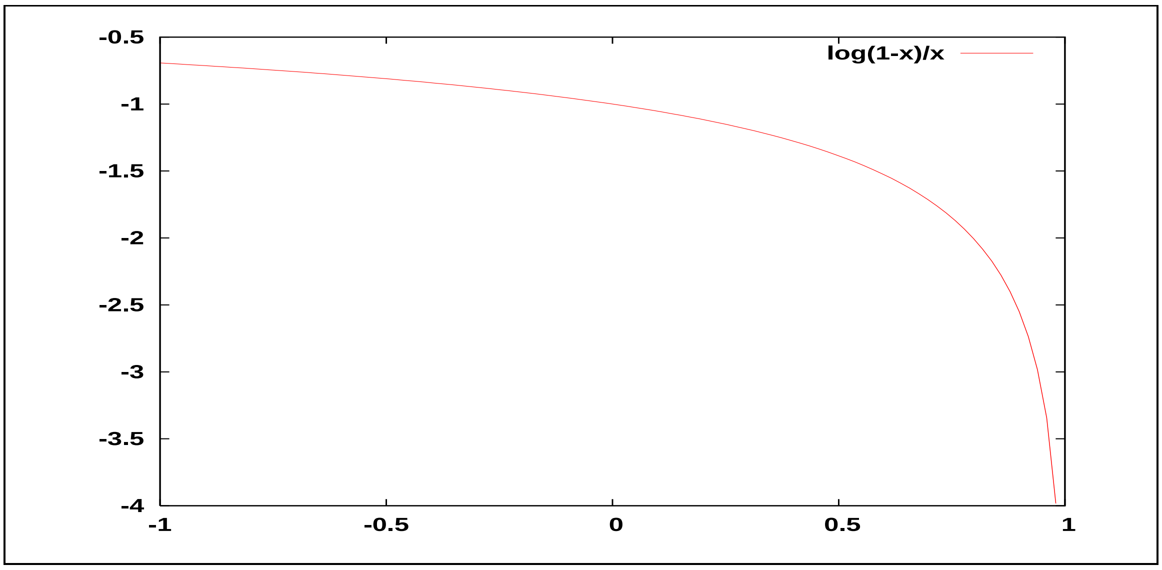


Figure 10.3: "Real" figure of $log(1-x)/x$

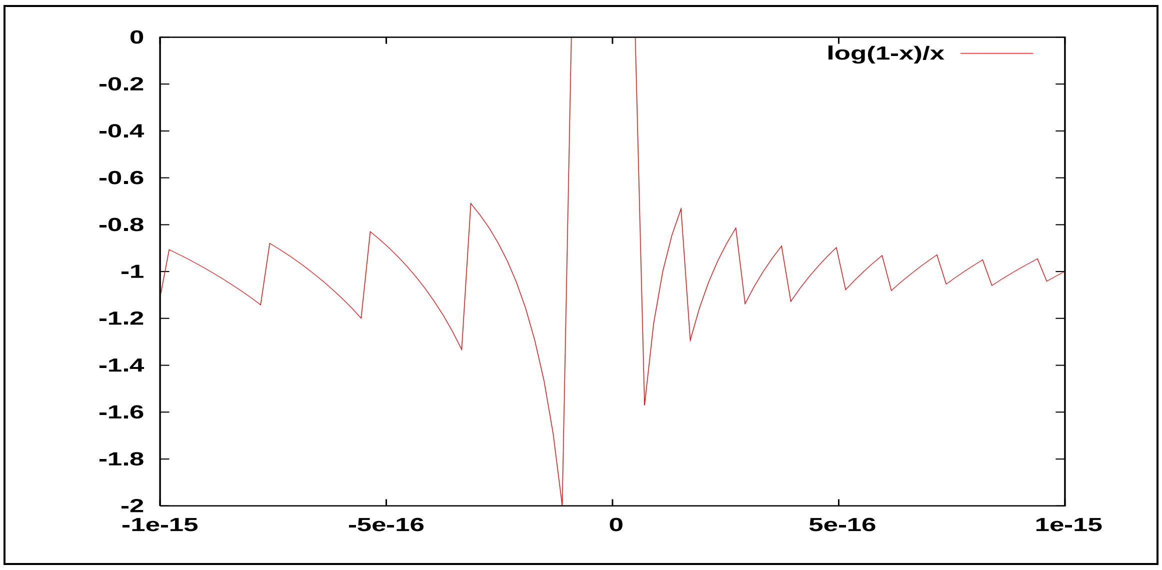


Figure 10.4: "Float" figure of $log(1-x)/x$

10.11.6 Handling pointers, aliases and dynamic data structures

Pointers (and dynamic data structures) are not currently handled by COTTAGE. The notion of pointers is related to the possibility of using the address of a memory zone (memory space) to access to its content, i.e. pointers are variables whose values are addresses. In C, the address of a variable or a dynamically allocated memory zone, can be assigned to a pointer variable. The main problem [25] in dealing with pointers by any static methods is that one must be able to compare the memory zones read or written by a dereferenced pointer (in C, if `p` is a pointer, then `*p` denotes its *dereference*). We illustrate that problem by the following example.

```
  int i = 0;
  int *p = &i;
  *p = 10;
  if (i > 5) ...
```

In this case, dereferenced pointer `*p` and variable `i` represent the same memory zone at some point in the program, and are called *aliases*. Hence, a definition for an alias is also a definition for all other ones. With the above example, any static method should be able to detect that the instructions in the `if` branch must be executed.

In a closely related work [25] with ours, only pointers that refer to a memory zone associated with a name (a variable declaration creates such a memory zone), were handled. To treat such pointers, a *points-to analysis* was used to statically calculate an over-estimated set of all possible pointing relations among the variables of the program. Pointers to anonym memory zones dynamically allocated by a call such as `malloc` in C, pointers to dynamic data structures (linked lists, trees, ...), as well as the arithmetic on pointers as defined by C, were not handled. Notice that if an input variable is a pointer to a dynamic data structure, automatic inference of the form of the data structure is not an easy task.

Chapter 11

Experiments

The objective of this chapter is to present our first experiments on the COTTAGE system. We first give a short description of the programs (benchmarks) used in our experiments. Our set of benchmarks consists of de facto programs used by the testing community, and of numerical programs drawn from a well-known numerical book [55]. Our test generation procedure and experimental results are then given. Many issues such as —efficiency, coverage and completeness, how to choose the parameters for our COTTAGE system, and comparison with related work— will finally be discussed.

11.1 Benchmarks

We have performed our experiments on a 900MHz UltraSparcIII+ machine, with the following programs.

- `NthRootBisect` (given in Figure 5.1) calculates the n-th root of a number using the Newton-Raphson method. This program uses integer and float variables, but no arrays nor procedures.
- `Sample` is the "sample" program with arrays, described in [21]. This program contains arrays whose values determine the control flow, as well as many data dependences.
- `Tritype` is a classical program [54], testing if three input integer numbers (representing the sides of a triangle) can actually form a triangle, and if so, determining the type of the triangle (equilateral, isosceles, . . .). It only contains integer variables, but has nested conditional instructions and many infeasible paths.
- `Proc` is the program `Program-1` in Figure 3.1, with nested procedure calls and a `halt` statement in a called procedure (which results in unconditional halt of the main program).
- `BSearch` [21, 26] is a binary search program involving arrays.
- The `CMichel` program is a small example from [50] with the instruction `if (16.0+x==16 && x>0) return 1; else return 0;`. Although the test is always mathematically false, there exist float values satisfying this test in C.

Table 11.1: Programs and Experimental results

Programs	Int.	Float	Array	Proc.	Timeout (sec.)	Epsilon	Nodes	Average (sec.)	Max (sec.)	Total (sec.)	Predicted Cover.	Actual Cover.
NthRootBisect	yes	yes	no	no	15	1e-16	10	0.04	0.2	0.4	100%	100%
Sample	yes	no	yes	no	15	1e-16	15	0.02	0.2	0.3	100%	100%
Tritype	yes	no	no	no	15	1e-16	24	0.01	0.1	0.3	100%	100%
Proc	yes	yes	yes	yes	15	1e-16	27	0.03	0.6	0.7	100%	100%
BSearch	yes	yes	yes	no	15	1e-16	10	0.02	0.1	0.2	100%	100%
CMichel	no	yes	no	no	15	1e-16	4	0.02	0.08	0.09	100%	100%
gaujac	yes	yes	yes	yes	15	1e-16	40	3.4	135.9	136.2	100%	100%
expint	yes	yes	no	no	15	1e-16	35	0.47	15.1	16.6	100%	100%
gamdev	yes	yes	yes	yes	15	1e-16	46	0.01	0.3	0.5	100%	100%
bessi	yes	yes	no	yes	15	1e-16	27	0.06	1.6	1.62	100%	100%
ei	yes	yes	no	no	15	1e-16	22	0.14	1.3	3	95%	95%
ei	yes	yes	no	no	15	1e-32	22	0.13	1.3	2.8	100%	100%
ei-dead	yes	yes	no	no	15	1e-32	27	0.14	1.9	3.8	92%	92%

The other programs are real scientific programs taken from [55], and involving math library functions (e.g. *log, exp, pow, sqrt*).

- The `gaujac` program calculates the Gauss-Jacobi integration formula. This program involves complex (nonlinear) expressions, 3 nested loops, arrays, and procedure calls.
- The `expint` program [55], which has also been experimented in [32], calculates exponential integrals, and involves nonlinear expressions and nested loops.
- The `gamdev` program [55] (also experimented in [32]) generates random numbers, and involves nonlinear expressions, nested loops, arrays, and procedure calls.
- The `bessi` program [55] calculates the modified Bessel functions, and involves procedure calls.
- The `ei` program [55] also calculates exponential integrals, and involves a very small constant (*1e-30*).
- Finally, `ei-dead` is the `ei` program extended with two unreachable statements, in and after the main loop.

The C code of these programs is given in Appendix B, as well as available at `www.info.ucl.ac.be/people/YDE/yde.html`. Table 11.1 also summarizes these programs.

11.2 Test generation procedure and experiments

Our test generation procedure consists in trying to generate a test case for each node of the ICFG, and then reporting the *predicted statement coverage* (the percentage of nodes for which the constraint solving algorithm found a float Java solution) and the *actual coverage* (the percentage of nodes for which the float Java solution is a test case of the C program). Note that when a test data is generated for a node, we also obtain a path traversing the node. Hence, all other nodes involved in the path are then marked as covered by the same test data.

Our experimental results are given in Table 11.1. For each program, Table 11.1 lists the values of the following parameters: *epsilon* (the size of the interval solutions) , *timeout* (timeout for solving a path constraint), *Nodes* (the number of nodes of the corresponding ICFG), *Average* (the average time in seconds spent on a node), *Max* (the maximum time in seconds spent on a node), *Total* (the total time in seconds to generate test cases for all the nodes), *Predicted-Coverage* (the predicted statement coverage), and *Actual-Coverage* (the actual statement coverage). Note that the value of parameter *labeling-level*, used in these experiments, is one by default.

For more detailed information on our experimental results, refer to Appendix A.

To determine whether the use of filtering during the selection of paths in Algorithm 9.1 (as indicated in Algorithm 9.1) or during the path constraint solving in Algorithm 8.2, is actually effective, we also carried out the same experiments where such filtering operations were deactivated. These experimental results will be given in Sections 11.10 and 11.11.

11.3 Efficiency

Even with the complex scientific programs, `gaujac`, `expint`, `gamdev`, `bessi` and `ei`, the time performance indicates that our method is practical. It is difficult to provide a time complexity analysis as the general problem of solving a set of constraints is NP-hard. Efficiency should therefore be measured on specific classes of problems. Moreover, choosing a “good” path reaching a node, where one quickly gets a test case, is another problem. Indeed, at a decision node, where its two successors have the same priority to be chosen during the path construction, the choice of the next successor has a great influence on the time complexity. Taking the `gaujac` program as an example, its related results reported in Table 11.1 correspond to the default behavior of our test data generator. However, when we change the branch taken by default at a node, the time needed to cover all the nodes is only 3.6 seconds! The speed-up here is thus around 38.7. Note that our implementation allows us to observe all paths reaching a node during the path construction, as well as to specify at specific node, the strategy for the path generation.

11.4 Coverage and completeness

For programs without dead code, the COTTAGE system is able to achieve 100% coverage on all the experimented programs, even the complex scientific ones. On all our experiments, the actual coverage is also the predicted coverage. This illustrates that the completeness issue raised in the previous chapter is, in practice, not really problematic. The `CMichel` example illustrates that the constraint solver is also able to find non mathematical solutions (here it found the value *x=1.3322...E-15* for the test). The coverage of the `ei-dead` example is only 92% because 2 nodes are unreachable; they have been detected by the system.

The achieved coverage justifies our approach to use an implementation language (Java) different from the language of the tested programs (C). This generic approach allows us to use the COTTAGE system for testing programs in other programming languages.

11.5 Procedures

Our interprocedural control dependence analysis, as described in [64], enables the system to handle procedures with greater precision than the classical intraprocedural analysis. In the `proc` example, the *halt* statement in a nested procedure is handled without any problem.

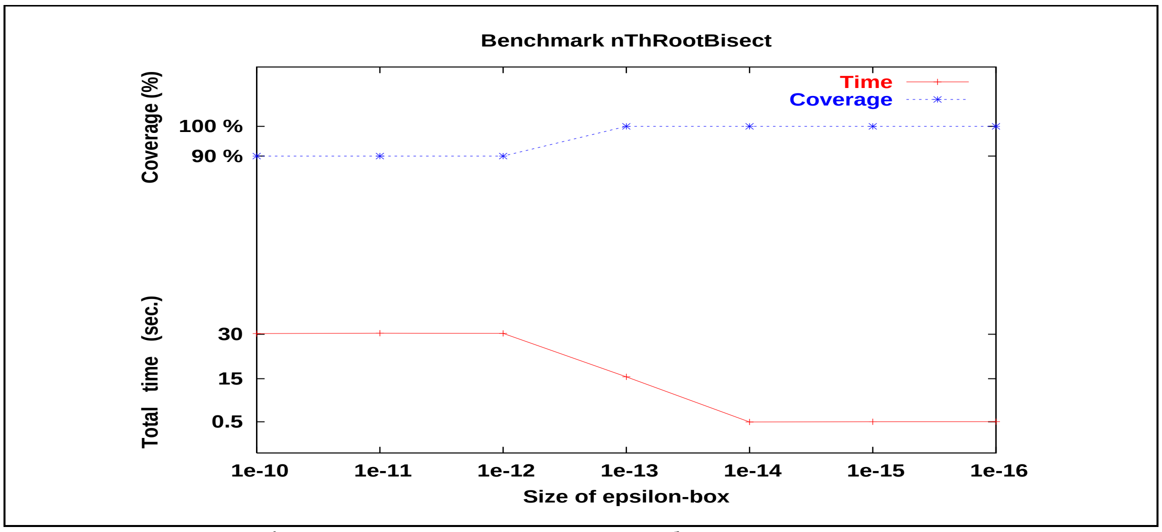


Figure 11.1: Experiments with `nThRootBisect`

11.6 Choosing the parameters

For some programs, the default value of the parameters (epsilon and timeout) has to be adapted to achieve completeness. The epsilon value may influence not only the coverage, but also the execution time, as illustrated in Figure 11.1 on the `nThRootBisect` example. With small epsilon values, our implementation of `FindSolution` with the default *labeling-level* (choosing the middle point) has more chance to find a float Java solution. With larger epsilon values, the system could generate many interval solutions before `FindSolution` finds a float Java solution, increasing thus the execution time, or reducing the coverage. More sophisticated implementation of `FindSolution` could be designed, but this is not a central point given the efficiency of the constraint solver for small epsilon values.

The `ei` program, involving a *1e-30* constant, illustrates the necessity to choose the epsilon value according to the constants used in the program. The default *1e-16* epsilon value achieves only 95% coverage while a *1e-32* value achieves 100% coverage. The execution time is even better (pruning is slightly more efficient).

For some programs such as `expint`, increasing the timeout parameter can affect the execution time. A too small timeout could also reduce the coverage.

It should be noted that the values chosen for the epsilon parameter are fix in our experiments. We could however choose a variable epsilon value, representing the following possibilities:

- The epsilon value corresponds to a fix number of floats.
- The epsilon value is proportional to the size of the initial box given to the path constraint.

11.7 Complex paths

In the `expint` and `ei` programs, a constant `MAXIT` is used to indicate the maximum number of iterations such programs are allowed in order to converge to a solution; they raise an error, by `exit(1)`, if no solution is found after `MAXIT` iterations. Our system is of course not able to find a test case achieving these error statements with the original value of `MAXIT` (100) as the number and the complexity of the path constraints is too high. In the experiments reported in Table 11.1, the value of `MAXIT` is 10 for `expint`, and 17 for `ei`.

11.8 Initial domain values

The choice of the initial domains of the input variables may influence the coverage and the efficiency of the system. A too small initial domain could not cover some of the nodes. A large domain may increase the computation time. In our experiments, we choose large initial domains to favor coverage. In the `expint` program for instance, the initial domains are $n = [1, 30]$ and $x = [-1000, 1000]$. For `ei`, we set $x = [-1, 10000]$ because $x < 0$ is an error case.

11.9 Over-estimation of interval solutions

Interval programming suffers from the well-known problem of multi-occurrences of the same variables in an expression. Consider, for example, the function $f(x) = x - x$, which always returns 0. However, its natural interval extension $F(X)$ is much less precise. For example, with the used interval library, $F([0, 1])$ returns the interval $[-1.0000000000000002, 1.0000000000000002]$, which contains much noise because we ideally hope for the interval $[0, 0]$. More surprisingly, a constraint such as $c(x) := x - x \neq 0$, has no float solutions while any point interval such as $[1, 1]$ is its interval solution, because $F([1, 1])$ returns the interval $[-4.9e - 324, 4.9E - 324]$.

As a consequence, the `ei-dead` program generates 246 interval solutions (see columns 13, 14 and 15 in Figure A.2), out of which only 6 interval solutions actually contain float solutions (found by `FindSolution`). The other 240 interval solutions contain no float solution (following an exhaustive enumeration). Notice however that the corresponding `ei` program generates only 6 interval solutions, all of which contain float solutions (found by `FindSolution`). We can conclude that the two unreachable instructions inserted in `ei-dead` are the cause of such 240

Table 11.2: Experimental results without the filtering in path generation

Programs	#path with filtering	#path without filtering	Error	Timeout (sec.)	Epsilon	Nodes	Total (sec.)	Predicted Cover.
NthRootBisect	5	5	no	15	1e-16	10	0.4	100%
Sample	2	950	StackOverflow	15	1e-16	15	*	*
Tritype	10	39	no	15	1e-16	24	1.5	100%
Proc	4	4	no	15	1e-16	27	0.7	100%
BSearch	3	8	no	15	1e-16	10	0.3	100%
CMichel	2	2	no	15	1e-16	4	0.09	100%
gaujac	12	*	out-of-bounds	15	1e-16	40	*	*
expint	10	30	no	15	1e-16	35	16.6	100%
gamdev	6	*	out-of-bounds	15	1e-16	46	*	*
bessi	22	22	no	15	1e-16	27	1.62	100%
ei	22	453	StackOverflow	15	1e-32	22	*	*
ei-dead	39	340	StackOverflow	15	1e-32	27	*	*

bad interval solutions. The purpose of the two unreachable instructions is: all the path constraints going through such instructions must involve two conditions not mutually satisfiable such as $exp_1 \leq exp_2$ and $exp_1 > exp_2$. Then these path constraints will have no float solution (i.e. the corresponding paths are infeasible). However such constraints can have interval solutions. For example, the constraint $x + y \leq 1$ & $x + y > 1$ has no float solution, while it can have $x = [0.5, 0.5]$ and $y = [0.5, 0.5]$ as an interval solution, because interval evaluation of $x + y$ returns the interval $[0.9999999999999999, 1.0000000000000002]$. Likewise, the `CMichel` program generates 3 interval solutions, one of which actually contains no float solution after an exhaustive enumeration.

The relative weakness of interval solutions in the above extreme situations does not influence, however, the feasibility of many software based on interval programming such as Numerica [36], as well as that of our system as indicated by our experimental results.

11.10 Influence of filtering on the path generation

Table 11.2 presents our experimental results where the filtering was deactivated during the path generation in Algorithm 9.1 (i.e. the line as indicated in Algorithm 9.1 was removed). The two columns named *#path with filtering* (the total paths generated when the filtering is active in Algorithm 9.1) and *#path without filtering* (the total paths generated in this case, i.e. without the filtering) aim at showing the actual influence of filtering on the path generation. The *Error* column aims to explain the programs with some of their columns marked by an * (i.e. unknown), whose experiments were halted due to the following reasons:

- Too many paths were generated, resulting in *StackOverflow* exceptions.
- Invalid paths were generated due to *index-out-of-bounds* exceptions with arrays.

We now take an example to illustrate the latter case. In the `gaujac` program, let's consider the following `else` branch

```
  for (i=1;i<=N;++i) {
```

Table 11.3: Experimental results without the filtering in constraint solving

Programs	Timeout (sec.)	Epsilon	Nodes	With filtering Total (sec.)	With filtering Predicted Cover.	Without filtering Total (sec.)	Without filtering Predicted Cover.	Without filtering Error
NthRootBisect	15	1e-16	10	0.4	100%	300.5	45%	
Sample	15	1e-16	15	0.3	100%	0.2	100%	
Tritype	15	1e-16	24	0.3	100%	0.3	100%	
Proc	15	1e-16	27	0.7	100%	0.14	100%	
BSearch	15	1e-16	10	0.2	100%	0.1	100%	
CMichel	15	1e-16	4	0.09	100%	0.09	100%	
gaujac	15	1e-16	40	136.2	100%	31.1	100%	
expint	15	1e-16	35	16.6	100%	300.4	13%	
gamdev	15	1e-16	46	0.5	100%	*	*	StackOverflow
bessi	15	1e-16	27	1.62	100%	300.4	3%	
ei	15	1e-32	22	2.8	100%	300.5	91%	
ei-dead	15	1e-32	27	3.8	92%	300.6	14%	

```
if (i == 1) { ... }
...
else {
  z=3.0*x[i-2]-3.0*x[i-3]+x[i-4];
}
...
```

This branch can only be reached if `i >= 4`. But since no filtering was used during the selection of paths, this branch was chosen in the first iteration of the loop (corresponding to `i = 1`). Therefore, an *index-out-of-bounds* on arrays was generated by the system, because `x[i-4]` becomes `x[-3]`, for instance.

For other programs such as `tritype`, more infeasible paths were generated, with a consequence that the *Total* time was increased considerably (1.5 seconds vs. 0.3 seconds in Table 11.1).

The experiments as reported in Table 11.2 confirm thus the necessity of using filtering during the generation of the paths in Algorithm 9.1.

11.11 Influence of filtering on the constraint solving

Table 11.3 presents our experimental results where the filtering was deactivated during the path constraint solving in Algorithm 8.2. The columns *With filtering* (related to Table 11.1) and *Without filtering* (this case) aim to show the influence of filtering on the path constraint solving.

For the half of the programs, the *Total* time is slightly better. This is perhaps due to the fact that when the path constraints have a lot of solutions, the filtering could be a waste of time. Moreover, in such cases, many test data could be generated by chance with the `QuickFindSolution` function. For other programs, although we fixed the timeout for the test data generation to five minutes, the coverage results were too bad compared with the presence of filtering. Regarding the `gamdev` program, even a StackOverflow exception was raised.

Table 11.4: A summary of test data generators

Methods	Ref.	Int.	Float	Arrays	Proc.	Path Coverage	Statement Coverage
Consistency	*this*	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*
Testgen	[47]	yes	yes	yes	yes	yes	yes
Relaxation	[32]	yes	yes	yes	yes	yes	yes
InKa	[26]	yes	no	yes	yes	no	yes
Genetic	[54]	yes	yes	yes	no	no	yes
Symbolic	[10]	yes	yes	yes[1]	yes	yes	no

[1] Array references depending on input variables are not handled

In conclusion, the use of filtering in the path constraint solving is necessary in the general case.

11.12 Comparison with existing methods

Table 11.4 summarizes the existing methods with functionalities close to our method (first line in the table). Two other methods offer the same functionalities, [47] and [32]. As in these methods, our prototype is able to achieve 100% coverage on the examples, but our set of examples contains more complex programs. It is difficult to compare the efficiency of the different methods because efficiency information is sometimes partial or missing. When this information is available, the measures can be incomparable (number of iterations versus execution time versus theoretical complexity). When it is comparable, one should consider the differences in the underlying hardware.

In [32], an execution time of 98 and 42 seconds (Windows NT, 400MHz Pentium II) is reported to find test data for two branches of the benchmark `expint`, and an execution time of 117.4 seconds to find test data for one branch of the benchmark `gamdev`. For the `expint` program, our system generates 10 path constraints to achieve a full coverage in 16.6 seconds; and for the `gamdev` program, our system generates 6 path constraints to achieve a full coverage in 0.5 seconds.

The experimented scientific programs taken from [55] are so far the most complex (in terms of the complexity of expressions) of our examples. In the literature, we rarely find such complex examples used by other methods. We do not claim however that our method is the best. Rather, it can be used as an alternative or a complement with other methods in test data generation. More importantly, our system strengthens the possibility of applying constraint programming for test data generation as stated in [26].

Part IV

Conclusion and Future Work

Chapter 12

Conclusion and Future Work

This chapter recalls the main contributions of this thesis, presents our conclusions from such contributions, and outlines possible directions for future work.

12.1 Conclusion

This work first described our consistency approach to the problem of structural test data generation, namely the generation of test data for imperative programs with float, integer and boolean variables, as well as procedure calls and arrays. Test programs (with procedure calls) are represented by an interprocedural control flow graph (ICFG). The testing criteria (path coverage, statement coverage, and branch coverage) are then defined in terms of the ICFG. Our purpose was thus to generate test data that will cause the program to traverse a specified path, node, or branch of the ICFG.

For path coverage, the search for test data is reduced to the solving of path constraints. Such a solving is based on consistency techniques, aiming at reducing the domains of the variables. A main originality of our method is thus a constraint solver dealing with float, integer and boolean variables, thereby suitable for test data generation. Included in that solver is a new interval logic framework able to deal with interval constraints involving integer, boolean and float variables, as well as the logical operators *and*, *or* and *not*. Conservation of float solutions is also a central point in building a "conservative" solver, where its filtering algorithms ensure not losing any float solutions. To reach such aim, we used the natural interval extension, such as also proposed in [50]. Moreover, as an extension of some results in [50], we proposed further theoretical ideas in order to build an interval library ensuring the conservation of float solutions. For example, in the case where only the rounding mode *near* is used such as in Java, we showed interval-extension formulas, as well as conditions that must be satisfied for such conservation. It should be noticed however that in practice, it is sometimes difficult to obtain a total conservation of float solutions due to the following reasons:

- The rounding of floating-point operations may be inexact.

- Since the properties of real operations are not preserved for their corresponding floating-point operations, we may be incapable to predict the variation of floating-point operations, as illustrated in Figure 10.4, for the calculation of their interval extensions so as to preserve all float solutions.

For statement coverage, suitable paths reaching the specified node are dynamically constructed. The search is guided by the interprocedural control dependence graph, as well as pruned by our e-box consistency filter. When such a path is found, our algorithm for path coverage is then applied. A dynamic approach to statement coverage was also proposed, by combining random test data generation, program execution and our path coverage method — but has not yet been implemented. An implementation for this dynamic approach should be investigated in future work.

Note again that our algorithms of test data generation for both statement coverage and path coverage, are incomplete. This is due to the undecidability of determining whether a node (or a path) is executable in the general case. As a consequence, our algorithms may loop to find test data for certain program elements of the test program.

We then presented our COTTAGE system, a 13,000 Java lines software, implementing our method, for test data generation of C programs. Various experiments, including complex programs (in terms of nonlinear expressions) taken from the numerical computing book [55], have been reported. These experiments showed the coverage of our system, as well as its versatility and flexibility to different classes of problems (integer and/or float variables; arrays, procedures, path coverage, statement coverage). They demonstrate the feasibility of the method, its efficiency and its potential to handle complex programs.

Our constraint solver could be combined with existing approaches based on dynamic methods (e.g. [32, 54]), especially when searching a test data exercising a specified statement of the program.

12.2 Future Work

Below are some directions for future work related to our current method. Note that possible extensions for our COTTAGE system were already discussed in greater detail in Section 10.11.

- The development of different strategies for the `FindSolution` function, such as using `local search` in epsilon interval solutions, should be investigated.
- Extension of the considered subset of C, such as pointers, is necessary to the testing of more realistic programs. One possibility may be to extend the partial work on pointers in [25]. Beside pointers, more research should be focused on dealing with dynamic data structures (graphs, trees, ...). Many problems remain to be solved, such as the automatic determination of shape for dynamic data structures when they are used as input.
- The possibility of error detection will also be considered, by adding new kinds of constraints modeling error conditions such as in [58].

- Data flow testing and its criteria can be incorporated in our work by inspiring ideas from [29], where paths covering data flow dependences are first constructed, and then used by our path coverage method to generate test data.

Notice that in this thesis, we focus our attention to the generation of test data for imperative programs. Therefore, all of our discussions on testing, testing criteria, and test data generation to cover some criteria, were limited to (traditional) imperative paradigm. However, some other programming paradigms (object-oriented, declarative, functional, ...) exist in parallel. And testing in the context of each paradigm should be a specific subject on its own, because of underlying differences among them. Even so, we present below some links of our work and traditional testing with object-oriented paradigm (more specifically *object-oriented testing*), which becomes more and more used in practice, such as Java, C++, etc.

Testing with object-oriented programs faces new concepts such as polymorphism, inheritance and encapsulation, which can present new kinds of errors and difficulties [4] that do not exist in traditional testing. This means that new criteria should be derived to help decide when we have done enough testing. However, traditional methods can still be adapted to a certain extent for object-oriented testing. For example, as reported in [4], a technique for testing all intra-class data flows was developed based on ideas from data-flow testing, where it offers a systematic means to exercise all possible data flows over all possible method activation sequences. As another example, an inter-method flow graph constructed during class testing [5] can be viewed as an interprocedural flow graph in our work, which in turn may be used by classical structural criteria for generating test data. For general discussions on testing of object-oriented software and its characteristics compared with traditional testing example, see [43, 5]. Notice that such work rather focused on black-box testing, which does not use the program's code to select tests. In other words, selection of tests is based on specifications, or from state models (state graphs) constructed from specifications. Since many software systems are not totally formally specified or not even informally specified, these methods may not be applicable. Even when applicable, specification-based testing may not be able to detect all faults caused by implementation details, as is the case in traditional testing.

White-box approaches to testing object-oriented software include the following, to name only a few. (1) [35] proposes a method for performing data flow testing on classes. Sequences of methods (can be called in any order from outside the class) that should be executed to test a class, are calculated by using possible data flow interactions between these methods. (2) A class of adequacy criteria used to test the behavior of exception-handling constructs in Java programs, is described in [60]. The approach presents techniques for generating testing requirements for the criteria using its control flow representations. It also gives a methodology for applying the criteria to unit and integration testing of programs containing exception-handling constructs. (3) [9] can be viewed as an extension of [35], where data flow analysis, symbolic execution and automated deduction are combined to generate method sequences for structural testing of classes. In this work, test case generation is realized to a certain extent, when instance variables are scalar and symbolic execution can be completed successfully.

The above approaches to object-oriented testing provide a basis for testing, but do not totally attack the problem of automatic test data generation. [9] seems to be an important step-stone in dealing with that problem, but further work still remains, for example, for the case where instance variables are objects. Moreover, dealing with input variables being objects, is likely the similar situation in traditional testing where input variables are (pointers to) dynamic data structures.

Appendix A

Detailed Experimental Results

In this appendix, we present our experimental results in more detail. They are divided into two cases respectively corresponding to Figure A.1 and Figure A.2: The first case represents the normal operation of Algorithm 8.2 while the second case removes the three lines indicated in the same algorithm, where exists a call to `QuickFindSolution`. We will first explain our experiments presented in Figure A.1.

A.1 With `QuickFindSolution`

In Figure A.1, the number of vertices (or nodes) of the ICFG is represented in column numbered 1. The parameters for each experiment —the 0 level, the size of interval solutions (epsilon), the timeout for constraint solving— are represented in columns numbered 2, 3 and 4. Note that in column 3 (epsilon), -16 stands for $1e - 16$, and so on.

Information on the path constraints created during the generation of test data for all nodes of the ICFG, is given in the columns numbered 5, 6, 7, 8 and 9. We have the following equality:
$5 = 6 + 7 + 8 + 9$
That is, the total generated path constraints (in column 5) are divided into four categories:

- 6: the path constraints that were successfully solved to provide a test data,
- 7: the path constraints that were interrupted during their solving, due to time-out,
- 8: the path constraints having no interval solutions,
- 9: the path constraints having interval solutions, but no float solution is found.

If a path constraint belongs to category 8, it may possibly be infeasible. Because it may be that float solutions actually exist, but they are not preserved during filtering. Note that as discussed earlier, our filtering algorithms were designed to preserve float solutions by using natural interval extensions. In the case where all interval extensions for the basic arithmetic operators, as well as for functions such as *sin* treated as basic operators, are guaranteed to preserve all float solutions, our filtering algorithms will not lose any float solutions. As a consequence, the path constraint

WITH QuickFindSolution

1: Total vertices
2: Labelling level
3: epsilon
4: timeout for constraint solving (sec.)
5: Total path constraints generated
6: Total solved path constraints
7: Total path constraints with timeout
8: Total path constraints with no interval solution
9: Total path constraints with interval solutions but no float solution
10: Total calls to QuickFindSolution
11: Total failed QuickFindSolution(s)
12: Total winning QuickFindSolution(s)
13: Total calls to FindSolution
14: Total failed FindSolution(s)
15: Total winning FindSolution(s)
16: Total solution-tests
17: The time needed to cover all ICFG vertices (sec.)
18: The maximum time spent on a vertex (sec.)
19: The average time spent on a vertex (sec.)
20: Coverage (%)

		Parameters			Path constraints					QuickFindSolution			FindSolution			Tests	Exec. time			
Function	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
NthRootBisect	10	1	-16	15	5	3	0	2	0	14	11	3	0	0	0	14	0.4	0.2	0.040	100
NthRootBisect	10	1	-30	15	5	3	0	2	0	14	11	3	0	0	0	14	0.4	0.2	0.040	100
sample	15	1	-16	15	2	2	0	0	0	0	0	0	2	0	2	2	0.3	0.2	0.020	100
tritype	24	1	-16	15	10	8	0	2	0	42	37	5	3	0	3	45	0.3	0.1	0.013	100
proc	27	1	-16	15	4	4	0	0	0	5	4	1	453	450	3	458	0.7	0.6	0.026	100
proc	27	2	-16	15	4	4	0	0	0	4	3	1	3	0	3	7	0.12	0.05	0.004	100
bsearch	10	1	-16	15	3	3	0	0	0	0	0	0	3	0	3	3	0.2	0.1	0.020	100
cMichel	4	1	-16	15	2	2	0	0	0	4	2	2	1	1	0	5	0.09	0.08	0.023	100
gaujac	40	1	-16	15	12	2	9	1	0	1	0	1	4277	4276	1	4278	136.2	135.9	3.405	100
gaujac	40	5	-16	15	12	2	9	1	0	1	0	1	3460	3459	1	28357	136.4	136	3.410	100
gaujac	40	10	-16	15	12	2	9	1	0	1	0	1	782	781	1	108815	136.4	136.1	3.410	100
expint	35	1	-16	15	10	8	1	1	0	80	72	8	0	0	0	80	16.6	15.1	0.474	100
ei	22	1	-32	15	22	6	0	16	0	28	22	6	0	0	0	28	1.9	0.9	0.086	100
eiDead	27	1	-32	15	39	6	0	17	16	399	393	6	240	240	0	639	3.8	1.9	0.141	92
eiDead	27	5	-32	15	39	6	0	17	16	399	393	6	240	240	0	1311	3.9	2	0.144	92
gamdev	46	1	-16	15	6	5	0	1	0	5	3	2	3	0	3	8	0.5	0.3	0.011	100
bessi	27	1	-16	15	22	4	0	18	0	25	22	3	1	0	1	26	1.62	1.6	0.060	100

Figure A.1: With QuickFindSolution

having no interval solutions is surely infeasible. But in reality, since the interval library used in our implementation was designed mainly to preserve mathematical solutions, our only conclusion is that the path constraint may be infeasible.

If a path constraint belongs to category 9, one of the following conclusions can be drawn. (1) The path constraint has interval solutions, but they are all over-estimated due to the well-known problem in interval evaluation (multiple occurrences of the same variables) already mentioned in Section 11.9; This means that no float solutions actually exist in them. (2) The path constraint has interval solutions containing float solutions, but function `FindSolution` cannot find them. If so, we can, for example, increase the value of the labeling level, or make an exhaustive enumeration on interval solutions. The latter case can become problematic for interval solutions near zero. Indeed, in an experiment, it took us more than five hours without being able to finish enumerating all floats in the interval $[0, 1e - 16]$. Of course, the five hours here do not include the time to test whether floating-points are actually float solutions. If such tests were carried out, many days would be required for testing such floats generated during these five hours. This means that exhaustive enumeration may be impossible for certain interval solutions. As also reported in [15], the interval $[1, 1.000001]$, although of very small size, contains not less than four billions and half of floats. Furthermore, it is important to note that the number of floats near zero increases far more considerably, i.e. the region near zero contains the most of floats. And the farther we go from zero, the lesser the number of floats is. Given the above difficulty with exhaustive enumerations, in the future work, we should envision more intelligent approaches to explore interval solutions.

The columns 10, 11 and 12 represent calls to function `QuickFindSolution`, integrated in Algorithm 8.2 and specified in Specification 8.1. Recall that the aim of `QuickFindSolution` is to test whether the middle point in the current domains box for all input variables is a test data. If so, no more filtering (may be very expensive) is needed to converge to a domains box of size epsilon (an epsilon interval solution), from which the finding of test data actually takes place. We have the following equality:

$10 = 11 + 12$

This means that the total calls to `QuickFindSolution` (10) are divided into two categories: 11 (the calls to `QuickFindSolution` where no test data is found) and 12 (the calls to `QuickFindSolution` with test data successfully generated).
The columns 13, 14 and 15 represent calls to function `FindSolution`, also integrated in Algorithm 8.2 but specified in Specification 8.2. The total calls to `FindSolution` given in column 13, are also divided into two parts given in columns 14 and 15, as the above case with `QuickFindSolution`. The purpose of the columns from 10 to 15 is to examine the "efficiency" of both `QuickFindSolution` and `FindSolution`. Note that the total calls to `FindSolution` is equal to the total interval solutions generated; Column 16 represents all solution tests done in `QuickFindSolution` and `FindSolution`, where each test verifies whether an assignment of values to the input variables is a float solution. Given the fact that the integration of `QuickFindSolution` in Algorithm 8.2 aims at the possibility of quickly finding test data for a path constraint, the results involving `QuickFindSolution` really play less important role, and they are there rather for our statistical purposes. On the contrary, it is more interesting to analyze the results with `FindSolution` in columns 13, 14 and 15.

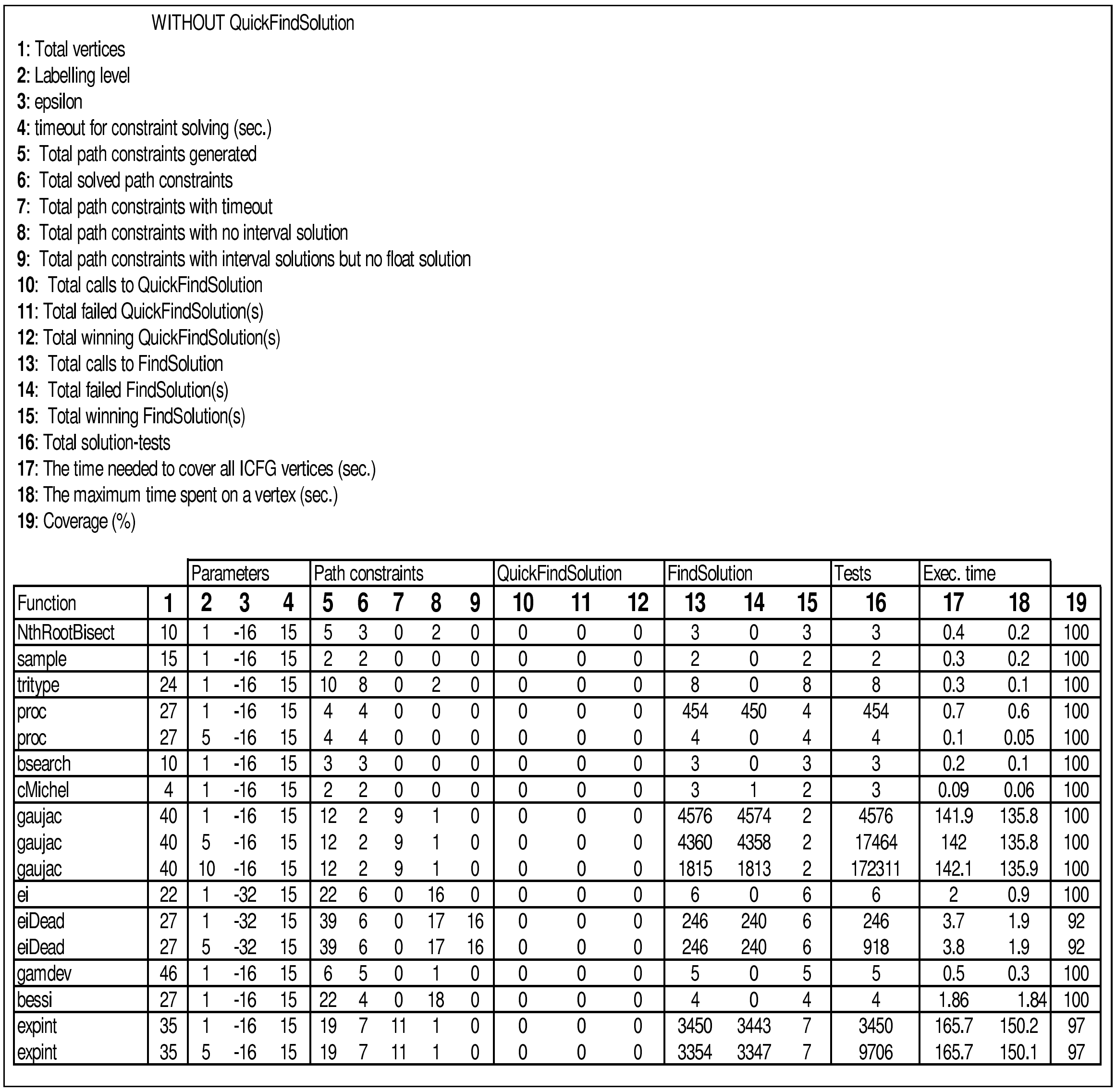
- For the `NthRootBisect` benchmark, all test data were found by `QuickFindSolution`. That is, no epsilon interval solutions were actually generated. This illustrates the possible benefit of integrating `QuickFindSolution` in Algorithm 8.2: Reaching epsilon interval solutions first prior to finding test data, may be more time-consuming than having `QuickFindSolution` integrated, as will also be confirmed when we bring into comparison Figure A.1 and Figure A.2.
- For the `sample` benchmark, contrary to the above case, all test data were found by `FindSolution`, i.e. test data were only generated once each interval solution had been found. We may wonder why no calls to `QuickFindSolution` were carried out. This is because `QuickFindSolution` is called only if the path constraint is completely expressed in terms of input variables. However, it is not the case, for example, if the path constraint involves arrays dependent on input variables.

- For the `tritype` benchmark, test data were however generated from both `QuickFindSolution` and `FindSolution`.
- For the `proc` benchmark, when the labeling level is one by default, out of 453 interval solutions, only 3 were successful with `FindSolution` in searching a test data. When we increased the labeling level to 2, test data were immediately found in all the three interval solutions generated. This illustrates the possible advantage of increasing the labeling level in other situations. The time performance is also far better for the latter case. For instance, the total time to generate test data for all ICFG vertices (column 17) is 0.12 (sec.), compared with 0.7 for the former case. Notice that the higher the value of the labeling level is, the more time is needed to find test data in an interval solution, but the more chance we have in obtaining test data. If the value of the labeling level is lower, more interval solutions can be generated, and thus the more time is required to generate them.
- For the benchmarks `cMichel`, `gaujac` and `eiDead`, the results with `FindSolution` were bad. Because the situation didn't change too much although our increasing in the labeling level. We wondered whether this is due to the labeling of `FindSolution` or the over-estimation of interval solutions (discussed in Section 11.9). As was also discussed in Section 11.9, for `cMichel` and `eiDead`, we successfully made an exhaustive enumeration on their interval solutions. Indeed, such interval solutions were all over-estimated (i.e. no float solutions exist in them). Unfortunately, an exhaustive enumeration on interval solutions of `gaujac` was not possible. However, in analyzing `gaujac`, we can easily see the problem of multi-occurrences of the same variables, the cause of the over-estimation of interval solutions.

The columns 17, 18 and 19 report the time performance of our experiments while column 20 reports the predicted coverage of all the nodes of the ICFG. Note that the actual coverage was the same as the predicted coverage in all of our experiments. That's why it doesn't occur in Figure A.1. Except for the `eiDead` program, where we inserted two unreachable lines of code (dead code), a 100% coverage was attained for all other programs. Moreover, the dead code introduced in `eiDead` was successfully detected. The time performance (execution times) was also reasonable and practical. After all, the time performance as well as the coverage, are actually a function of many factors such as the machine, the initial domains for the input variables, the parameters for each experiment (columns 2, 3 and 4), . . . Since the time performance is relative, it should be understood as indicative. Likewise, since the coverage is relative, a bad choice of some of the above factors may results in a poor coverage.

A.2 Without `QuickFindSolution`

We are now in a position to give a short analysis of Figure A.2. Our purpose is double: measuring the effects of removing `QuickFindSolution` from Algorithm 8.2, as well as evaluating both the results involving `FindSolution` and the possible over-estimation of interval solutions. Note that in this context, test data are only generated once each interval solution is found. This also explains why the columns



WITHOUT QuickFindSolution

1: Total vertices
2: Labelling level
3: epsilon
4: timeout for constraint solving (sec.)
5: Total path constraints generated
6: Total solved path constraints
7: Total path constraints with timeout
8: Total path constraints with no interval solution
9: Total path constraints with interval solutions but no float solution
10: Total calls to QuickFindSolution
11: Total failed QuickFindSolution(s)
12: Total winning QuickFindSolution(s)
13: Total calls to FindSolution
14: Total failed FindSolution(s)
15: Total winning FindSolution(s)
16: Total solution-tests
17: The time needed to cover all ICFG vertices (sec.)
18: The maximum time spent on a vertex (sec.)
19: Coverage (%)

		Parameters			Path constraints					QuickFindSolution			FindSolution			Tests	Exec. time		
Function	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
NthRootBisect	10	1	-16	15	5	3	0	2	0	0	0	0	3	0	3	3	0.4	0.2	100
sample	15	1	-16	15	2	2	0	0	0	0	0	0	2	0	2	2	0.3	0.2	100
tritype	24	1	-16	15	10	8	0	2	0	0	0	0	8	0	8	8	0.3	0.1	100
proc	27	1	-16	15	4	4	0	0	0	0	0	0	454	450	4	454	0.7	0.6	100
proc	27	5	-16	15	4	4	0	0	0	0	0	0	4	0	4	4	0.1	0.05	100
bsearch	10	1	-16	15	3	3	0	0	0	0	0	0	3	0	3	3	0.2	0.1	100
cMichel	4	1	-16	15	2	2	0	0	0	0	0	0	3	1	2	3	0.09	0.06	100
gaujac	40	1	-16	15	12	2	9	1	0	0	0	0	4576	4574	2	4576	141.9	135.8	100
gaujac	40	5	-16	15	12	2	9	1	0	0	0	0	4360	4358	2	17464	142	135.8	100
gaujac	40	10	-16	15	12	2	9	1	0	0	0	0	1815	1813	2	172311	142.1	135.9	100
ei	22	1	-32	15	22	6	0	16	0	0	0	0	6	0	6	6	2	0.9	100
eiDead	27	1	-32	15	39	6	0	17	16	0	0	0	246	240	6	246	3.7	1.9	92
eiDead	27	5	-32	15	39	6	0	17	16	0	0	0	246	240	6	918	3.8	1.9	92
gamdev	46	1	-16	15	6	5	0	1	0	0	0	0	5	0	5	5	0.5	0.3	100
bessi	27	1	-16	15	22	4	0	18	0	0	0	0	4	0	4	4	1.86	1.84	100
expint	35	1	-16	15	19	7	11	1	0	0	0	0	3450	3443	7	3450	165.7	150.2	97
expint	35	5	-16	15	19	7	11	1	0	0	0	0	3354	3347	7	9706	165.7	150.1	97

Figure A.2: Without QuickFindSolution

related to `QuickFindSolution` were all filled with zeros. It is important to note also that the experiments reported in this figure were done on the same environment (with the same machine, the same parameters, ...) as those reported in Figure A.1. Otherwise, our comparison below is senseless.

The effects of removing `QuickFindSolution` are clearly seen when we make a comparison between Figure A.2 and Figure A.1 in terms of time performance, indicated by columns 17 and 18 in Figure A.2 and by columns 17, 18 and 19 in Figure A.1. For all the programs under test, the execution time generally increased when without `QuickFindSolution`. A little increase in execution time was seen in most cases, except for program `expint`, where the total time to cover all ICFG vertices (column 17) was 165.7, compared with 16.6 (with `QuickFindSolution`). The coverage in the case with `expint` was even worse. Only a 97% coverage was reached, even when we increased the labeling level.

We now focus our attention to columns 13, 14 and 15 related to `FindSolution`. We find again the same situation as was already discussed in the previous section: a lot of "failed" calls to `FindSolution` for `gaujac`, `eiDead`, as well as `expint` (Note however that for `expint` in Figure A.1, all test data were found by `QuickFindSolution`.), even when we increased the labeling level. By going into detailed reading of these programs, we find the following common characteristic: They contain loops whose body consists of many calculations involving many variables with multiple occurrences. In that case, any path going through such loops several times, may lead to a possible explosion of the effects of multi-occurrences of the same variables, which in turn is the original cause of the over-estimation of interval solutions. Our conclusion is that if there are many over-estimated interval solutions, it is obvious to see results concerning `FindSolution` such as those shown in Figure A.2 and Figure A.1.

Appendix B

Benchmarks

B.1 NthRootBisect.c

```
#include <math.h>

double nThRootBisect(double a, int n, double e) {
  double l, h, c;

  l = 1; h = a;
  while ((h - l)*(h - l) >= e) {
    c = (l + h)/2;
    if (pow(c, n) - a == 0) return c;
    if ((pow(l, n) - a)*(pow(c, n) - a) < 0) h = c;
    else l = c;
  }
  return h;
}
```

B.2 Sample.c

```
int sample(int a[10], int b[10], int target) {
  int i, fa, fb;

  i = 0;
  fa = 0;
  fb = 0;
  while (i <= 9) {
    if (a[i] == target) fa = 1;
    ++i;
  }
  if (fa == 1) {
    i = 0;
    fb = 1;
    while (i <= 9) {
      if (b[i] != target) fb = 0;
      ++i;
    }
  }
  if (fb == 1) return 0;
  else return 1;
}
```

B.3 Tritype.c

```
int tritype(int i, int j, int k) {
  int trityp;

  if ((i == 0) || (j == 0) || (k == 0)) trityp = 4;
  else {
    trityp = 0;
    if (i == j) trityp = trityp+1;
    if (i == k) trityp = trityp+2;
    if (j == k) trityp = trityp+3;
    if (trityp == 0) {
      if ((i+j <= k)||(j+k <= i)||(i+k <= j)) trityp = 4;
      else trityp = 1;
    }
    else if (trityp > 3) trityp = 3;
    else if ((trityp == 1) && (i+j > k)) trityp = 2;
    else if ((trityp == 2) && (i+k > j)) trityp = 2;
    else if ((trityp == 3) && (j+k > i)) trityp = 2;
    else trityp = 4;
  }
  return trityp;
}
```

B.4 Proc.c

```
#include <stdio.h>

void b(double a[10]);
void c(double *x, double *y);
int f(int i);

void proc(double a[10], int c) {
  int i;
  i = 1;
  while (i <= c) {
    b(a);
    ++i;
  }
}

void b(double a[10]) {
  int i, j, fi, fj;

  printf("i j ? "); scanf("%d %d", &i, &j);
  fi = f(i);
  fj = f(j);
  if (fi < fj) c(&a[i], &a[j]);
  else c(&a[j], &a[i]);
}

void c(double *x, double *y) {
  double t;

  if (*x > *y) {
    t = *x;
    *x = *y;
    *y = t;
  }
}

int f(int i) {
  if (i >= 0 && i <= 9) return i;
  else exit(1);
}
```

B.5 BSearch.c

```
int bsearch(double a[10], double elem) {
  int low = 0;
  int high = 9;
  int mid;
  while (high >= low) {
    mid = (low + high)/2;
    if (elem == a[mid]) return 1;
    if (elem > a[mid]) low = mid + 1;
    else high = mid - 1;
  }
  return 0;
}
```

B.6 CMichel.c

```
int cMichel(double x) {
  if (16.0+x == 16.0 && x > 0) return 1;
  else return 0;
}
```

B.7 gaujac.c

```
#include <math.h>
#define EPS 3.0e-14
#define MAXIT 10
#define N 6

double gammln(double xx);

void gaujac(double x[N], double w[N], double alf, double bet) {
  int i,its,j;
  double alfbet,an,bn,r1,r2,r3;
  double a,b,c,p1,p2,p3,pp,temp,z,z1,gl1,gl2,gl3,gl4;

  for (i=1;i<=N;++i) {
    if (i == 1) {
      an=alf/N;
      bn=bet/N;
      r1=(1.0+alf)*(2.78/(4.0+N*N)+0.768*an/N);
      r2=1.0+1.48*an+0.96*bn+0.452*an*an+0.83*an*bn;
      z=1.0-r1/r2;
    } else if (i == 2) {
      r1=(4.1+alf)/((1.0+alf)*(1.0+0.156*alf));
      r2=1.0+0.06*(N-8.0)*(1.0+0.12*alf)/N;
      r3=1.0+0.012*bet*(1.0+0.25*fabs(alf))/N;
      z -= (1.0-z)*r1*r2*r3;
    } else if (i == 3) {
      r1=(1.67+0.28*alf)/(1.0+0.37*alf);
      r2=1.0+0.22*(N-8.0)/N;
      r3=1.0+8.0*bet/((6.28+bet)*N*N);
      z -= (x[0]-z)*r1*r2*r3;
    } else if (i == N-1) {
      r1=(1.0+0.235*bet)/(0.766+0.119*bet);
      r2=1.0/(1.0+0.639*(N-4.0)/(1.0+0.71*(N-4.0)));
      r3=1.0/(1.0+20.0*alf/((7.5+alf)*N*N));
      z += (z-x[N-4])*r1*r2*r3;
    } else if (i == N) {
      r1=(1.0+0.37*bet)/(1.67+0.28*bet);
      r2=1.0/(1.0+0.22*(N-8.0)/N);
      r3=1.0/(1.0+8.0*alf/((6.28+alf)*N*N));
      z += (z-x[N-3])*r1*r2*r3;
    } else {
      z=3.0*x[i-2]-3.0*x[i-3]+x[i-4];
    }

    alfbet=alf+bet;
    for (its=1;its<=MAXIT;++its) {
      temp=2.0+alfbet;
      p1=(alf-bet+temp*z)/2.0;
      p2=1.0;
      for (j=2;j<=N;++j) {
        p3=p2;
        p2=p1;
        temp=2*j+alfbet;
        a=2*j*(j+alfbet)*(temp-2.0);
        b=(temp-1.0)*(alf*alf-bet*bet+temp*(temp-2.0)*z);
        c=2.0*(j-1+alf)*(j-1+bet)*temp;
        p1=(b*p2-c*p3)/a;
      }
      pp=(N*(alf-bet-temp*z)*p1+2.0*(N+alf)*(N+bet)*p2)/(temp*(1.0-z*z));
      z1=z;
      z=z1-p1/pp;
      if (fabs(z-z1) <= EPS) break;
    }
    if (its > MAXIT) {
      printf("too many iterations in gaujac\n");
      exit(1);
    }
    x[i-1]=z;
    gl1 = gammln(alf+N);
    gl2 = gammln(bet+N);
    gl3 = gammln(N+1.0);
    gl4 = gammln(N+alfbet+1.0);
    w[i-1]=exp(gl1+gl2-gl3-gl4)*temp*pow(2.0,alfbet)/(pp*p2);
  }
}

double gammln(double xx) {
  double x,y,tmp,ser;
  static double cof[6]={76.18009172947146,-86.50532032941677,
                        24.01409824083091,-1.231739572450155,
                      0.1208650973866179e-2,-0.5395239384953e-5};
  int j;

  y=x=xx;
  tmp=x+5.5;
  tmp -= (x+0.5)*log(tmp);
  ser=1.000000000190015;
  for (j=0;j<=5;++j) ser += cof[j]/++y;
  return -tmp+log(2.5066282746310005*ser/x);
}
```

B.8 expint.c

```
#include <math.h>
#define MAXIT 10
#define EULER 0.5772156649
#define FPMIN 1.0e-30
#define EPS 1.0e-7

double expint(int n, double x) {
  int i,ii,nm1;
  double a,b,c,d,del,fact,h,psi,ans;

  nm1=n-1;
  if (n < 0 || x < 0.0 || (x==0.0 && (n==0 || n==1)))
    exit(1);
  else {
    if (n == 0) ans=exp(-x)/x;
    else {
      if (x == 0.0) ans=1.0/nm1;
      else {
        if (x > 1.0) {
          b=x+n;
          c=1.0/FPMIN;
          d=1.0/b;
          h=d;
          for (i=1;i<=MAXIT;++i) {
            a = -i*(nm1+i);
            b += 2.0;
            d=1.0/(a*d+b);
            c=b+a/c;
            del=c*d;
            h *= del;
            if (fabs(del-1.0) < EPS) {
              ans=h*exp(-x);
              return ans;
            }
          }
          exit(1);
        } else {
          if (nm1 != 0) ans = 1.0/nm1; else ans = -log(x)-EULER;
          fact=1.0;
          for (i=1;i<=MAXIT;++i) {
            fact *= -x/i;
            if (i != nm1) del = -fact/(i-nm1);
            else {
              psi = -EULER;
              for (ii=1;ii<=nm1;++ii) psi += 1.0/ii;
              del=fact*(-log(x)+psi);
            }
            ans += del;
            if (fabs(del) < fabs(ans)*EPS) return ans;
          }
          exit(1);
        }
      }
    }
  }
  return ans;
}
```

B.9 gamdev.c

```
#include <math.h>

#define IA 16807
#define IM 2147483647
#define AM 4.656612875245797e-10
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV 67108864
#define EPS 1.2e-7
#define RNMX 0.99999988

long iy=0;
long iv[NTAB];
double ran1(long *idum);

double gamdev(int ia, long *idum) {
  int j;
  double am,e,s,v1,v2,x,y;
  double r1,r2,r3;

  if (ia < 1) exit(1);
  if (ia < 6) {
    x=1.0;
    for (j=1;j<=ia;++j) {
      r1 = ran1(idum);
      x *= r1;
    }
    x = -log(x);
  } else {
    do {
      do {
        do {
          v1=ran1(idum);
          r2 = ran1(idum);
          v2=2.0*r2-1.0;
        } while (v1*v1+v2*v2 > 1.0);
        y=v2/v1;
        am=ia-1;
        s=sqrt(2.0*am+1.0);
        x=s*y+am;
      } while (x <= 0.0);
      e=(1.0+y*y)*exp(am*log(x/am)-s*y);
      r3 = ran1(idum);
    } while (r3 > e);
  }
  return x;
}

double ran1(long *idum) {
  int j;
  long k;
  double temp;

  if (*idum <= 0 || iy == 0) {
    if (-(*idum) < 1) *idum=1;
    else *idum = -(*idum);
    for (j=NTAB+7;j>=0;--j) {
      k=(*idum)/IQ;
      *idum=IA*(*idum-k*IQ)-IR*k;
      if (*idum < 0) *idum += IM;
      if (j < NTAB) iv[j] = *idum;
    }
    iy=iv[0];
  }
  k=(*idum)/IQ;
  *idum=IA*(*idum-k*IQ)-IR*k;
  if (*idum < 0) *idum += IM;
  j=iy/NDIV;
  iy=iv[j];
  iv[j] = *idum;
  temp=AM*iy;
  if (temp > RNMX) return RNMX;
  else return temp;
}
```

B.10 bessi.c

```
#include <math.h>
#define ACC 40.0
#define BIGNO 1.0e10
#define BIGNI 1.0e-10

double bessi0(double x);

double bessi(int n, double x) {
  int j;
  double bi,bim,bip,tox,ans,bsi0x;

  if (n < 2) exit(1);
  if (x == 0.0)
    return 0.0;
  else {
    tox=2.0/fabs(x);
    bip=ans=0.0;
    bi=1.0;
    for (j=2*(n+(int)sqrt(ACC*n));j>0;--j) {
      bim=bip+j*tox*bi;
      bip=bi;
      bi=bim;
      if (fabs(bi) > BIGNO) {
        ans *= BIGNI;
        bi *= BIGNI;
        bip *= BIGNI;
      }
      if (j == n) ans=bip;
    }
    bsi0x = bessi0(x);
    ans *= bsi0x/bi;
    if (x < 0.0 && n != ((int)n/2)*2) return -ans;
    else return ans;
  }
}

double bessi0(double x) {
  double ax,ans,y;

  ax=fabs(x);
  if (ax < 3.75) {
    y=x/3.75;
    y*=y;
    ans=1.0+y*(3.5156229+y*(3.0899424+y*(1.2067492
        +y*(0.2659732+y*(0.360768e-1+y*0.45813e-2)))));
  } else {
    y=3.75/ax;
    ans=(exp(ax)/sqrt(ax))*(0.39894228+y*(0.1328592e-1
        +y*(0.225319e-2+y*(-0.157565e-2
        +y*(0.916281e-2+y*(-0.2057706e-1+y*(0.2635537e-1+y*(-0.1647633e-1
        +y*0.392377e-2))))))));
  }
  return ans;
}
```

B.11 ei.c

```
#include <math.h>
#include <stdio.h>

#define EULER 0.57721566
#define MAXIT 17
#define FPMIN 1.0e-30
#define EPS 6.0e-8

double ei(double x) {
  int k;
  double fact,prev,sum,term;

  if (x <= 0.0) exit(1);
  if (x < FPMIN) return log(x)+EULER;
  if (x <= -log(EPS)) {
    sum=0.0;
    fact=1.0;
    for (k=1;k<=MAXIT;++k) {
      fact *= x/k;
      term=fact/k;
      sum += term;
      if (term < EPS*sum) break;
    }
    if (k > MAXIT) exit(1);
    return sum+log(x)+EULER;
  } else {
    sum=0.0;
    term=1.0;
    for (k=1;k<=MAXIT;++k) {
      prev=term;
      term *= k/x;
      if (term < EPS) break;
      if (term < prev) sum += term;
      else {
        sum -= prev;
        break;
      }
    }
    return exp(x)*(1.0+sum)/x;
  }
}
```

B.12 ei-dead.c

```
#include <math.h>
#include <stdio.h>
#define EULER 0.57721566
#define MAXIT 17
#define FPMIN 1.0e-30
#define EPS 6.0e-8

double eiDead(double x) {
  int k;
  double fact,prev,sum,term;

  if (x <= 0.0) exit(1);
  if (x < FPMIN) return log(x)+EULER;
  if (x <= -log(EPS)) {
    sum=0.0;
    fact=1.0;
    for (k=1;k<=MAXIT;++k) {
      fact *= x/k;
      if (k > MAXIT) return 0;
      term=fact/k;
      sum += term;
      if (term < EPS*sum) break;
    }
    if (k > MAXIT) exit(1);
    if (term >= EPS*sum) return 0;
    return sum+log(x)+EULER;
  } else {
    sum=0.0;
    term=1.0;
    for (k=1;k<=MAXIT;++k) {
      prev=term;
      term *= k/x;
      if (term < EPS) break;
      if (term < prev) sum += term;
      else {
        sum -= prev;
        break;
      }
    }
    return exp(x)*(1.0+sum)/x;
  }
}
```

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[2] F. Benhamou, W. Older, and A. Vellino. Constraint logic programming on boolean, integer and real intervals. *Journal of Symbolic Computation*, 1995.

[3] F. Benhamou and Tourvaïne. Prolog iv: language and algorithmes. In *IVème Journées Francophones de Programmation en Logique*, pages 51–65, 1995.

[4] R. V. Binder. Object-oriented testing: Myth and reality, 1995. Available at http://www.rbsc.com/pages/onlineix.html.

[5] R. V. Binder. The free approach to object-oriented testing: An overview, 1996. Available at http://www.rbsc.com/pages/onlineix.html.

[6] L. Bougé. *Modèlisation de la notion de Test de Programmes: Application à la production de Jeux de Tests.* PhD thesis, Université Paris VI, November 1982.

[7] R. S. Boyer, B. Elspas, and K. N. Levitt. Select–a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, 1975.

[8] M. M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, November 1994.

[9] U. A. Buy, A. Orso, and M. Pezzè. Automated testing of classes. In *ISSTA'00*, pages 39–48, 2000.

[10] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[11] A. Coen-Porisini and F. D. Paoli. Array representation in symbolic execution. *Computer Languages*, 18(3):197–216, 1993.

[12] S. Cornett. Code coverage analysis, 2002. Available at http://www.bullseye.com/coverage.html.

[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In *ESOP'05*, pages 21–30, 2005.

[14] D. Coward and D. Ince. *The Symbolic Execution of Software, The Sym-Bol System.* Chapman & Hall, 1995.

[15] F. Delobel. *Résolution de systèmes de contraintes réelles non linéaires.* PhD thesis, Université de Nice-Sophia Antipolis(UNSA), Jan. 2000.

[16] R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, 2(2):109–127, April 1993.

[17] E. W. Dijkstra. *A discipline of programming.* Prentice Hall, 1976.

[18] J. Doppke and A. Klauser. Pl concepts: Parameter passing. Available at http://www.cs.colorado.edu/~humphrie/pl/param.html.

[19] J. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[20] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering (CCSSE'99)*, pages 21–28, 1999.

[21] R. Ferguson and B. Korel. The chaning approach for software test data generation. *ACM Transactions on Software Engineering Methodology*, 5(1):63–86, 1996.

[22] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its uses in optimization. *ACMTransactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[23] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[24] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2):156–173, 1975.

[25] A. Gotlieb. *Génération de cas de test structurel avec la programmation logique par contraintes.* PhD thesis, Université de Nice-Sophia Antipolis, Jan. 2000.

[26] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In *Computational Logic*, pages 399–413, 2000.

[27] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *ASE'01*, pages 5–12, 2001.

[28] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering (TSE)*, 9(6):686–709, 1983.

[29] N. Gupta and R. Gupta. Data flow testing. In *The Compiler Design Handbook: Optimizations and Machine Code Generation*, pages 247–268, 2002.

[30] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering(FSE-6)*, pages 231–244, Orlando, Florida, Nov. 1998.

[31] N. Gupta, A. P. Mathur, and M. L. Soffa. UNA based iterative test data generation and its evaluation. In *14th IEEE International Conference on Automated Software Engineering(ASE'99)*, pages 224–232, Cocoa Beach, Florida, Oct. 1999.

[32] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *15th IEEE International Conference on Automated Software Engineering(ASE00)*, September 2000.

[33] M. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, Mar 1997.

[34] M. J. Harrold, A. J. Offutt, and K. Tewary. An approach to fault modeling and fault seeding using the program dependence graph. *Journal of Systems and Software*, 36(3):273–295, 1997.

[35] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *SIGSOFT FSE'94*, pages 154–163, 1994.

[36] P. V. Hentenryck, L. Michel, and Y. Deville. *Numerica. A modeling language for global optimization.* The MIT Press, Cambridge, Massachusetts, London, England, 1997.

[37] T. Hickey. An interval arithmetic library, 2000. Available at http://interval.sourceforge.net/interval/index.html.

[38] T. J. Hickey, Q. Ju, and M. H. van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

[39] S. Hollasch. Ieee standard 754 floating point numbers. Available at http://stevehollasch.com/cgindex/coding/ieeefloat.html.

[40] R. Hower. Software qa and testing resource center. Available at http://www.softwareqatest.com/.

[41] IEEE Computer Society. *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754-1985, 1985.

[42] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20(19):503–581, 1994.

[43] P. C. Jorgensen and C. Erickson. Object-oriented integration testing. *Commun. ACM*, 37(9):30–38, 1994.

[44] J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[45] O. Koné and R. Castanet. Test generation for interworking systems. *Computer Communications*, 23:642–652, 2000.

[46] B. Korel. Automated software test data generation. *IEEE Trans. Software Eng.*, 16(8):870–879, 1990.

[47] B. Korel. Automated test data generation for programs with procedures. In S. J. Ziel, editor, *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 209–215, 1996.

[48] B. Marre. Toward automatic test data set selection using algebraic specifications and logic programming. In *Proceedings of the ICLP'91*, pages 202–219, 1991.

[49] D. Melski and T. W. Reps. Interprocedural path profiling. In *Computational Complexity*, pages 47–62, 1999.

[50] C. Michel, M. Rueher, and Y. Lebbah. Solving constraint over floating-point numbers. In *Seventh International Conference on Principles and Practice of Constraint.* Springer Verlag, LNCS, 2001.

[51] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, pages 3–17, 2004.

[52] R. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[53] A. J. Offutt, Z. Jin, and J. Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. *Software Practice and Experience*, 29(2):167–193, Jan. 1997.

[54] R. P. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[55] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C. The Art of Scientific Computing. Second Edition.* Cambridge University Press, 1992.

[56] R. S. Pressman. *Software engineering (3rd ed.): a practitioner's approach.* McGraw-Hill, Inc., 1994.

[57] C. V. Ramamoorthy, S.-B. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Trans. Software Eng.*, 2(4):293–300, 1976.

[58] D. J. Richardson and M. C. Thompson. An analysis of test data selection criteria using the RELAY model of fault detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, June 1993.

[59] RTCA SC-167 / EUROCAE WG-12. *Software considerations in airborne systems and equipment certification. Document No. RTCA/DO-178B.* RTCA, Inc., 1992.

[60] S. Sinha and M. J. Harrold. Criteria for testing exception-handling constructs in java programs. In *ICSM'99*, pages 265–, 1999.

[61] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *Software Engineering and Methodology*, 10(2):209–254, 2001.

[62] N. T. Sy and Y. Deville. COTTAGE: Test data generation based on consistency techniques. Technical Report, UCL/FSA/INGI, November 2004. Available at http://www.info.ucl.ac.be/people/YDE/yde.html.

[63] N. T. Sy and Y. Deville. Automatic test data generation for programs with integer and float variables. In *16th IEEE International Conference on Automated Software Engineering(ASE01)*, November 2001.

[64] N. T. Sy and Y. Deville. Consistency techniques for interprocedural test data generation. In *Proceedings of the ESEC/FSE'03*, pages 108–117, 2003.

[65] N. Tracey, J. Clark, K. Mander, and J. A. McDermid. An automated framework for structural test-data generation. In *ASE'98*, pages 285–288, 1998.

[66] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[67] H. Waeselynck. *Vérification de Logiciels Critiques par le Test Statistique*. PhD thesis, Institut National Polytechnique de Toulouse, Jan. 1993.

[68] E. J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM J. Comput.*, 8(4):587–598, 1979.

[69] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-path tests for c functions. In *ASE'04*, pages 290–293, 2004.