Integrated Module Testing and Module Verification

von **Tatiana Mangels**

Dissertation

zur Erlangung des Grades einer Doktorin der
Ingenieurwissenschaften
– Dr.-Ing. –

Vorgelegt im Fachbereich 3 (Mathematik & Informatik)
der Universität Bremen
im Oktober 2013

Datum des Promotionskolloquiums: 10.12.2013

Gutachter: Prof. Dr. Jan Peleska (Universität Bremen)
Prof. Dr. Rolf Drechsler (Universität Bremen)

To my family

Zusammenfassung

In der vorliegenden Dissertation ist ein integriertes Vorgehen für die formale Verifikation durch Model Checking und Modultest beschrieben. Der Schwerpunkt liegt dabei auf der Verifikation von C Funktionen. Spezifikationsorientiertes Testen und funktionale Verifikation erfordern eine formalisierte Spezifikation der Module. Für diesen Zweck wurde eine Sprache zur Annotation als Erweiterung der Syntax von Vor- beziehungsweise Nach-Bedingungen erarbeitet und wird in der vorliegenden Arbeit vorgestellt. Diese Sprache zur Annotation erlaubt die Definition von logischen Bedingungen welche den Zustand eines Programms vor seiner Ausführung mit dem nach der Ausführung verbinden. Weiterhin wurde für die Nachvollziehbarkeit der überdeckten Anforderungen eine Syntax zur Spezifizierung von Testfällen erarbeitet. Die spezifizierten Korrektheitsbedingungen können außerdem durch die Einführung von entsprechenden Hilfsvariablen verfeinert werden. Über die Spezifikation der zu testenden Modulen hinaus wird die Sprache zur Annotation ebenfalls zur Modellierung des Verhaltens von externen Funktionen verwendet, welche nicht unmittelbar Teil der zu testenden Funktion beziehungsweise Prozedur sind, jedoch von dieser aufgerufen werden.

Durch die Spezifikation von Vor- beziehungsweise Nach-Bedingungen sowie von Testfällen reduziert sich die Generierung von Testdaten sowohl für strukturelles als auch für funktionales Testen jeweils auf ein Erreichbarkeitsproblem innerhalb des Kontrollflussgraphen des Moduls. Dieses wiederum ist aus dem Bounded Model Checking bekannt. Zur Lösung des Erreichbarkeitsproblems wird in der vorliegenden Arbeit symbolische Ausführung verwendet. Der Vorteil der symbolischen Ausführung ist ihre Genauigkeit und ihre Fähigkeit mehrere Programmeingaben gleichzeitig zu berücksichtigen. Dennoch hat die symbolische Ausführung auch Einschränkungen wie zum Beispiel die Verarbeitung von Aliasing oder der von Aufrufen von externen Funktionen. Diese Einschränkungen werden analysiert und es werden neue Algorithmen zur Behandlung der zentralen indentifizierten Probleme erarbeitet. Weiterhin werden Strategien für die Auswahl von Testfälle und für das Expandieren der unterliegenden Datenstruktur entwickelt und vorgestellt. Diese Strategien minimieren die Anzahl der untersuchten Zustände beim Erreichen der maximalen Codeabdeckung.

Die entwickelten Algorithmen und Strategien wurden im Testdatengenerator CTGEN implementiert. CTGEN generiert Testdaten sowohl für eine C1 Codeabdeckung als auch für eine funktionale Abdeckung. Weiterhin unterstützt der implementierte Generator die automatische Erzeugung von Stubs. Dabei erfüllen die Daten welche ein Stub während der Ausführung eines Tests zurückgibt die Spezifikation der entsprechenden externen Funktion. CTGEN wird außerdem mit anderen konkurrierenden Testdatengeneratoren verglichen. Er liefert dabei konkurrenzfähige Resultate.

Abstract

In this dissertation an integrated approach to formal module verification by model checking and module testing is described. The main focus lays on the verification of C functions. Specification-based testing and functional verification require a formalized module specification. For this purpose an annotation language as an extension of a pre-/post-condition syntax is developed and discussed. This annotation language allows the definition of logical conditions relating the program's pre-state to its post-state after executing the module. For requirements tracking a test case specification is developed. The correctness conditions can be refined by the introduction of auxiliary variables. Besides the specification of the module under test, the presented annotation language allows to model the behavior of external functions called by the module under test.

By the specification of pre- and post-conditions as well as test cases, test data generation for both structural and functional testing is reduced to a reachability problem (as known from bounded model checking) within the module's control flow graph. These reachability problems are investigated using symbolic execution. The strength of symbolic execution is in its precision and its ability to reason about multiple program inputs simultaneously, but it also has limitations like aliasing or external function calls. These in turn are analyzed and new algorithms are developed which overtake most of the detected limitations. The expansion and selection strategies for test case selection are developed and described. They allow to minimize the size of investigated states and the number of generated test cases, while achieving maximal branch coverage.

The developed algorithms and strategies are implemented in the test generator CTGEN, which generates test data for C1 structural coverage and for functional coverage. It also supports automated stub generation where the data returned by a stub during test execution depends on the specification provided by the user. CTGEN is evaluated and compared with competing tools and produces competitive results.

Acknowledgments

Here I am, writing the last words of my thesis. This was a long way and sometimes I lost any hope that I would be able to some day finish it. Thankfully, I had people around me who believed in me and supported me during my work. First of all, I would like to thank my supervisor, Professor Dr. Jan Peleska for guiding me and giving me helpful advice when I was stuck, not to mention all of his support during the last phase of my dissertation. Furthermore, I would like to thank Siemens AG for supporting my work through a research grant. Also, I would like to thank my husband for his belief in me, which always encouraged me to go on. I thank my colleague Florian Lapschies for inspiring discussions and sometimes just for listening to me and, of course, for his solver. I thank all my colleagues at the University of Bremen for the friendly atmosphere and fascinating conversations. I thank my father for solving my writer's block. And, last but not least, I thank my mother and my daughter for simply loving me.

Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

CACC	Correlated Active Clause Coverage
CDCL	Conflict Driven Clause Learning
CFG	Control Flow Graph
CTGEN	Test Generator for C
IEEE	Institute of Electrical and Electronics Engineers
IMR	Intermediate Model Representation
ISTQB	International Software Testing Qualifications Board
MCDC	Modified condition/decision coverage
NIST	National Institute of Standards and Technology
RTTL	Real-Time Test Language
STCT	Symbolic Test Case Tree
UUT	Unit Under Test

1 Introduction

1.1 Objectives

In this dissertation an integrated approach to formal module verification by model checking and module testing is presented. Under *verification* we understand all quality assurance activities that check whether an object fulfills the specified requirements [55]. In particular, reviews, walk-throughs, inspections, formal verification, static analysis and testing are verification activities. Within the software life cycle module verification has its established place, and static analysis, testing and formal verification are approved techniques for this purpose. As was pointed out in [80] it is recommended to use these techniques in an integrated manner. Thus, they can benefit from one another:

- Test cases can be used as counterexamples for violated assertions, thus supporting the static analysis and formal verification processes.

- Compared to functional testing, static analysis is more successful when investigating non-functional properties, such as worst case execution time or the absence of run-time errors.

- If algorithms are too complex to be tested or analyzed in an exhaustive way, formal verification is the technique of choice.

The focus of this thesis is on the verification of C functions and procedures (hereafter referred to as *module* or the *unit under test (UUT)*). *Unit testing* is a well-known approach, widely used in practice, by which a single module is tested separately with respect to its functional correctness. Within the scope of this thesis tests investigating non-functional properties are not considered since these are often more successfully investigated by means of formal verification, static analysis or abstract interpretation.

Specification-based testing and functional verification require a formalized module specification. For this purpose we define an annotation language including a pre- and postcondition syntax. This allows us to define logical conditions relating the program's prestate to its poststate. More complex correctness conditions, such as for example logical statements over the number of function calls performed by the UUT, may also be specified. In this case, auxiliary variables are introduced. By the specification of pre- and postconditions the test case generation for both *structural* and *functional* testing reduces itself to a *reachability problem* within the module's Control Flow Graph (CFG).

The ideas introduced within this thesis are incorporated into CTGEN, an automatic test generation tool, based on symbolic execution. Since covering every branch in a program is in general an undecidable problem, the objective of CTGEN is to generate a test that produces as high a coverage for the module under test as possible. For each UUT CTGEN performs symbolic analysis and generates a test in RT-Tester syntax [44], which can be compiled and executed.

1.2 Motivation

Ten years ago, the U.S. Department of Commerce's National Institute of Standards and Technology (NIST) estimated that due to low software quality the U.S. economy loses $59,5 billion annually [99]. Although the study was conducted in 2002, the quality of software is in general still a significant issue. The authors of "The economics of Software Quality" [61] state in their book among others the following reasons:

1. Software of low quality is expensive, and the costs are proportional to the size of the project. Table 1.1 illustrates typical costs for development of low-, average- and high-quality software. *"High quality"* here refers to software where the development process *"includes effective defect prevention, effective pretest defect removal such as inspections and static analysis, and much more effective testing than for the other columns."* The authors declare, that testing alone was never enough to achieve high-quality software, but it is still an essential part of the quality assurance process.

2. Software errors affect everybody. Software is among the most used products in history, we use it every day and almost everywhere. A software failure can lead to consequences from simple inconvenience up to life hazard.

The issue of software quality is especially important in the development of safety-critical systems. To address this, quality standards [4, 40, 31] were established. But, as mentioned in [80], these standards do not see 100% correct software as a principal goal since the code correctness does not automatically guarantee system safety. Standards request (a) identification of the criticality level of software components, i.e. its contribution to system safety or, on the contrary, risks and hazards that the possible component failure may cause, (b) the software shall be developed and verified with state-of-the-art techniques and with effort symmetrical to the criticality level of the component. Depending on the criticality level, standards define precisely which techniques should be applied and which effort is seen as adequate. So, tests should [80]:

1. Execute each functional requirement at least once.

2. Produce complete code coverage according to the coverage criteria: statement, branch or modified condition/decision coverage. The applicable coverage criteria is defined in standards corresponding to the software criticality level.

Function Points	Low Quality	Average Quality	High Quality
10	$6,875	$6,250	$5,938
100	$88,561	$78,721	$74,785
1,000	$1,039,889	$920,256	$846,636
10,000	$23,925,127	$23,804,458	$18,724,012
100,000	$507,767,782	$433,989,557	$381,910,810

Table 1.1: Software Costs by Size and Quality Level [61].

3. Show the appropriate integration of the software on the target hardware.

However, the manual elaboration of test data and the development of test procedures exercising this data on the UUT is time consuming and expensive. The objective for the development of CTGEN is to support the verification process and to help providing the required results faster and with less effort compared to a manual approach. Under the assumption that requirements were assigned to corresponding modules by means of an annotation language, CTGEN provides tests with related functional requirements coverage. In case of requirement violation a counter example is generated, which, in turn, supports finding the defect in the affected module. Furthermore, CTGEN aims at producing complete branch coverage.

1.3 Software Testing

The study of NIST came to the following conclusion: *“The path to higher software quality is significantly improved software testing”* [78].

This section outlines our understanding of the generic term *software testing*, which has varying definitions in literature. According to the Institute of Electrical and Electronics Engineers (IEEE) Guide to the Software Engineering Body of Knowledge [19] testing is *“an activity performed for evaluating product quality, and for improving it, by identifying defects and problems. [...] Software testing consists of the dynamic verification of the behavior of a program [...] against the expected behavior”*. Myers [76] defines testing as *“the process of executing a program with the intent of finding errors”*. According to Binder’s view [16], testing is *“the design and implementation of a special kind of software system: one that exercises another software system with the intent of finding bugs“*. International Software Testing Qualifications Board (ISTQB) Standard Glossary of Terms [3] sees testing as *“The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.“* These are only a few examples of definitions, the literature provides many more.

The aforementioned definitions have in common that they focus on the aim of testing, which is to find “errors”, “bugs” or “defects”, but they differ in other aspects. While IEEE, Myers and Binder see testing as “executing” or “exercising” a program or software (also called *dynamic testing*), ISTQB has a broader understanding of testing which includes *dynamic* as well as *static* testing activities. Static testing is defined [3] as *“testing of a component or system [...] without execution of that software”*. So testing is not only exercising the program under test and observing the results but also activities like inspections, walk-throughs, reviews [76] or static analysis [7].

Furthermore, the purpose of testing is extended from an intent to find “errors”or “bugs” [76, 16] to evaluation and improvement of quality [19] or the demonstration that the system under test is fit for its purpose and to ensure that it satisfies the specified requirements [3].

In this thesis we see testing as it is defined by ISTQB and focus on dynamic testing, namely on the definition of test data and test procedures which should provide the basis for a conclusion whether the module satisfies the defined requirements.

1.3.1 Test Techniques

Over the years numerous different test design techniques were suggested [76, 7, 55, 84]. Based on models of the software system, typically in the form of requirements or design documents, they are divided into two areas: black box and white box testing.

Black Box Testing Black box testing (also referred to as functional testing or specification-based testing) is based only on the specification of the software under test and does not consider its design or implementation structure. The point of view of the test designer in black box testing is *outside* of the test object. Thus, the software under test is seen as a black box.

One important question in the design of tests is *"Which test cases will most likely detect the most errors?"*[76]. Black box testing offers a number of techniques which help to approach this problem systematically and make the test design more effective. In the following we list the most common techniques and give a short description of each of them.

- **Random Testing** [76, 84] is the simplest and least effective method. Following this approach, the software is tested by selecting random inputs from the possible range of values and comparing the output with the expected result, which is derived from the software specification. It is unlikely that a randomly selected input set will discover the most errors. To illustrate this we consider the following example:

  ```
  if(x == 2){
    ERROR;
  }
  ```

  where `x` is an integer input variable not set before the `if`-statement. The probability, that the line with an `ERROR` will be executed by random testing is in the order of $1/n$, where n is the range of the integer data type. However, despite its limitations, random testing is largely used in test generation since it can be automated easily. Another advantage is that this technique can be used in combination with other approaches when the software under test is so complex, that it is infeasible to apply other methodologies exhaustively.

- **Equivalence Partitioning** [55] is a technique whose basic idea is to partition input or output space into equivalence classes. The equivalence classes are derived from the software specification and it is assumed that all members of the same class behave in the same way. In this way, testing only one representative of the equivalence class leads to the same result as testing all of its members.

  The equivalence partitioning technique makes it possible to derive the completeness of the test suite by measuring the coverage of the equivalence partitions. Furthermore, by testing only one member of the class this technique avoids redundant tests. However, the probability of failure detection depends on the quality of the partitioning as well as on which representatives of the equivalence class were chosen for the test cases.

- **Boundary Value Analysis** [55, 7] is strongly related to the equivalence partitioning technique. A *boundary value* is a value on the boundaries of the equivalence class. Such a value demands additional attention because errors often appear at the boundaries of the equivalence classes [76].

- **Cause-Effect Graphing** [76]. The disadvantage of both the boundary value analysis and the equivalence partitioning is that they see different input data as independent and do not consider their combinations. Still, it is possible that one and the same input in combination with a second input will uncover a fault in the software under test, while the combination with a third input will not be successful in uncovering a fault. To test all possible combinations of inputs often is an infeasible task, since the number of test cases derived from the Cartesian product of the equivalence partitions is usually quite large. The cause-effect graphing is a technique that uses dependencies and aids to select test cases in a systematic manner. First, the specification is divided into smaller pieces, from which inputs (causes) and outputs (effects) are derived. The causes and effects are linked using the Boolean graph, which is transformed into a decision table. Thereby each column of this table corresponds to a test case.

White Box Testing White box testing (also referred to as structural or glass-box testing) is yet another approach to design test cases. It is based on the structure of the software implementation. The point of view of the test designer in white box testing is *inside* of the test object. The general idea of white box testing techniques is to execute each part of the source code at least once. The logic of the program is analyzed and test cases are designed, executed and compared against the expected results. It is important that the source code is never used as a basis for the determination of expected results. These must be derived from the specification.

Depending on the focus of examination, the following basic white box techniques are defined [7, 80, 107]:

- **Statement coverage** (C0) requires that each statement in the program is executed at least once. This is the weakest criterion, since in `if`-statements without `else` clauses the input, which evaluates the `if` condition to *false* is irrelevant and will be ignored. Therefore, the possibly missing `else` branch will not be detected by this technique.

- **Branch coverage** (C1) requires, that additionally to statement coverage each decision in the program is evaluated at least once to *true* and at least once to *false*. So, contrary to statement coverage missing `else` clauses are considered.

- **Modified condition/decision coverage** (MCDC) requires, that additionally to branch coverage every condition in every decision has taken all possible outputs at least once and that it was shown that each condition in each decision independently affected the outcome of the decision. To show that a condition independently affects an outcome, all other conditions in the decision must be fixed while only the condition under consideration is manipulated. MCDC coverage is a stronger criterion than C1 coverage. It is able to uncover faults which are masked by other conditions in the decision. This coverage criterion is required, for example, when testing avionic software of criticality level A.

- **Path coverage** (C2) requires, that each path in the program under test is executed at least once. This is the strongest criterion in white box testing, but complete path testing is not feasible for programs with loops or for programs with a large branching factor.

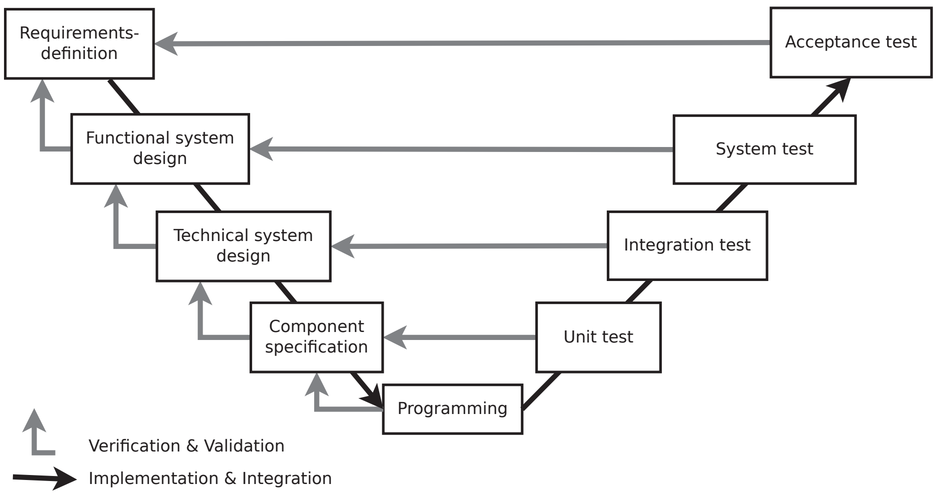


Figure 1.1: General V-model [7].

Black box and white box testing techniques uncover different types of faults. Test cases designed with black box testing techniques can only demonstrate that each requirement in the specification was implemented, whereas test cases designed with white box testing techniques can demonstrate that each implemented piece of code corresponds to a specific requirement. As a consequence, Myers [76] suggests to use elements of both design techniques and use white box testing techniques to supplement black box based test case design.

All presented techniques give an instrument to argue about the completeness of the performed testing in addition to aiding in the design of test cases. In this thesis we use white box testing techniques as a criterion to reason about completeness of the generated test suite. The developed test generator supports statement (C0) and branch (C1) coverage, whereas path coverage (C2) and MCDC coverage are not supported. The discussion about possible solutions for the integration of MCDC and path coverage into the test generator can be found in Section 7.4. Equivalence partitioning and boundary value analysis are out of the scope of this thesis. We discuss how the test generator can be expanded to support these techniques in Section 7.4.

1.3.2 Test Levels

The traditional view of the software life cycle suggests that software testing is performed at different levels along the development and maintenance processes. In the literature many test levels are introduced, but the most established ones are *unit (component)*, *integration*, *system* and *acceptance testing* [55, 7, 74, 59]. In the general *V-model* shown in Figure 1.1 each of these test levels is associated with a development process so that each development process has a corresponding test level.

- **Unit (component) test** is performed at the lowest level of the software development process. It verifies the functionality of software pieces which are separately testable in isolation. Such pieces can be functions, classes, subroutines and so on. Typically, unit testing uses both functional and structural techniques [19].

- **Integration test** can be performed as soon as two or more components are integrated into a system or a subsystem. The purpose of an integration test is not to find errors but to verify if the software components interact accordingly to the specification. Like unit testing, an integration test typically uses both functional and structural techniques.
- **System test** verifies whether the system as a whole meets the specified requirements. A system test considers not only functional, but also non-functional requirements, such as security, speed, accuracy and reliability. The system test should be performed in an environment as similar as possible to the intended one to evaluate external interfaces to other applications, hardware devices or the operating environment [7, 19]. During a system test functional techniques are typically used.
- **Acceptance test** is performed similar to a system test at the highest level of integration and executed in the intended environment. Nevertheless, on this level the goal is not to find defects in the product. An acceptance test evaluates if the system complies with the customer's requirements.

In this thesis we focus on structural testing at the unit test level. We do not consider functional testing, since by introducing pre- and postconditions as well as test cases, we reduce the problem of obtaining a functional test coverage to reaching structural test coverage (see Chapter 3).

1.4 Contributions

In this section we outline the contributions made by this thesis:

- Selection and expansion strategies minimizing the size of the structure that underlies the test case selection process (symbolic test case tree) and the number of test cases needed for achieving the desired coverage (Chapter 4).
- Handling of external function calls, which is one of the most important challenges for test data generation tools [89]. In this thesis a method for the automated generation of a *mock object* that replaces the external function by a test stub with the same signature is described (Section 5.12.2). This method also calculates values for the stub's return data and output parameters as well as for global variables which can be modified by the stubbed function in order to fulfill a path condition. Furthermore, using this technique, exceptional behavior of external functions can be simulated.
- Another challenge for test data generation tools is the handling of symbolic pointers and offsets [89]. To approach this challenge, a memory model was designed within our research group [80]. The corresponding algorithms for handling pointer and aliasing problems (in particular pointer arithmetics) were developed in the context of this thesis and are described in Section 5.7.
- An annotation language for supporting specification-based testing and functional verification was developed. As stated in Section 1.2, the standards demand that each functional requirement should be executed at least once. However, to our best knowledge none of the test data generating tools supports requirement tracing. The designed annotation language allows CTGEN to achieve this (Chapter 3).

The aforementioned techniques are incorporated into the design and development of CTGEN, a unit test generator for C code [72]. An overview of the architecture of CTGEN and its functionality is given in Chapter 2. CTGEN is able to produce test data for functional coverage derived from the specified pre- and postconditions as well as from test cases and C1 structural coverage. The generator also provides automated stub generation where the data returned by the stub during the execution of the test may be specified by means of the annotation language. CTGEN can cope with the typical aliasing problems present in low-level C, including pointer operations, structures and unions. Furthermore, CTGEN is able to generate complete test procedures which can be compiled and executed against the module under test. CTGEN was used in industrial scale test campaigns for embedded systems code in the automotive domain and demonstrated competitive results. Particularly when handling functions of high complexity, the results of CTGEN were better than, for example, those of KLEE [22] (Chapter 6).

1.5 Related Work

The content of this section was originally published in [72].

The idea of using symbolic execution for test data generation is not new, as it is an active area of research since the 70's [65, 28]. In the past a number of test data generation tools [22, 23, 18, 11, 69, 8, 100, 93, 47, 49, 86] were introduced. Nevertheless, to the best of our knowledge, only Pex (with Moles) supports automatic stub generation as provided by CTGEN. Furthermore, CTGEN seems to be the only tool supporting traceability between test cases and requirements. From the experimental results available from other tool evaluations we conclude that CTGEN outperforms most of them with respect to the UUT size that still can be handled for C1 coverage generation.

DART [47] is one of the first concolic testing tools to generate test data for C programs. It falls back to concrete values by external function calls, and does not support symbolic pointers. CUTE [93] is also a concolic test data generator for C, and, like DART, falls back to concrete values by external function calls. It supports pointers but collects only equalities/inequalities between them, while CTGEN supports all regular pointer arithmetic operations.

SAGE [49] (which is built on top of DART), is a very powerful concolic testing tool utilizing white box fuzzing. It is fully automated and is used on a daily basis by Microsoft within the software development process. According to the authors, SAGE uncovered about half of all bugs found in Windows 7. SAGE has a precise memory model, that allows accurate pointer reasoning [41] and is very effective because it works on large applications instead of small units, which allows to detect problems across components. Nevertheless, SAGE uses concrete values for sub-function calls which cannot be symbolically represented and, as far as we know, it does not support the specification of pre- and postconditions.

Pex [100] is an automatic white-box test generation tool for .NET, developed at Microsoft Research. It generates high coverage test suites applying dynamic symbolic execution for parametrized unit tests (PUT). Similarly to CTGEN it uses annotations to define the expected results and the Z3 SMT Solver to decide on the feasibility of execution paths. It also supports complex pointer structures [101]. As long as stubs for external functions are not generated by the user, Pex cannot handle such a call symbolically, while CTGEN recognizes the necessity for a stub and generates it automatically.

Another approach using symbolic execution is applied by KLEE [22], the successor of EXE [23]. KLEE focuses on the interactions of the UUT with the running environment – command-line arguments, files, environment variables etc. It redirects calls accessing the environment to *models*, describing external functions in sufficient depth to allow the generation of the path constraints required. Therefore, KLEE can handle library functions symbolically only if a corresponding model exists, and all unmodelled library and external function calls are executed with concrete values. This may reduce the coverage to be generated due to random testing limitations. Furthermore, KLEE does not provide a fully automated detection of inputs: they must be determined by the user either by code instrumentation or by the command line argument defining the number, size and types of symbolic inputs.

Pathcrawler [18] is also a concolic testing tool. It tries to generate path coverage for C functions. In contrast to CTGEN, it supports only one dimensional arrays and does not support pointer comparisons and external function calls.

Another approach to test data generation in productive industrial environments is based on bounded model checking [8]. The authors used CBMC [26], a Bounded Model Checker for ANSI-C and C++ programs, for the generation of test vectors. The tool supports pointer dereferencing and arithmetic as well as dynamic memory and more. However, since CBMC is applied to generate a test case for each block of the CFG of the UUT, CTGEN is able to achieve full decision coverage with fewer test cases in most situations. For handling external function calls, the authors of [8] use nondeterministic choice functions available in CBCM as stubs, and CBCM evaluates all traces resulting from all possible choices. However, the tool can only simulate return values of external functions and does not consider the possibility of manipulating values of global variables. Though CBMC allows assertions and assumptions in the function body, the authors use them only to achieve branch coverage, not for checking functional properties.

PathFinder [86] is a symbolic execution framework, that uses a model checker to generate and explore different execution paths. PathFinder works on Java byte code, one of its main applications is the production of test data for achieving high code coverage. PathFinder does not address pointer problems since these do not exist in Java. For handling external function calls, the authors propose *mixed concrete-symbolic solving* [85], which is more precise than CTGEN's solution with stubs - it will not generate test data that is impossible in practice. However, mixed concrete-symbolic solving is incomplete, i.e. feasible paths do exist, for which this method fails to find a solution. Furthermore, by definition of the accurate pre- and postconditions the problem regarding impossible inputs can be avoided using CTGEN.

Table 1.2 summarizes the results of our comparison.

1.6 Overview

This thesis is organized as follows: Chapter 2 gives an overview over the test data generator CTGEN developed in the course of this thesis. The architecture of the CTGEN and an example of its invocation are presented. Chapter 3 introduces the annotation language which allows the specification of a module under test. The detailed characterization of the language is given and illustrated by an example. Chapter 4 presents the proposed expansion and selection strategies. Chapter 5 provides an introduction to symbolic execution and discusses its limitations. The memory model that underlies the symbolic execution algorithms is introduced and procedures for reasoning about atomic and complex data types like structures, unions, arrays and pointers are discussed. The algorithms for handling function calls are described. Chapter 6 presents experimental results and the evaluation of the developed test data generator CTGEN.

	CTGEN	PEX	CUTE	KLEE	PathCrawler	CBMC for SCS	DART	SAGE	PathFinder
Platform	Linux	Windows	Linux	Linux	Linux		Linux	Windows	Linux
Language	C	.NET	C	C	C	C	C	machine code	Java
CAPABILITIES									
C0	Y	Y	Y	Y	Y	Y	Y	Y	Y
C1	Y	Y	Y	Y	Y	Y	Y	Y	Y
MC/DC	N	Y	N	N	N	N	N	N	Y
C2	N		Y	Y	Y	N	Y	Y	Y
Pre-/Post	Y	Y	Y	Y	Y	N	N	N	Y
Requirements tracing	Y	N	N	N	N	N	N	N	N
Auxiliary vars	Y	N	NA	N	N	N	N	N	N
Pointer arithmetics	Y	Y	N	Y	Y	Y	N	Y	-
Pointer dereferencing	Y	Y	N	Y	Y	Y	N	Y	-
Pointer comparison	Y	Y	Y	Y	N	Y	N	Y	-
Function pointer	N	NA	N	NA	N	Y	N	NA	
Arrays	Y	Y	Y	Y	P	Y		Y	Y
Symbolic offset	Y	NA	N	Y			N	Y	
Complex dynamic data structures (lists...)	N	Y	Y		N	Y	N		Y
External function calls	Y	P	N	P	N	Y	N	N	P
Automatic stub handling	Y	P	N	N	N	N	N	N	N
Float/double	Y	N	N	N	Y	N	N	Y	Y
Recursion	N	Y	NA	Y	N	Y			Y
Multithreading	N	N	Y	N	N	N	N		Y
Automatic detection of inputs	Y	Y	N	N	Y	Y	Y	Y	
TECHNIQUES									
SMT solver	SONOLAR	Z3	lp_solver	STP	COLIBRI		lp_solver	Z3	choco, IASolver, CVC3
Concolic testing	N	Y	Y	Y	Y	N	Y	Y	N
STCT or acyclic graph with reNuse of nodes	STCT	STCT		application states		transition relation			Y
Depth-first search	N	N	Y	N	Y	N	Y	N	N

Table 1.2: Test Data Generating Tools [72].

2 CTGEN

CTGEN is an automatic test generation tool, based on symbolic execution. The objective of CTGEN is to cover every branch in the program, which is an undecidable problem, so in practice CTGEN tries to generate a test that produces as high a coverage for the module under test as possible. For each UUT CTGEN performs symbolic analysis and generates a test in RT-Tester syntax [44], which can be directly compiled and executed. The test specifies initial values for input parameters, global variables and for the data to be set and returned by sub-functions called by the UUT. Apart from atomic integral data types, CTGEN supports floating point variables, pointer arithmetics, structures and arrays and can cope with the typical aliasing problems in C, caused by array and pointer utilisation. Function pointers, recursive functions, dynamic memory, complex dynamic data structures with pointers (lists, stacks etc.) and concurrent program threads are not supported. CTGEN does not check the module under test for run-time errors but rather delegates this task to the abstract interpreter developed in our research group [82].

CTGEN does not rely on knowledge about all parts of the program (such as undefined or library functions). Where several other unit test automation tools [93, 23, 47] fall back to the invocation of the original sub-function code with concrete inputs if an external function occurs on the explored path, CTGEN automatically generates a *mock object* replacing the external function by a test stub with the same signature. Furthermore, it calculates values for the stub's return data, output parameters and global variables which can be modified by the stubbed function in order to fulfill a path condition. In this way, CTGEN can also simulate exceptional behavior of external functions. It is possible but not required to customize stub behavior by using pre- and postconditions described in Chapter 3. If no restrictions were made, however, the stub's actions can deviate from the real behavior of the external function.

The content of this chapter was originally published in [72]. Here we present a reworked and extended version.

2.1 Architecture

CTGEN is structured into two main components (see Fig. 2.1):

The *preprocessor* operates on the UUT code. It consists of (1) the CTGEN preprocessor transforming code annotations as described in Chapter 3, (2) a `GCC plugin` based on [70], compiling the prepared source code into a textual specification, consisting of one or several Control Flow Graphs (CFGs) in 3-address code, and symbol table information like function signatures, types and variables, and (3) parsers, transforming CFGs and symbol table information into the Intermediate Model Representation (IMR).

The *analyzer* operates on the IMR. Its building blocks and the interaction of these are described below. The *Symbolic Test Case Generator* is responsible for lazy expansion of the CFGs related to the function under test and its sub-functions. Moreover, it handles the selection of paths, each beginning with the start node of the CFG and containing yet uncovered transitions (for more details see Chapter 4). If such a path can be found, it is passed to the *Symbolic Interpreter*, which traverses the path and symbolically

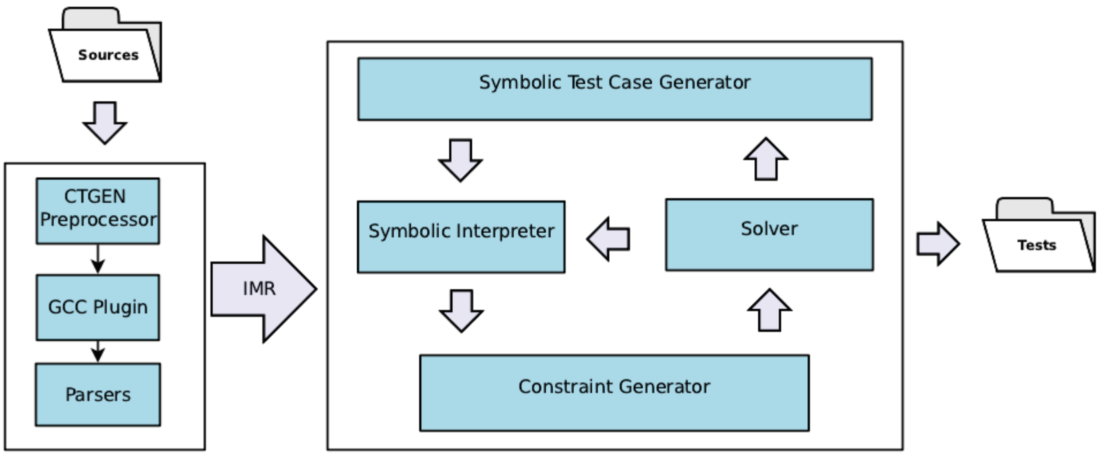


Figure 2.1: CTGEN overview [72].

calculates the effect of its statements in the memory model. As soon as the next node on the path is guarded by a non-trivial condition, the *Constraint Generator* [80] is called and resolves all pointers and array references occurring in this condition. It also passes the resulting constraint to the *Solver*. CTGEN uses a SMT solver (SONOLAR) which has been developed in our research group [82]. SONOLAR supports integral and floating point data types, arrays and bit vectors. If the solver can find a solution for the constraint, the solution is passed back to the Symbolic Interpreter, which continues to follow the path under investigation. Otherwise, if the constraint is infeasible, the solver passes the result to the *Symbolic Test Case Generator*, which then learns from this fact and tries to produce another path containing still uncovered transitions. When no such paths can be found, a unit test is generated based on the collected solutions (if any) and is stored in the file system.

2.2 Invoking CTGEN

In this section we will give an overview of how the CTGEN tool can be invoked and which output it produces. To illustrate the process we will demonstrate how CTGEN is used on a simple example. The program shown in Figure 2.2 contains a trivial implementation of the `checkAvailable()` routine, which sets the global variable `rainActive` to one if and only if global variables `rainSensor` and `rainFunction` have non-zero values and, correspondingly, sets the global variable `solarActive` to one if and only if global variables `solarSensor` and `solarFunction` have non-zero values.

Here we describe the most elementary way of using CTGEN, e.g. without the definition of any pre- or postconditions (this will be discussed later, see Chapter 3). First, the GCC plugin translates the given C code into a textual specification of the CFG and the symbol table information (the plugin output for `checkAvailable()` routine is listed in Appendix 1.2). The CFG characterization contains the description of single blocks and how they relate to each other (for a more detailed discussion of CFG see Section 4.1). Furthermore, location specification and scope information for each statement are documented. The scoping information is required to enable the identification of variables with identical names

```
int rainSensor = 0, rainFunction = 0, rainActive = 0;
int solarSensor = 0, solarFunction = 0, solarActive = 0;

void checkAvailable(){
  if(rainSensor && rainFunction){
    rainActive = 1;
  } else {
    rainActive = 0;
  }
  if(solarSensor && solarFunction){
    solarActive = 1;
  } else {
    solarActive = 0;
  }
}
```

Figure 2.2: A C program that implements `checkAvailable()` routine.

used within a statement, since GCC allows to use variables with identical names in different scopes. The symbol table information includes the list of all used types, all defined global variables and all defined functions. Each function specification contains information about its parameters, return type and all used local variables.

After the CFG and the symbol table information are produced by the plugin, the generator part can be invoked. The generator is called with the following parameters:

```
ctgen --pathForGeneratedTest $TESTPROJECT/unit_test_autogen
--sourceFile cfg_ex.c
```

The parameter `pathForGeneratedTest` defines, where the generated test will be stored. In the example it is stored in a test project in the directory `unit_test_autogen`. The parameter `sourceFile` defines which file should be analyzed. In the example the file `cfg_ex.c` is passed, where function `checkAvailable()` is defined. For more detailed information about the usage of CTGEN see Appendix 7.4.

After the invocation of the test generator, the directory `$TESTPROJECT/unit_test_autogen` has the structure shown in Figure 2.3. For each module, defined in the file `cfg_ex.c` a new unit test is generated. Since in our example the given source file contains only the definition of the `check-Available()` routine, only one new unit test is generated. This test conforms to the RT-Tester syntax and holds three sub-directories: `conf`, `stubs` and `specs`.

The directory `conf` contains the test configuration file (`unit.conf`) and the test documentation input (`unit.rttdoc`). The test configuration file specifies how the executable test case has to be built, where the test specific stubs can be found and the test integration level (here unit test). The test documentation input defines the headline of the test, the test objectives and the description of the test driver. Furthermore, the automated documentation generation derives a verdict for the test from the test execution log.

The directory `stubs` contains the test stub specification file, where generated stubs (when required) are defined. In our example the function `checkAvailable()` does not call any other functions, so that no stubs are generated. Consequently, the `local.stubs` file is empty.

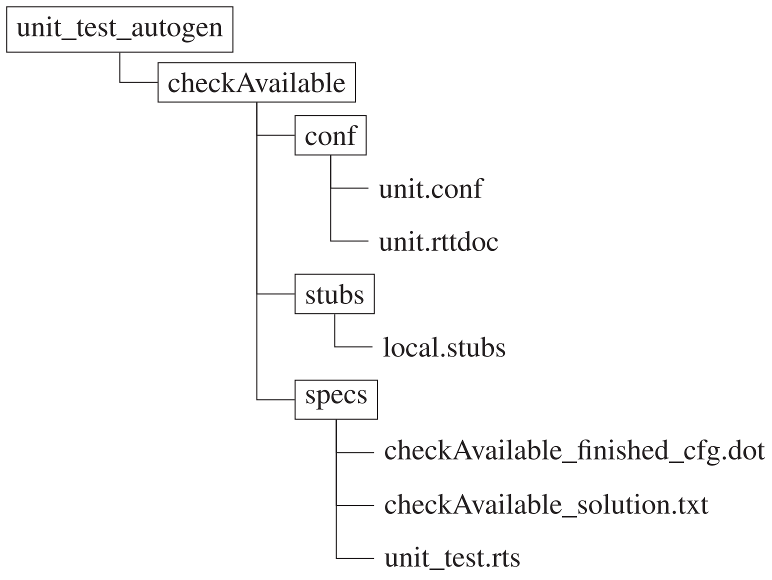


Figure 2.3: Directory structure of the test generated for the `checkAvailable()` routine.

The directory `specs` contains the actual test specification script (`unit_test.rts`). The test script is written in *Real-Time Test Language* (RTTL) (for more information on RTTL see [44]). It defines generated test cases. Each test case defines the values of input variables and invokes the UUT. The test script generated for the `checkAvailable()` function can be observed in Appendix 1.3. Furthermore, the directory `specs` contains the solution file `checkAvailable_solution.txt`, where detailed information on the test generation process can be found. For each test case the chosen path, its path constraint and its solution, found by the SMT solver, are listed. At the bottom of the file statistic information about covered branches can be found. The solution file generated for our example is listed in Appendix 1.4. Additionally, a graphical output of the CFG of the UUT is produced. This graphic visualizes the state of the coverage completion after the generation process is finished. All covered edges and statements are drawn blue, all statements and edges that could not be covered are drawn red. The `checkAvailable()` routine from our example could be completely covered. Therefore, all statements and edges are drawn blue (see Appendix 1.5).

The generated test can be compiled and executed with RT-Tester. To measure the actual code coverage we use `gcov`. Executing tests independently of CTGEN excludes the influence of potential bugs in CTGEN and verifies that the generated test runs the code as was claimed by CTGEN.

3 Annotation Language

Assertional methods for program verification were introduced by Floyd [43] in the late sixties. His ideas were refined by Hoare [56] and Dijkstra [36, 37]. The main idea of this approach can be described as follows: if the precondition of a program is true before the program is executed, the postcondition must hold true. The annotation language that we present in this chapter was introduced in [72]. This annotation language makes use of the assertional methods mentioned above and allows users to specify the expected behavior of a module under test by means of appropriate pre- and postconditions, to refine the specification with help of auxiliary variables, to introduce the functional coverage by the definition of test cases relating pre- and postconditions to the corresponding requirements and to reason about global variables, initial values of variables and return values of a module under test.

Some of the contents of this chapter were already introduced in [72]. However, we here present an extended and refined version.

3.1 Definition

For the definition of the annotation language we have chosen the approach used in sixgill [51]: the annotations are specified as GCC macros which are understood by the CTGEN preprocessor. Thus, the annotations can be turned on and off as needed. One of the critics on formal methods is that the overhead needed to learn the techniques and the formal languages is too time consuming [52]. Therefore, we have decided to keep the annotations in standard C syntax, so that no additional expertise is expected from the user. All annotations are optional. If there are no annotations, CTGEN will try to cover all branches and detect unreachable code, using arbitrary type-compatible input data.

Pre- and postconditions are defined as follows:

```
__rtt_precondition(PRE);
__rtt_postcondition(POST);
```

A precondition indicates, that the expected behavior of the specified function is only guaranteed if the condition `PRE` is true. A postcondition specifies, that after the execution of a function the condition `POST` must hold. Furthermore, (as discussed in Section 5.12.3) pre- and postconditions also affect stub generation in CTGEN. Pre- and postconditions have to be defined at the beginning of the body of a function. `PRE` and `POST` are Boolean C expressions, including function calls. All variables occurring in these conditions must be global, be input respectively output parameters or refer to the return value of the specified function. To specify conditions involving the return value of the UUT the CTGEN variable

```
  __rtt_return
```

is introduced. The annotation

```
__rtt_initial(VARNAME);
```

is used in annotation expressions (in particular, in postconditions) for referring to the initial value of the variable VARNAME, valid before the function was executed.

To reason over local variables, auxiliary variables are used. Auxiliary variables cannot occur in assignments to non-auxiliary variables or in control flow conditions [9, 80]. They can be defined as follows:

```
__rtt_aux(TYPE, VARNAME);
```

In this way, an auxiliary variable of the type TYPE with the name VARNAME will be declared and can be used in the following CTGEN annotations in the same way as regular variables.

For a more detailed specification of the expected behavior of the function, test cases are used:

```
__rtt_testcase(PRE, POST, REQ);
```

The argument PRE defines a precondition and the argument POST a postcondition of the current test case. The argument REQ is a string tag defining a functional requirement that corresponds to the pre- and postcondition of this test case. If there is more than one, the requirements can be listed separated by a comma. For each generated test data set that satisfies a precondition from the test case assertions over pre- and postconditions will automatically be inserted into the generated test:

```
/** @rttPrint
  * This test case evaluates whether the function example()
  * behaves correctly
  * @tag TC_UNIT_EXAMPLE_001
  * @condition PRE
  * @event The unit under test example() is called.
  * @expected POST
  * @req REQ
  */
@rttAssert(PRE, "TC_UNIT_EXAMPLE_001");
@rttCall(example());
@rttAssert(POST, "TC_UNIT_EXAMPLE_001");
```

Global variables which are allowed to be modified in a function can be specified by means of the annotation:

```
__rtt_modifies(VARNAME);
```

CTGEN traces violations, even in cases where a prohibited variable is modified by means of pointer dereferencing. For each breach of a modification rule an assertion is generated, which records the line number where the illegal modification occurred, e. g.

```
// violated var VARNAME in line(s) 1212, 1284
@rttAssert(FALSE);
```

The annotation

```
  __rtt_assign(ASSIGNMENT);
```

is intended for assignments to auxiliary variables. In the following example an auxiliary variable a_aux is first declared using __rtt_aux() it may then be used in a postcondition. To define its value, __rtt_assign() is used in the function body.

```
__rtt_aux(int, a_aux);
__rtt_postcondition(a_aux == 0);
...
int b;
...
__rtt_assign(a_aux = b);
```

The annotation

```
  __rtt_assert(COND);
```

can be used in different places of the function to ensure a specific property. If the condition `COND` is seen to fail during test generation an assertion recording the line number where the violation occurs is inserted into the generated test.

An example of a specification of the expected behavior of a function is illustrated in Figure 3.1 (the original code is highlighted in light gray, inserted annotations have a white background). The function `alloc()` returns a pointer `allocp` to `n` successive characters if there is still enough room in the buffer `allocbuf` and zero if this is not the case. First, by using `__rtt_modifies` we state that `alloc()` can only modify `allocp`, and that the modification of `allocbuf` is consequently prohibited. The annotation `__rtt_precondition` specifies that the expected behavior of `alloc()` is guaranteed only if the parameter `n` is greater or equal to zero and `allocp` is not a `NULL`-pointer. Furthermore, `__rtt_postcondition` states that after the execution of the function under test `allocp` must still be within the bounds of the array `allocbuf`. Finally, test cases are defined for the situations where (a) memory can still be allocated and (b) not enough memory is available.

After preprocessing by CTGEN, this example looks as shown in Figure 3.2 (the original code is highlighted in light gray, code corresponding to preprocessed annotations has a white background): the function body is executed only if the precondition holds. Concerning the postcondition and test cases, appropriate `if` statements are generated with branches for both outcomes: when the test case (postcondition) fails and also when it passes. The test driver generated by CTGEN for this example as well as other produced outputs are listed in Appendix 2.

3.2 Proof Mode versus normal Test Mode.

Using symbolic execution for program verification was proposed in 1970's [65, 35]. To prove the verification condition it is sufficient to execute all program paths of a preprocessed annotated UUT symbolically. As pointed out in [80], this way the verification problem is reduced to a reachability problem: a test data set reaching `__rtt_testcase_error` (see Figure 3.2) at the same time uncovers a violation of the defined properties and produces a counter example. Otherwise, if it can be shown that `__rtt_testcase_error` is unreachable, this proves the validity of the corresponding test case. Furthermore, the goal to obtain functional test coverage is also reduced to reaching structural test coverage, because branch coverage implies a coverage of requirements mentioned in the test cases.

However, in order to prove that some branch is unreachable, all paths through the function under test must be explored. This is known to be a problem of exponential complexity. Since this approach is not feasible for all functions, CTGEN allows to choose between *proof mode* and *normal test mode*. In proof mode CTGEN tries to prove, that no postcondition or test case condition violation is possible, whereas in normal test mode it attempts to cover only branches which document the test case execution.

```
char allocbuf[ALLOCSIZE];
char *allocp = allocbuf;

char *alloc(int n){
  __rtt_modifies(allocp);
  __rtt_precondition(n >= 0 && allocp != 0);
  __rtt_postcondition(allocp != 0 && allocp <= allocbuf + ALLOCSIZE);
  __rtt_testcase(allocbuf + ALLOCSIZE - __rtt_initial(allocp) < n,
                 __rtt_return == 0,
                 "CTGEN_001");
  __rtt_testcase(allocbuf + ALLOCSIZE - __rtt_initial(allocp) >= n,
                 __rtt_return == __rtt_initial(allocp),
                 "CTGEN_002");
  char *retval = 0;
  if(allocbuf + ALLOCSIZE - allocp >= n){
    allocp += n;
    retval = allocp - n;
  }

  return retval;
}
```

Figure 3.1: Example: Specification of expected behavior.

```
char allocbuf[ALLOCSIZE];
char *allocp = allocbuf;

char *alloc(int n){
  char * __rtt_return__;
  __rtt_modifies(allocp);
  // precondition
  if(n >= 0 && allocp != 0){
    char *retval = 0;
    if(allocbuf + ALLOCSIZE - allocp >= n){
      allocp += n;
      retval = allocp - n;
    }

    // postcondition
    if(allocp != 0 && allocp <= allocbuf + ALLOCSIZE){
      __rtt_testcase(n >= 0 && allocp != 0,
                     allocp != 0 && allocp <= allocbuf + ALLOCSIZE, "");
    } else {
      __rtt_testcase_error(n >= 0 && allocp != 0,
                           allocp != 0 && allocp <= allocbuf +
                           ALLOCSIZE, "");
    }
    // testcase 1
    if(allocbuf + ALLOCSIZE - __rtt_initial(allocp) < n){
      if(__rtt_return == 0){
        __rtt_testcase(allocbuf + ALLOCSIZE - __rtt_initial(allocp) < n,
                       __rtt_return == 0,
                       "CTGEN_001");
      } else {
        __rtt_testcase_error(allocbuf + ALLOCSIZE - __rtt_initial(allocp) < n,
                             __rtt_return == 0,
                             "CTGEN_001");
      }
    }
    // testcase 2
    if(allocbuf + ALLOCSIZE - __rtt_initial(allocp) >= n){
      if(__rtt_return == __rtt_initial(allocp)){
        __rtt_testcase(allocbuf + ALLOCSIZE - __rtt_initial(allocp) >= n,
                       __rtt_return == __rtt_initial(allocp),
                       "CTGEN_002");
      } else {
        __rtt_testcase_error(allocbuf + ALLOCSIZE - __rtt_initial(allocp) >= n,
                             __rtt_return == __rtt_initial(allocp),
                             "CTGEN_002");
      }
    }
  }

  return retval;
}
```

Figure 3.2: Preprocessed specification from Figure 3.1.

4 Symbolic Test Case Generation

As described in Section 2.1, the symbolic test case generator is responsible for the selection of test cases, which the symbolic interpreter supported by the solver tries to cover. The symbolic test case generator uses feedback of the solver about eventual infeasibility of a symbolic test case for learning to avoid detected infeasible paths in the following test case generations. A central data structure in the symbolic test case generator is a *Symbolic Test Case Tree* (STCT), which stores bounded paths through the *control flow graph* (CFG) of the UUT. To build a STCT the CFG is expanded node by node. To minimize the size of a STCT and maximize the coverage obtained by generated test cases, we introduce expansion and selection strategies.

This chapter provides formal definitions of the CFG, the STCT and the algorithms utilized by the proposed expansion and selection strategies.

4.1 Control Flow Graph

The Control Flow Graph (CFG) represents the control flow of programs. Nodes of a CFG are statements of the respective program, whereas edges represent the control flow between the statements. There are several approaches regarding the definition of CFGs. They differ in the level of abstraction of the representation of sections of statements which are always executed consecutively and the handling of branches as well as the merging of branches [50]. We define control flow graph as follows:

Definition 4.1. *Control Flow Graph* of function f we define here as a directed cyclic Graph $CFG = (N_C, E_C, V)$ where

- N_C is a finite set of nodes, so that for each GIMPLE statement[1] s of f there $\exists n_s \in N_C$,
- V is a finite set of variables of f,
- *Guard* is a set of Boolean formulae over V,
- $E_C \subseteq N_C \times Guard \times N_C$ is a finite set of edges, so that for each pair of statements (s, s') such that s' is executed directly after s, there $\exists e \in E_C$, where e is a tuple $(n_s, g, n_{s'})$. Node n_s we call *source* of the edge e and node $n_{s'}$ – *target* of the edge e. g is the guard condition of the edge e, which means, that the edge e can be taken by the program execution only if the condition g evaluates to $true$. Node n_s is called *predecessor* of node $n_{s'}$ and node $n_{s'}$ – *successor* of node n_s,

The node corresponding to the first statement of f is called *start node*. The node corresponding to the last statement of f is called *exit node*.

[1]Under a GIMPLE statement we understand a three-address statement representation [1].

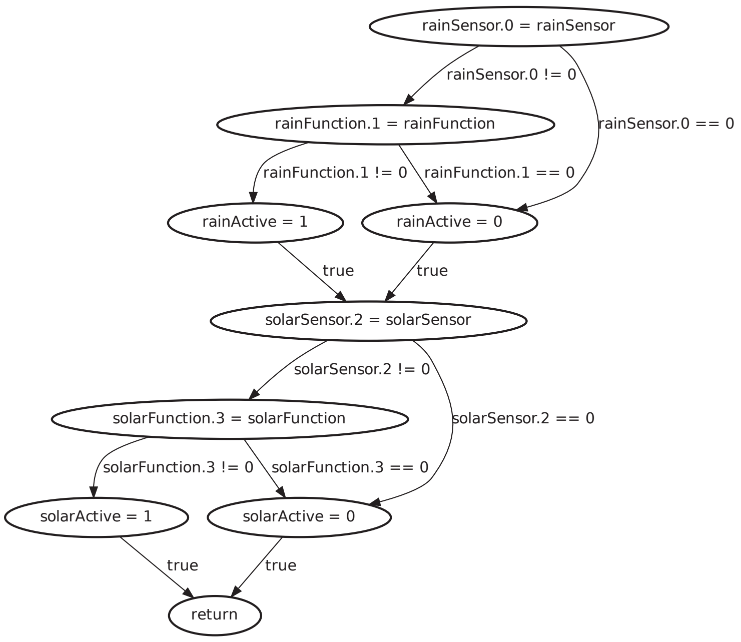


Figure 4.1: Control flow graph of the `checkAvailable()` routine.

To illustrate the definition of a CFG, we use the example discussed in Section 2.2. Figure 4.1 shows the corresponding CFG after converting the `checkAvailable()` routine into 3-address code. The start node corresponds to the first statement of the function (`rainSensor.0 = rainSensor`) and the exit node corresponds to the `return` statement.

Definition 4.2. *A path* p is a finite sequence of edges $< e_0, e_1, \ldots e_n >$ so that for all i the target of the edge e_i is the source of the edge e_{i+1}. The source node of the first edge e_0 is the *start node* of p, the target node of the last edge e_n is the *end node* of p. If the start node of the path is the start node of the CFG, the path is called *S-path*. An *S-path* whose end node is the exit node of the CFG is called *complete* [50].

Since a CFG is a cyclic graph, an infinite number of paths through a function exists. In this thesis we consider only S-paths.

4.2 Symbolic Test Case Tree

STCT is a central data structure in the symbolic test case generator of CTGEN. It stores bounded paths through the CFG. During expansion of a CFG node n, each outgoing edge of n is analyzed and each of its target nodes receives a new corresponding STCT leaf, even if this target node has already been expanded. The nodes are labeled with a number k, so that (n,k) is a unique identifier of a STCT node, while n may occur several times in the STCT if it can be reached on different (or the same cyclic) CFG paths. The STCT root corresponds to the CFG start node [12].

Definition 4.3. Given a $CFG = (N_C, E_C, V)$, associated $STCT_l$ of length $l \in \mathbb{N}$ is a tuple

$$STCT_l = (N_l, E_l, L_l, \varphi_l, \sigma_l, \psi_l, \eta_l)$$

where

- $N_l \subseteq N_C \times \mathbb{N}$ is a finite subset of tree nodes corresponding to CFG nodes which can be reached with a path of length l,
- $E_l \subseteq N_l \times Guard \times N_l$ is a finite subset of tree edges corresponding to CFG edges which can be reached with a path of length l,
- $L_l \subseteq N_l$ is a set of leaves of the tree,
- $\varphi_l : N_l \to N_C$ is a function that maps nodes of STCT to the corresponding nodes of the CFG,
- $\sigma_l : N_C \to \mathbb{N}$ is a function that keeps track of the number of the STCT nodes corresponding to each CFG node,
- $\psi_l : E_C \to E_l^*$ is a function that maps edges of the CFG to the list of corresponding edges in STCT. It can also be empty if the source node of the edge is not expanded yet.
- $\eta_l : E_l \to E_C$ is a function that maps edges of the STCT to the corresponding edges of the CFG.

STCT of an arbitrary length we denote as $STCT = (N, E, L, \varphi, \sigma, \psi, \eta)$.

To illustrate the definition of the STCT, we abstract the CFG from Figure 4.1 by ignoring all guards which have the value *true* and by replacing the remaining guards through Boolean variables. The node labels are also simplified (see Figure 4.2). The fully expanded STCT of the abstracted graph is shown in Figure 4.3. Node n_4 can be reached on three different paths: $< (n_0, a, n_1), (n_1, b, n_2), (n_2, \varepsilon, n_4) >$, $< (n_0, a, n_1), (n_1, !b, n_3), (n_3, \varepsilon, n_4) >$ and $< (n_0, !a, n_3), (n_3, \varepsilon, n_4) >$. Thus, the STCT has three corresponding nodes: $(n_4, 0)$, $(n_4, 1)$ and $(n_4, 2)$. And corresponding $\varphi(n_4, 0) = n_4$ and $\sigma(n_4) = 3$. Furthermore, since nodes n_4 and n_5 occur several times in the STCT, edge (n_4, c, n_5) occurs several times as well. So $\psi(n_4, c, n_5) = \{((n_4, 0), c, (n_5, 0)),\ ((n_4, 1), c, (n_5, 1)),\ ((n_4, 2), c, (n_5, 2))\}$ and, corresponding, $\eta((n_4, 1), c, (n_5, 1)) = (n_4, c, n_5)$.

4.3 Expansion and Selection Strategies

After the definitions of the CFG and the STCT are introduced, we discuss the proposed expansion and selection strategies. The content of this section was originally published in [72], so here we present an extended version with detailed examples and algorithm specifications.

To select a new test case, the symbolic test case generator takes an edge in the CFG which is still uncovered. Subsequently, it finds a corresponding STCT edge and follows it bottom-up to the start node. The path is then returned to the test data generator by the symbolic test case generator for further investigation.

The depth-first search used by several test generating tools [18, 93, 47, 23] allows reusing the information of the shared part of the execution path, but on the other hand can cause the generator to get

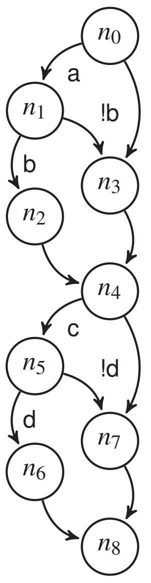


Figure 4.2: Abstracted control flow graph of the `checkAvailable()` routine.

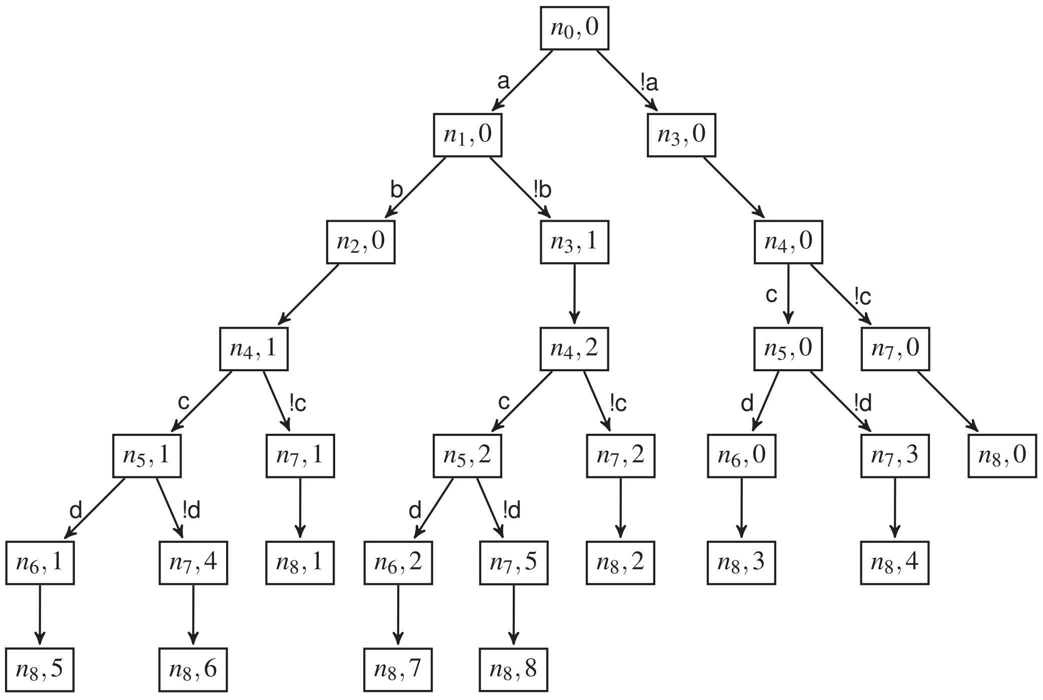


Figure 4.3: Symbolic test case tree of the `checkAvailable()` routine.

"stuck" analyzing a small part of the program and generating a lot of new test cases but no (or little) new coverage. Pex [100] avoids using depth-first and backtracking techniques by storing the information of all previously executed paths. CTGEN behaves similarly: during the symbolic execution of a path it stores information already gained in the form of computation histories and path constraints associated with each branching point of the path under consideration.

Our expansion and selection strategies are motivated as follows:

- The larger the explored range of the variable values participating in the guard conditions closest to the start node, the higher the probability to cover more branches depending on these variables further down in the CFG. So we prioritize edges after their proximity to the start node. This allows to achieve more coverage when it is not possible to explore the function completely due to its size. Furthermore, this approach minimizes the number of paths that must be explored to achieve 100% C1 coverage, which in turn reduces the overall time for generation.

- A path is interesting for the test case generation for achieving C1 coverage only if it contains edges that are still uncovered with a non-trivial guard condition. Otherwise no new coverage can be achieved by interpreting this path. We expand until the STCT contains a new uncovered edge whose guard condition is not always *true*, or until no expansion can find any additional uncovered edges. At the same time we try to minimize the size of the STCT. We stop the expansion process as soon as an uncovered edge with a non-trivial guard condition occurs. We call this approach *incremental expansion*.

- To further minimize the size of the STCT we incrementally expand only the end node of the path under consideration (initially the root node), select a continuation for it and hand it over to the solver. We continue in a step by step manner until the path is complete. After that and according to prioritization of edges, a new path is selected. If the selected path is infeasible, the responsible branch is deleted from the STCT and the selection and expansion process is continued with the alternative branch.

The loop constructs are unwinded according to our expansion strategy: the loop body is incrementally expanded until the exit condition is reached. If the exit condition can be evaluated to *true*, the loop is exited. Otherwise it is further expanded. This process is bounded by a configurable parameter that defines the maximum possible depth of the STCT.

The incremental approach allows us to use *incremental solving* supported by the solver SONOLAR. By this approach the feasibility constraint is not sent all at once but only the constraint corresponding to the last guard condition. That, in turn, allows us to execute at least a part of the path for which the complete feasibility constraint would exceed the possibilities of the solver symbolically.

A simple example of our expansion and selection strategy for the CFG from Figure 4.2 is illustrated in Figure 4.4. After the initial expansion (Figure 4.4(a)) a path $<((n_0,0),\, a,\, (n_1,0))>$ is selected (nodes and edges that belong to the path are drawn blue, those that do not belong are drawn black). After the path is interpreted and evaluated as feasible, its last node $(n_1,0)$ is expanded (Figure 4.4(b)) and a new path (continuation of the last one) $<((n_0,0),\, a,\, (n_1,0)),\, ((n_1,0)),\, b,\, (n_2,0))>$ is selected. After this path is interpreted as well, its last node $(n_2,0)$ is expanded until the new edge with a guard condition appears (Figure 4.4(c)). Afterwards, a new path (continuation of the previous one) is selected. This

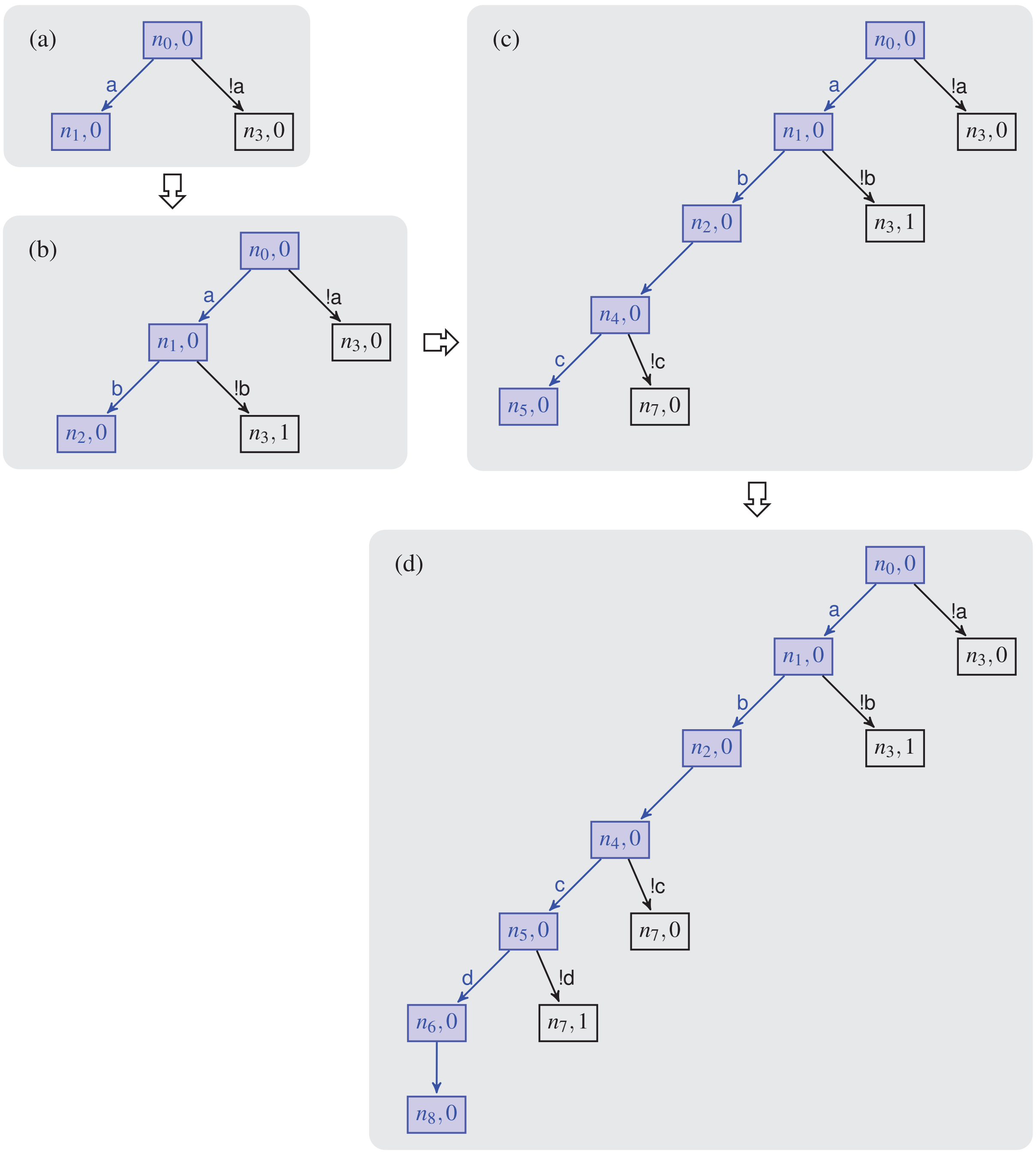


Figure 4.4: Expansion/Selection example.

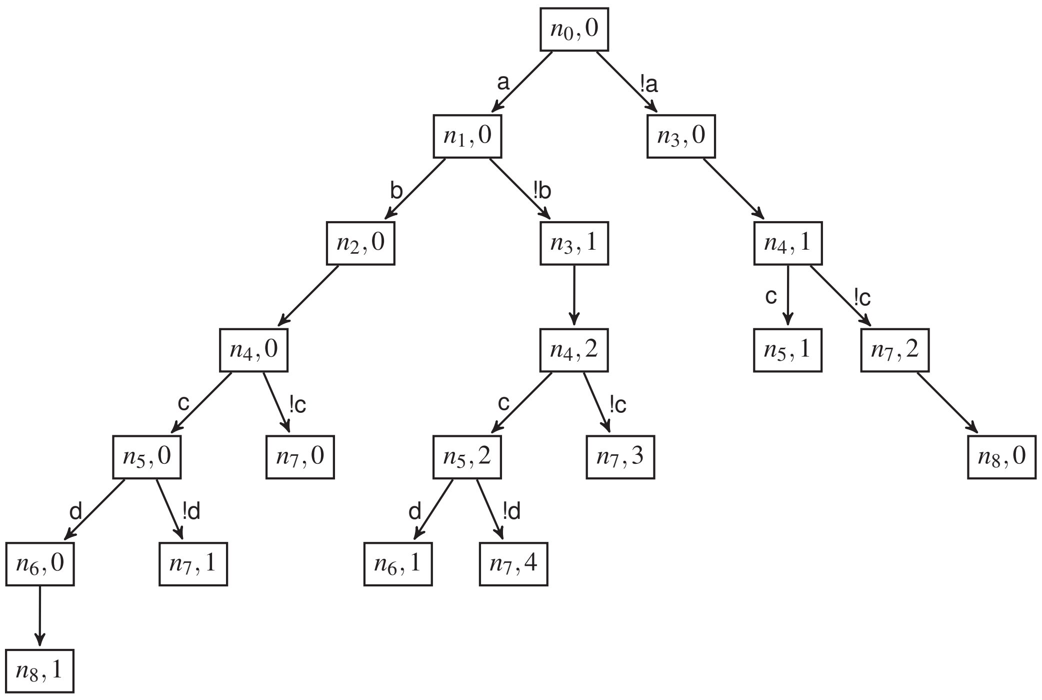


Figure 4.5: Expansion/Selection example (final).

process continues until the path is complete (Figure 4.4(d)). Finally a new path, corresponding to the prioritization of edges, is selected (here $< ((n_0,0),\ !a,\ (n_3,0)) >$) and the whole process is repeated.

Figure 4.5 shows the final version of the STCT expanded corresponding to our algorithm. Compared to the fully expanded STCT from Figure 4.3 it is smaller in size and with growing size of the CFG this difference also increases. Furthermore, for functions whose size makes it impossible to fully expand their STCT, our strategies enable us to explore the path through the function until the end and not only the first n steps after which the expansion is not possible anymore. The ability to explore the whole path gives the generator more flexibility and allows CTGEN to maximize the coverage in the last steps of the path in contrast to an exploration of only the first n steps.

After having described expansion and selection strategies informally, we consider them in a more systematic way in the next sections.

4.3.1 Incremental Expansion Algorithm

Initialization of $STCT_0 = (N_0, E_0, L_0, \varphi_0, \sigma_0, \psi_0, \eta_0)$ is done as follows:

$$N_0 = \{(s,0)\}$$
$$E_0 = \emptyset$$
$$L_0 = \{(s,0)\}$$
$$\varphi_0(v) = start, \quad v = (s,0)$$
$$\sigma_0(v) = \begin{cases} 1, & v = start \\ 0, & \forall v \in N_C \setminus \{start\} \end{cases}$$
$$\psi_0(e) = \emptyset, \quad \forall e \in E_C$$
$$\eta_0(e) = \emptyset, \quad \forall e \in E_0$$

where $start$ denotes the start node of the CFG and $(s,0)$ the corresponding STCT node.

Algorithm 1 shows the function *expandLeaf()* that performs our incremental expansion algorithm. Its inputs are the CFG, the existing STCT and the leaf to be expanded. It modifies the STCT by expanding the given leaf until the STCT receives a new non-trivial guard condition and at least one uncovered edge. If it is not possible to expand further, we have reached the end of the function or the maximum number of permitted expansions.

The example from Figure 4.4 illustrates the proposed algorithm. Presume the STCT is expanded as is shown in Figure 4.4(b). Suppose we call our algorithm *expandLeaf()* with $(n_2,0)$ as a leaf to expand. Suppose further, that maximal permitted size of the STCT is big, so that we cannot reach it with the given CFG. In the first iteration all outgoing edges of the corresponding CFG node n_2 are considered. In our example n_2 has only one outgoing edge: (n_2,ε,n_4) (see Figure 4.1). The node n_4 was not expanded yet, hence its counter is zero, so a new leaf $(n_4,0)$ is created and a new edge $((n_2,0),\varepsilon,(n_4,0))$ as well as the new leaf is added to the STCT. Although the edge (n_2,ε,n_4) is still uncovered, its guard condition is trivial, and so the loop exit condition is not fulfilled (remember, the maximum size is unreachable in our example and the working set `S` contains the new leaf $(n_4,0)$). So the expansion is repeated, this time all outgoing edges of the node n_4 – (n_4,c,n_5) and $(n_4,!c,n_7)$ – are considered. Two new leaves $(n_5,0)$ and $(n_7,0)$ are created and two new edges $((n_4,0),c,(n_5,0))$ and $((n_4,0),!c,(n_7,0))$ as well as newly created leaves are added to the STCT. This time considered edges have nontrivial guard conditions, and so the loop exit condition is evaluated to $true$.

```
inout: $STCT = (N, E, L, \varphi, \sigma, \psi, \eta)$;
input: $(l,k)$ a leaf from $STCT$;
       $CFG = (N_C, E_C, V)$;
output: expanded - indicator if expansion was successful
function expandLeaf($(l,k)$, $CFG$, $STCT$){
  newGuard = false;
  notCovered = false;
  expanded = false;
  S = empty;
  S = S.push($(l,k)$);
  if(!maxSize){
    repeat{
      $(n,k)$ = S.pop();
      $n = \varphi((n,k))$; // get the corresponding CFG node

      // consider its successors
      forall $e = (n, g, n')$ in $E_C$ {
        expanded = true;

        //create new leaf $(n',k')$ corresponding to the CFG node $n'$
        $k' = \sigma(n')$;
        S = S.push($(n',k')$);

        // update STCT
        $N = N \cup \{(n',k')\}$; // add new node
        $E = E \cup \{((n,k), g, (n',k'))\}$; // add new edge
        $L = (L \setminus \{(n,k)\}) \cup \{(n',k')\}$; // erase expanded and add new leaf

        // adjust domains of STCT functions
        $D(\varphi) = D(\varphi) \cup \{(n',k')\}$;
        $D(\eta) = D(\eta) \cup \{((n,k), g, (n',k'))\}$;

        $\varphi((n',k')) = n'$;
        $\sigma(n') = k' + 1$;
        $\psi(e)$ = $\psi(e) \cup \{((n,k), g, (n',k'))\}$;
        $\eta(((n,k), g, (n',k'))) = e$;

        if($e$ not covered){
          notCovered = true;
        }
        // check the guard condition of edge e
        if($g$ not const true){
          newGuard = true;
        }
      }
    }until (S.empty() || (notCovered && newGuard) || maxSize)
  }
  return expanded;
}
```

Algorithm 1: Incremental expansion algorithm.

```
inout: $STCT = (N,E,L,\varphi,\sigma,\psi,\eta)$;
input: $CFG = (N_C,E_C,V)$;
output: expanded - indicator if expansion was successful
function expandAll($STCT$, $CFG$){
  expanded = false;
  if(maximal size is not reached){
    forall $l$ in $L$ {
      expanded =  expanded || expandLeaf($l$, $CFG$, $STCT$);
    }
  }
  return expanded;
}
```

Algorithm 2: Expansion algorithm.

The presented algorithm *expandLeaf()* attempts to expand only one STCT leaf which it receives as input. In case that the selection algorithm (discussed in Section 4.3.2) is not able to find any path containing still uncovered edges with non-trivial guard condition, the function $expandAll()$ is invoked (Algorithm 2). If the maximum size of the STCT is not reached yet, this function attempts to expand all leaves of the STCT and reports success if at least one of them could be expanded.

4.3.2 Path Selection Algorithm

We distinguish two path selection algorithms: (1) the algorithm for the selection of an initial path e.g. at the very beginning of the generation process or after the path under consideration was completed and (2) the algorithm for selection of a continuation of the path under consideration.

First, we consider the algorithm $select()$, which handles the selection of an initial path (see Algorithm 3). It receives the existing STCT as input, the set of still uncovered CFG edges U and an indicator *expanded* which notifies whether the last STCT expansion was successful or not. The edges in set U are sorted according to their priorities. This algorithm distinguishes two cases: (1) when the last expansion of the STCT was successful and (2) when it was not successful, i.e. that the STCT was already fully expanded or has reached its maximum allowed size. If the last expansion was successful, the algorithm traverses over all uncovered edges. If an edge has a non-trivial guard condition and was already expanded (this means there is at least one corresponding edge in the STCT), an S-path is selected. The end node of the selected path is the target node of the chosen edge. The second case reflects the situation when after a successful expansion it was already attempted to select a path with a new, not already evaluated path constraint (case (1)). This attempt fails and the attempt to expand the STCT further fails as well (this can happen, e.g. when the function under consideration has only sequential code without any branches). In this case the algorithm tries to select the longest path possible. For this reason, it traverses the set of uncovered edges U in reversed order, to test the edges farthermost from the start node first. Here it is not important anymore if the guard condition of the edge is trivial or not.

The function $selectContinuation()$ for the selection of a continuation of the path under consideration is shown in Algorithm 4. Similarly to the previous algorithm it receives as inputs the existing STCT, the set of still uncovered CFG edges U and an indicator *expanded* which notifies whether the last STCT expansion was successful or not. Additionally, a path α is passed, which should be extended. First, the

```
input:  $STCT = (N,E,L,\varphi,\sigma,\psi,\eta)$
        $U \subseteq E_C$ − uncovered edges from CFG
        expanded − indicator if the last expansion of the STCT was successful
output: selected path $p$
function select($STCT$, $U$, expanded){
  $p$ = NULL;
  if(expanded){
    forall $e=(u,g,v)$ in $U$ {
      if($g$ not const true){
        forall $e'=(u', g, v')$ in $\psi(e)$ { // for all corresponding STCT edges
          $p$ = $<(s,g_0,w), \ldots e'>$; // path from start node till $e'$
          return $p$;
        }
      }
    }
  } else {
    forall $e=(u,g,v)$ in $U$ in reversed order {
      forall $e'=(u', g, v') \in \psi(e)$ { // for all corresponding STCT edges
        $p$ = $<(s,g_0,w), \ldots e'>$; // path from start node till $e'$
        return $p$;
      }
    }
  }
  return $p$;
}
```

Algorithm 3: Path selection algorithm.

algorithm checks whether the path α is defined. If not, the already discussed function *select*() is invoked and the path, selected by it, is returned. Otherwise, if the path α is defined, the end node of this path is considered and all its outgoing STCT edges are put into the working queue Q. This queue is traversed and, if the corresponding CFG edge is uncovered, an S-path p is selected, whose end node is the target node of the chosen edge. If the uncovered edge has a non-trivial guard condition, the selected path p is returned, otherwise *selectContinuation*() with p as path to expand is called. This is done because we always try to select a path with a new, not already evaluated path constraint and only a new not trivial guard condition can cause infeasibility. When the recursive call of *selectContinuation*() was successful, the newly selected path is returned, or p otherwise. If the corresponding STCT edge is already covered, its target node is examined and all outgoing STCT edges are put into the working queue Q. In this way either a following uncovered edge will be found or the queue will finally be empty since we are working on a STCT, which is finite and has no cycles.

```
input: $STCT = (N,E,L,\varphi,\sigma,\psi,\eta)$
       $U \subseteq E_C$ - uncovered edges from CFG
       $\alpha$ - initial path
       expanded - indicator if the last expansion of the STCT was successful
output: selected path $p$
function selectContinuation($\alpha$, $STCT$, $U$, expanded){
  $p$ = NULL;
  if($\alpha$ is NULL){
    return select($STCT$, $U$, expanded);
  }
  $w$ = end node of $\alpha$;
  Q = $\{\forall (u,g,v) \in E: \; u = w\}$ // all STCT edges with source w

  while(!Q.empty()){
    $(u,g,v)$ = Q.pop();
    if( $\eta((u,g,v)) \in U$ ){ // if corresponding CFG edge is not covered
      $p$ = $<(s,g_0,u'), \dots (u,g,v)>$; // path from start node till considered edge
      if($g$ not const true){
        return $p$;
      } else {
        $p_{new}$ = selectContinuation($p$);
        if($p_{new}$) $p$ = $p_{new}$;
        return $p$;
      }
    } else {
      $Q = Q \cup \{\forall (u',g,v') \in E: \; u' = v\}$ // add all STCT edges with source v
    }

  }
  return $p$;
}
```

Algorithm 4: Path continuation selection algorithm.

We illustrate the discussed algorithms on an example from Figure 4.4. Initially *selectContinuation*() is called with input α equal to `NULL`. Function *select*() is called and, since the edge $((n_0,0),\, a,\, (n_1,0))$ has the highest priority, is uncovered and has a non-trivial guard condition, the path $p_0 =< ((n_0,0),\, a,$

$(n_1,0)) >$ is selected (Figure 4.4(a)). To select the next path *selectContinuation*() is called with input p_0 and the path $p_1 =< ((n_0,0),\ a,\ (n_1,0)),\ ((n_1,0)),\ b,\ (n_2,0)) >$ is selected (Figure 4.4(b)) already after the first iteration because the edge $((n_1,0),\ b,\ (n_2,0))$ is uncovered and has a non-trivial guard condition. Its priority is not important anymore since now we search for a continuation of the existing path. For the next selection *selectContinuation*() is called with the input p_1 and the path $p_2 =< ((n_0,0),\ a,\ (n_1,0)),$ $((n_1,0)),\ b,\ (n_2,0)),((n_2,0),\ \varepsilon,\ (n_4,0)),((n_4,0),c,(n_5,0)) >$ is selected (Figure 4.4(c)). This time it has required the recursive call of *selectContinuation*() with input $p_{2'} =< ((n_0,0),\ a,\ (n_1,0)),\ ((n_1,0)),\ b,$ $(n_2,0)),((n_2,0),\ \varepsilon,\ (n_4,0)) >$, since the edge $((n_2,0),\ \varepsilon,\ (n_4,0))$ is uncovered but its guard condition is trivial.

4.3.3 Pruning of Infeasible Branches

Until now we only have considered situations when the solver was able to find a variable set that satisfies the derived path constraint so that the selected path could be continued. Here we assume the case when the solver declares the passed constraint as infeasible. To learn from this fact and to avoid the infeasible path in further generations, all STCT edges, beginning with the edge which caused the infeasibility are pruned. This task is performed by the algorithm *prune*(). It takes a STCT and an edge as inputs and removes the given edge and all its followers from the STCT. Since the source of the pruned edge was already expanded, it is not a leaf anymore and hence will not be considered for further expansions. In this way, the infeasible edge is definitely erased from the STCT and cannot occur on any path selected for further generations.

```
inout: $STCT = (N,E,L,\varphi,\sigma,\psi,\eta)$
input: edge $e = (u, g, v)$ from $E$;
procedure prune($e$, $STCT$){
  // for all following edges
  forall $e' = (v, g', w)$ in E{
    prune($e'$, $STCT$);
  }
  $N = N \setminus v$;
  $E = E \setminus e$;
  if($v \in L$) $L = L \setminus v$;

  $e_C = \eta(e)$; // find out corresponding CFG edge
  $\psi(e_C)$ = $\psi(e_C) \setminus \{e\}$;

  // adjust domains of STCT functions
  $D(\varphi) = D(\varphi) \setminus \{v\}$;
  $D(\eta) = D(\eta) \setminus \{e\}$;
}
```

Algorithm 5: Pruning algorithm.

4.3.4 Execution of the Selected Path

During the symbolic execution of a selected path, the collected information in form of a memory specification and path constraints is stored for future executions. The memory specification is stored at the branching points of the STCT and holds the memory configuration which is the result of the symbolic execution of all predecessing nodes of the path plus the node where the configuration is stored. The partial path constraints are stored at edges with non-trivial guard conditions and hold the resolution of the conjunction of all previous guard conditions on the path including the guard condition of the respective edge. In this way, when the selected path contains a part of an already executed path, it is not necessary to execute anew the path steps which they have in common.

```
inout: $p$ – path to be executed
       $mem$ – memory configuration
       $\Phi$ – path constraint
       $U \subseteq E_C$ – uncovered edges from CFG
procedure executeSymbolic($p$, $mem$, $\Phi$, $U$){

  $(u, g, v) = \kappa(p)$;
  if (!$u.covered$){
    executeExpression($u.expression$, $mem$);
    $u.covered = true$;
  }
  repeat{
    $c$ = resolveConstraint($g$, $mem$);
    if($c \neq true$){
      $\mu(u) = mem$;
      $\Phi = \Phi \wedge c$;
      $f((u, g, v)) = \Phi$;
    } else {
      $U = U \setminus \{\eta((u, g, v))\}$;
      executeExpression($v.expression$, $mem$);
      $v.covered = true$;

      $\kappa(p)++$;
      $(u, g, v) = \kappa(p)$;
    }
  }until($c \neq true$ || $p$ is executed);
}
```

Algorithm 6: Symbolic execution of a selected path.

Algorithm 6 shows the procedure *executeSymbolic*() which performs the described approach. Before we discuss how this algorithm operates, we introduce the following auxiliary functions:

$\kappa : E^* \rightarrow E$	Maps a path to the corresponding step, where the symbolic execution must be continued.
$\mu : N \rightarrow M^*$	Maps a STCT node to the memory model stored there.
$f : E \rightarrow Guard$	Maps a STCT edge to the partial path constraint stored there.

The algorithm *executeSymbolic*() receives as inputs the path p, whose execution must be performed, memory configuration *mem* and constraint Φ. The path p can already be partially executed. Thus, the memory configuration *mem* specifies the memory state after performing the symbolic execution of

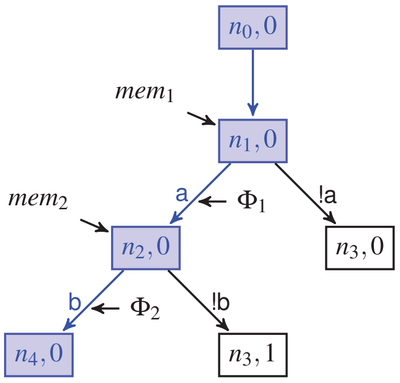


Figure 4.6: Example of information storage.

already covered path steps and constraint Φ represents the condition that must hold to reach the path step which must be executed next, accordingly to the path settings.

The execution starts at the first path step not yet executed. If the source node of the edge corresponding to this step is not yet covered, the procedure $executeExpression()$ (Algorithm 33) is invoked, which performs the symbolic execution of the expression associated with the considered node. (The principles of symbolic execution as well as algorithms for performing the symbolic execution of expressions and resolution of path constraints are discussed in Chapter 5). The results of the performed symbolic execution are stored in the memory specification *mem*, and the executed node is flagged as “covered”. Then the path is executed symbolically step by step. The guard condition of the edge corresponding to the current step is processed by the algorithm $resolveConstraint()$ (Algorithm 11 is discussed in Chapter 5). If the resolution result became a non-trivial value, the current memory configuration is stored at the source node of the considered edge, the resolution result is added to the path constraint Φ and is stored at the edge. Otherwise the CFG edge corresponding to the current path step is removed from the set of uncovered edges U, the target node of the edge is executed symbolically by invocation of the procedure $executeExpression()$ and is flagged as covered. Thereupon the current path step is updated and the procedure is repeated until the resolution result c becomes a non-trivial value or the path is executed completely.

To illustrate the discussed algorithm, we consider an example in Figure 4.6. Suppose, the path, whose symbolic execution must be performed is $< ((n_0,0),\varepsilon,(n_1,0)),\ ((n_1,0),a,(n_2,0)) >$. None symbolic execution was already performed, all edges and nodes are uncovered. Therefore the current path step is set to $((n_0,0),\varepsilon,(n_1,0))$. The node $(n_0,0)$ is not yet covered, so that its symbolic execution is performed first. Afterwards, the resolution of the guard of the edge $((n_0,0),\varepsilon,(n_1,0))$ is done and the returned result is $true$, since the guard condition is ε. Thus, the edge $((n_0,0),\varepsilon,(n_1,0))$ is removed from the set of uncovered edges, the expression corresponding to the node $(n_1,0)$ is executed symbolically and the path step is set to $((n_1,0),a,(n_2,0))$. The resolution of the guard condition a is performed, which returns a non-trivial result. Therefore, the current memory configuration mem_1 is stored at the node $(n_1,0)$ and the current partial path constraint Φ_1 at the edge $((n_1,0),a,(n_2,0))$. Since the last resolution result was not equal to $true$, the algorithm terminates.

4.3.5 Recalling Stored Information

Algorithm 7 shows the procedure *reset*(), which restores already collected information for the newly selected path. In order to find the gathered information, the procedure *reset*() traverses the path p in reversed order. Since the selected trace must contain uncovered edges, it is safe to assume that the last node of the trace was not already executed and to start the search at its predecessor. As soon as a stored memory configuration is detected, the procedure *reset*() stores it in an output parameter *mem* and sets the path step, where the symbolic execution must be continued. Since the memory configuration is stored at the branching point and the partial path constraint afterwards, the memory configuration will be found first. Therefore, as soon as a partial path constraint is found, the output parameter Φ is set and the procedure is stopped. If the path is traversed completely and no path constraint is found, the output parameter Φ remains $true$.

```
inout:  $p$ – path to be reset
output: $mem$ – recalled memory specification
        $\Phi$ – recalled path constraint
procedure reset($p$, $mem$, $\Phi$){

  $\kappa(p) = head(p)$;
  $\Phi = true$;
  interpretedNodeReached = $false$;
  foreach $(v, g, w) = last(p)$ downto $head(p)${
    if(!interpretedNodeReached && $\mu(v)$){
      $mem$ = $\mu(v)$;
      $\kappa(p)$ = $(v, g, w)$;
      interpretedNodeReached = $true$;
    }
    if($f((v, g, w))$){
      $\Phi$ = $f((v, g, w))$;
      break;
    }
}
```

Algorithm 7: Resetting the selected path.

To illustrate the discussed algorithm we consider an example in Figure 4.6. After the execution of the path $< ((n_0,0),\varepsilon,(n_1,0)),\ ((n_1,0),a,(n_2,0)),\ ((n_2,0),b,(n_4,0)) >$ (drawn blue) the memory configuration is stored at nodes $(n_1,0)$ (mem_1) and $(n_2,0)$ (mem_2). Path constraints Φ_1 and Φ_2 are stored at the edges $((n_1,0),a,(n_2,0))$ and $((n_2,0),b,(n_4,0))$ respectively. Suppose now that the path $< ((n_0,0),\varepsilon,(n_1, 0)),\ ((n_1,0),\ a,(n_2,0)),\ ((n_2,0),!b,(n_3,1)) >$ is selected. The first analyzed node $(n_2,0)$ holds the memory configuration mem_2. Consequently, the output parameter *mem* is set and the next path step to execute is set to $((n_2,0),!b,\ (n_3,1))$. The edge holds no information about path constraints. Thus, the procedure continues with the next path step $((n_1,0),a,(n_2,0))$. Since `interpretedNodeReached` is already set to $true$, only the corresponding edge is examined. It holds partial path constraint Φ_1, so that the algorithm terminates.

4.4 The Generation Algorithm

In this section we present the complete generation algorithm. The procedure *generateTestCases*() takes the CFG of a module to be explored as input (Algorithm 8). First, it performs initializations: the STCT and the set of uncovered edges U receive their initial values. Then it carries out the initial STCT expansion with the CFG start node as the leaf to be expanded and the initialization of the memory configuration.

The actual generation algorithm is fulfilled in a loop that terminates when the coverage goal was achieved or no path can be selected and no expansions are possible anymore. In general, the algorithm picks a path according to the edge priorities (the call of *selectContinuation*() with `path` equal to `NULL`) and tries to complete this path step by step. So the interior loop terminates when the path under consideration is completed (i.e the end node of the path is the CFG exit node) or the coverage goal was achieved or no path selection and no expansion are possible anymore. When the path selection was successful, the chosen path is reset first (Algorithm 7). In this operation all already cumulated information is recalled so that no repeated actions take place. Then, as long as the path is not executed completely and is feasible, the symbolic execution is performed step by step by invocation of the procedure *executeSymbolic*() (Algorithm 6). After each termination of *executeSymbolic*() the solver is called. If the solver declares the path condition as infeasible, the pruning algorithm is invoked and the generation algorithm falls back to the last feasible path, which then will be continued with an alternative branch. Otherwise, the interpreted edge is deleted from the set of uncovered edges U, the current path step is updated and the interpretation process continues until the path is executed completely or turns out to be infeasible. If the path could be executed completely, the last node of the path is expanded and the path is saved as `lastFeasiblePath` for a possible fallback. In the next loop iteration the continuation of this path is selected and the execution process is repeated until the path is complete. When the path is completed, we get a complete test case. Thus, the solution found by the solver is processed and a text representation of the test case in RT-Tester syntax is generated. We do not discuss the generation of a textual representation in this thesis since this generation consists in most cases only of a straightforward processing of the found solution. If the generation of a textual representation requires more complicated transformations, e.g. like in case of stub generation, the proceeding is illustrated by examples to the corresponding symbolic execution algorithms (see e.g. Section 5.12.2). Afterwards, the path is set to an initial value and the process is repeated.

If no path could be selected by the selection algorithm and the last expansion was successful, the uncovered part of a function under consideration contains only sequential code. In this case the function *expandAll*() is invoked to ensure that nothing is missed and *selectContinuation*() is called one more time. Now, if the expansion was not successful, the selection algorithm will also accept a path that contains no non-trivial guard conditions. Otherwise, if the expansion was successful, according to the expansion algorithm the STCT should now contain a new uncovered edge so that the selection algorithm will be able to select a new path.

However, if no expansion is possible anymore, no path can be selected, the STCT maximum size is not exceeded and the CFG still has uncovered edges, this means, that these edges are unreachable and the generation algorithm has detected unreachable code.

```
input:   $CFG = (N_C, E_C, V)$;
procedure generateTestCases($CFG$){

  $STCT = (N_0, E_0, L_0, \varphi_0, \sigma_0, \psi_0, \eta_0)$;
  $U = E_C$;
  expanded = expandLeaf($s$, $CFG$, $STCT$);
  initialize $mem$;

  while($U \neq \emptyset$){

    $p$ = NULL;
    lastFeasiblePath = NULL;
    feasible = $true$;

    while($U \neq \emptyset$ && $p$ is not complete){

      if(!feasible){
        $p$ = lastFeasiblePath;
        feasible = true;
      }
      $p$ = selectContinuation($p$, $STCT$, $U$, expanded)
      if($p$ != NULL) {
        reset($p$, $mem$, $\Phi$);
        while($p$ is not executed && feasible){
          executeSymbolic($p$, $mem$, $\Phi$, $U$);
          feasible = solve($\Phi$);
          $e = \kappa(p)$;
          if(!feasible){
            prune(e, $STCT$);
          } else {
            $U = U \setminus \{\eta(e)\}$;
            $\kappa(p)++$;
          }
        }
        if(feasible){
          $(u,g,w)$ = $last(p)$;
          if($U \neq \emptyset$ && $p$ is not complete && $w \in L$){
            expanded = expandLeaf($w$, $CFG$, $STCT$);
          }
          lastFeasiblePath = $p$;
        }
      } else if(expanded){
        expanded = expandAll($STCT$, $CFG$);
      } else {
        break;
      }
    }

    if($p$ == NULL && !expanded){
      break;
    }

    processSolution();
    $p$ = NULL;
  }
}
```

Algorithm 8: Complete generation algorithm.

5 Symbolic Execution

Symbolic execution was introduced in the 70s [65, 28, 27, 20, 90, 60] as an improved testing technique. Recently it got again a new interest and became a relevant field of research due to enhanced methods in constraint solving technology, the increase of computational power and algorithmic advances so that symbolic execution became more tractable [24]. It finds now application in different domains [84, 46, 29]:

- Test data generators use symbolic execution to build constraints on input data;
- Formal verification systems apply symbolic execution to derive logical predicates which are then proved by theorem provers;
- Development tools use symbolic execution to exercise or examine program transformations.

In this chapter we first give an introduction to symbolic execution (Section 5.1) and discuss its limitations (Section 5.2). Then we introduce the memory model developed in our research group and used in this dissertation (Section 5.3). We discuss the algorithm for symbolic execution (Section 5.4) that was used as a foundation to reason about complex data types like structures, unions, arrays and pointers (Sections 5.6-5.11). Besides, we demonstrate the principles of operation of the constraint generator (Section 5.5). And finally, we describe how the handling of function calls is solved (Section 5.12).

5.1 Introduction to Symbolic Execution

Symbolic execution is a well-known technique, that addresses the problem of automatic generation of test inputs. It was introduced in 1976 by James C. King [65]. Symbolic execution is similar to a normal execution process, the difference being that the values of program inputs are seen as symbolic variables, not concrete values. In this way, *"it offers an advantage that one symbolic execution may represent a large, usually infinite, class of normal executions* " [65].

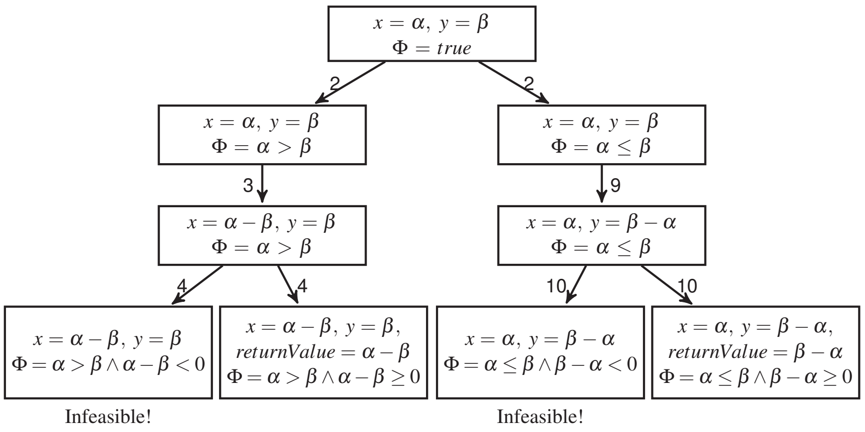
During symbolic execution when a variable is updated to a new value it is possible that this new value is an expression over symbolic variables. When the program flow comes to a branch where the branch condition depends on a symbolic variable, this condition can be evaluated both to *true* or to *false*, depending on the value of the symbolic variable. Through the symbolic execution of a path, it becomes a *path condition*, which is a conjunction of all branch conditions occurring on the respective path.

The *state* of symbolic execution includes the values of program variables, a program counter and a path condition. The *symbolic execution tree* represents paths investigated during the symbolic execution of a module. The tree nodes are associated to executed statements and represent program states. The nodes are connected by directed edges which represent program transitions.

Figure 5.1 shows an example of the symbolic execution tree for the function `diff()`, that calculates the absolute value of a difference of two integers. On the left side (Figure 5.1(a)) the code of the function

```
 1  int diff(int x, int y){
 2    if(x > y){
 3      x = x - y;
 4      if(x < 0){
 5        assert(false);
 6      }
 7      return x;
 8    } else {
 9      y = y - x;
10      if(y < 0){
11        assert(false);
12      }
13      return y;
14    }
15  }
```

(a) The C code of the diff() routine.


$x=\alpha,\ y=\beta$
$\Phi = true$
2
2
$x=\alpha,\ y=\beta$
$\Phi=\alpha>\beta$
$x=\alpha,\ y=\beta$
$\Phi=\alpha\le\beta$
3
9
$x=\alpha-\beta,\ y=\beta$
$\Phi=\alpha>\beta$
$x=\alpha,\ y=\beta-\alpha$
$\Phi=\alpha\le\beta$
4
4
10
10
$x=\alpha-\beta,\ y=\beta$
$\Phi=\alpha>\beta\wedge\alpha-\beta<0$
$x=\alpha-\beta,\ y=\beta,$
$returnValue=\alpha-\beta$
$\Phi=\alpha>\beta\wedge\alpha-\beta\ge0$
$x=\alpha,\ y=\beta-\alpha$
$\Phi=\alpha\le\beta\wedge\beta-\alpha<0$
$x=\alpha,\ y=\beta-\alpha,$
$returnValue=\beta-\alpha$
$\Phi=\alpha\le\beta\wedge\beta-\alpha\ge0$
Infeasible!
Infeasible!

(b) The symbolic execution tree of the diff() routine.

Figure 5.1: Symbolic execution example.

is listed, on the right side (Figure 5.1(b)) – the corresponding symbolic execution tree is shown. To simplify the understanding, we denote all variables with English letters (a, b, ...) and all symbolic values with Greek letters (α, β, ...). *returnValue* denotes the symbolic return value of the function. Initially, the path constraint Φ is set to *true* and the variables x and y have symbolic values α and β respectively. At each branching point an assumption about the inputs has to be made to distinguish between alternative paths. These assumptions are added to the path condition Φ. For example, the `if`-statement in line 2 can be evaluated to *true* as well as to *false*, so that both alternatives `then` and `else` are possible for this statement. Therefore the path condition Φ is updated accordingly: it becomes $\Phi = \alpha > \beta$ for the `then`-branch and $\Phi = \alpha \le \beta$ for the `else`-branch.

Path conditions are used to indicate infeasible paths. If a path condition cannot be evaluated to true, the symbolic execution of the corresponding path will not be continued, since there is no input data that could execute it. This means, that this symbolic state is unreachable. If no feasible path that executes a particular statement exists, the statement is unreachable. For example, in Figure 5.1(b) the path condition $\Phi = \alpha > \beta \wedge \alpha - \beta < 0$ at the leftmost node is infeasible and, since there is no other path that executes the statement in line 5, this statement is unreachable.

To reason about path conditions and hence about feasibility of paths, a constraint solver is used. When the solver determines a path condition as feasible, it calculates concrete values which can then be used as concrete inputs to explore the corresponding path.

King and his colleagues have implemented the presented principles of symbolic execution in EFFIGY, the interactive symbolic executor. This system was able to handle simple programs in a PL/I style programming language. King evaluated it on the basis of a couple of small examples and showed that his approach was promising. However, the limitations of theorem provers of that time restrained the possibilities of EFFIGY. For example, the executor was not able to handle variable storage-referencing, i.e. when the array read or write was dependent on a symbolic expression, and it could deal only with integer variables.

5.2 Limitations of Symbolic Execution

Despite recent development in the field of decision procedures and constraint solvers, rapid growth of computing power and new algorithmic developments, symbolic execution still suffers from a number of limitations. One of the limitations is, that the symbolic execution tree can become endless in case of loops. To avoid this, we limit the growth by a boundary – i.e. the depth of the search tree is restricted by a parameter. Another scalability problem in symbolic execution is path explosion. The number of paths in a program that must be explored grows exponentially in branching structure and even worse in the loops. To deal with this problem, we developed search and expansion strategies discussed in Chapter 4.

Further limitations of symbolic execution were listed in a case study [89]. Although the study analyzes concolic testing tools, the detected restrictions are caused by symbolic execution, since as soon as the concolic testing tool falls back to random testing due to limitation of symbolic execution, it loses all advantages by exploration of new paths. The authors distinguish the following limitations:

1. **Float/double data type variables**. This limitation is inherited from the underlying constraint solver. Symbolic execution is dependent on the availability of a solver to handle constraints with float or double variables and since many constraint solvers do not support this, the absence of this characteristic was selected as one of the limitations.

2. **Non-linear arithmetic operations** including multiplication, division and modular. Similar to the previous limitation, here the limitation is caused by the underlying constraint solver.

3. **Bitwise operations**. Similar to the previous two limitations in this case the restriction is conditioned rather by solver limitations and is not supported by some test generating tools.

4. **External function calls**. This limitation is caused by the fact that symbolic execution cannot handle code that is not available. This includes invocation of standard or user library functions or other components, whose code is not accessible.

5. **Pointers**. It is not possible to handle pointers in the same way as scalar variables by assigning them a symbolic value, since the program behavior is influenced not only by the concrete value of the pointer but also by the contents of the memory where the pointer points to. Besides, pointers introduce further problems through aliasing, when the value of a variable can be changed not only by a direct assignment but also by dereferencing of the pointer pointing to this variable.

6. **Symbolic offsets**. This limitation was pointed out already by King in his introduction of symbolic execution [65] and is still an issue [91]. Array access dependent on a symbolic expression may be ambiguous in many cases even if all information collected about inputs that occur in this symbolic expression is analyzed.

7. **Function pointers**. De-referencing of the function pointer pointing to an external function call consequently leads to the same troubles, but the problem is even more complicated when a function pointer is used as an input – in this case any function can be invoked by means of this function pointer.

In our opinion the following two limitations are missing in the study [89]:

```
 1  extern int calcTest(int c);
 2  float globalArr[10];
 3
 4  void example1(char *p, int c){
 5    if(p != NULL){
 6      if(c % 2 == 0){
 7        (*p) = calcTest(c);
 8      }
 9    }
10  }
11
12  void example2(int i){
13    if(globalArr[i] > 0){
14      ...
15    }
16  }
```

Figure 5.2: Limitations of symbolic execution.

8. **Recursive data structures** like trees, linked lists or queues. Like by simple pointers symbolic values cannot be used for such data structures, since feasibility of path constraints involving values of such data types depends not only on the concrete values of these variables but also on the shape of the corresponding data structure.

9. **Multithreading**. The introduction of parallelism leads to exponential growth of the number of interleavings of concurrent events. Executing such programs symbolically requires the support of partial order reduction, which needs derivation of interconnections between memory accesses in the program [92]. This involves alias analysis, pointer arithmetics etc.

We illustrate some of these limitations by the example from Figure 5.2. The statement in line 5 contains a pointer limitation, since parameter `p` is a pointer and is used in a comparison. Line 6 contains a non-linear arithmetic operation and line 7 an external function call. Line 13 contains two limitations: variable storage-referencing and usage of a float variable.

The authors [89] evaluated classified limitations on six industrial and open source systems for twelve test data generating tools, among them KLEE [22], EXE [23] and Pex [100]. The results showed that most dominant limitations, which prevented tested tools from generating high coverage, are pointers and external function calls.

The test generator developed in the scope of this dissertation successfully overcame limitations 1-6. The resolution of some limitations (1-3, 6) is due to the underlying constraint solver SONOLAR [82]. Other limitations (4-5) were solved by the techniques of symbolic execution designed in this dissertation. These techniques are demonstrated in the following sections of this chapter.

5.3 Memory Model

Due to aliasing in C/C++ the value of a variable can be changed not only by directly referencing its name but also by assignments to dereferenced pointers pointing to this specific variable. In the case of arrays different index expressions may reference the same array element. This makes it difficult to identify variable changes along program paths involving pointer and array expressions. To solve this problem, a memory model consisting of a history of *memory items* was introduced in [80, 70]:

Definition 5.1. *Memory item m is defined as the following structure:*

$$m \quad =_{def} \quad \begin{array}{|c|c|c|c|c|c|c|c|}\hline m.v_0 & m.v_1 & m.a & m.t & m.o & m.l & m.val & m.c \\ \hline\end{array}$$

where

- $m.v_0$ is the first computational step where m is valid,
- $m.v_1$ is the last computational step where m is valid, or ∞ for items valid beyond the actual computational step,
- $m.a$ is the symbolic base address,
- $m.t$ is the type of the specified $m.val$,
- $m.o$ is the start offset from base address in bits, where value is stored,
- $m.l$ is the offset from the base address to first bit following the stored value, so $m.l - m.o$ specifies the bit-length of the memory location represented by the item,
- $m.val$ is the value specification,
- $m.c$ is the validity constraint.

Each memory item is defined by its base address, offset, length, value expression and time interval (measured in computational steps) where it is valid. Computational steps are defined as memory modifications which are stored in corresponding memory items. Memory items are valid within an interval $[m.v_0,\ m.v_1]$ where the lower bound defines the first and the upper bound the last computation in which the memory item was a part of the configuration. Furthermore, the stored values are not resolved to concrete or abstract valuations but are specified symbolically.

We illustrate the concept of the memory item by an example. Suppose, the symbolic execution is performed on a 32-bit architecture and has arrived in computational step n by statement `c1 = c - 1`, where `c` and `c1` are variables of type `char`. The following memory item m will be created:

$$m \quad =_{def} \quad \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&c1 & char & 0 & 8 & c_n - 1 & true \\ \hline\end{array}$$

To represent memory blocks representing the same or related values (for example array values), the following concept is introduced:

Definition 5.2. *Family of memory items* is defined as the following structure [80]:

$$m_{p_0,\dots p_k} \quad =_{def} \quad \begin{array}{|c|c|c|c|c|c|c|c|}\hline v_0 & v_1 & a & t & o(p_0,\,\dots,\,p_k) & l(p_0,\,\dots,\,p_k) & val(p_0,\,\dots,\,p_k) & c(p_0,\,\dots,\,p_k) \\ \hline\end{array}$$

so that

$$\begin{array}{lll} m_{p_0,\ldots p_k} & =_{def} \{m' \mid & m'.v_0 = v_0 \wedge m'.v_1 = v_1 \wedge m'.a = a \wedge m'.t = t \wedge \\ & & (\exists p'_0, \ldots, p'_k : m'.o = o\,[p'_0/p_0, \ldots, p'_k/p_k] \wedge \\ & & m'.l = l\,[p'_0/p_0, \ldots, p'_k/p_k] \wedge \\ & & m'.val = val\,[p'_0/p_0, \ldots, p'_k/p_k] \wedge \\ & & m'.c = c\,[p'_0/p_0, \ldots, p'_k/p_k]\,)\,\} \end{array}$$

For example, the integer array `a[10]`, which values are all set to zero on a 32-bit architecture, would be represented as

$$m_i \quad =_{def} \quad \begin{array}{|c|c|c|c|c|c|c|c|} \hline n & \infty & \&a[0] & int & 32\cdot i & 32\cdot i+32 & 0 & 0 \le i \wedge i < 10 \\ \hline \end{array}$$

where $i \in \{0, \ldots, 9\}$. In this way, all ten elements of this array are represented with a single memory item, each element aligned according to its index on an index-dependent offset from the base address $\&a[0]$. Consequently, if we are interested in the second array element (i is equal to 1) we can derive, that the element is located at the offset 32 up to 64 and its value is 0.

Definition 5.3. *A symbolic execution space S_S of a module U* with corresponding $CFG = (N_C, E_C, V)$ is defined as [80]:

$$\begin{array}{rll} S_S & =_{def} & N_C \times \mathbb{N}_0 \times M \\ M & =_{def} & dataSegment \times heapSegment \times stackSegment \\ dataSegment & =_{def} & M-Item^* \\ heapSegment & =_{def} & M-Item^* \\ stackSegment & =_{def} & M-Item^* \\ M-Item & =_{def} & \mathbb{N}_0 \times \{\mathbb{N}_0 \cup \infty\} \times BaseAddress \times Types \times Offset \times OffsetPlusLength \times \\ & & Value \times Constraint \\ BaseAddress & =_{def} & String \\ Offset & =_{def} & OffsetPlusLength =_{def} Value =_{def} Constraint =_{def} Expr(Sym \times \mathbb{N}_0) \\ Sym & =_{def} & \text{symbols of } U \cup P \\ P & =_{def} & \text{parameters for families of memory items} \end{array}$$

Every symbolic state is a triple $(node, n, mem)$, where *node* is a node in the corresponding CFG, characterizing the current progress of the symbolic execution, n is a computational step counter and *mem* the current memory state. The memory is divided into data segment, heap segment and stack. Memory items are stored in one of these partitions corresponding to the allocation of the associated variable.

Offsets, values and constraints are specified symbolically by means of expressions over variables of module U and auxiliary parameters from the specification of families of memory items (as we illustrated in the last example).

As we will show, this approach – despite aliasing – allows not only to find out the actual memory area where a new value is written to but also enables us to handle pointer comparisons and pointer arithmetics.

5.4 Basic Symbolic Execution Algorithm

Symbolic execution is performed after the following rule [80]:

$$\frac{n_i \xrightarrow{g}_{CFG} n_j}{(n_i, n, mem) \longrightarrow_G (n_j, n+1, mem')},$$

where

- n_i, n_j are CFG nodes,
- $\xrightarrow{g}_{CFG}$ denotes an edge in the CFG with the guard condition g,
- (n_i, n, mem) and (n_j, n, mem) represent consecutive symbolic states,
- $\longrightarrow_G$ denotes symbolic execution step (G stands here for "GIMPLE operational semantics").

According to this rule, the symbolic execution step can be performed on the symbolic level whenever a corresponding edge exists in the CFG of the module U. However, this transition can be infeasible, i.e. that there exists no assignment of the inputs that evaluates the corresponding path constraint to $true$. In this case this step is not executed.

A symbolic execution step is performed in three phases [80]:

1. For each base address and offset which are modified by the current statement a new memory item m' is created, that should be added to the memory configuration.
2. For each new memory item m' a check is performed, if m' invalidates any of the existent memory items, i.e. all memory items m whose corresponding memory area overlaps with the memory area of the new memory item m' are found. This can happen only if the base addresses of m and m' are equal and offset-characterized areas of m and m' have a non-empty intersection.
3. For each invalidated memory item new memory items m''_i are created. These memory items characterize the memory area of m which could still stay unaffected by m'. In case when memory areas of m and m' are equal, no memory items m''_i will be created but m' replaces m completely.

We illustrate the described procedure by the following example. Suppose, there is an integer array `a[10]` whose values were set to zero in the n_0-th computation step. On a 32-bit architecture it is represented as follows:

$$m = \begin{array}{|c|c|c|c|c|c|c|c|}\hline n_0 & \infty & \&a[0] & int & 32\cdot i & 32\cdot i+32 & 0 & 0 \le i \wedge i < 10 \\ \hline\end{array}$$

Suppose further, that there were no modifications of any element of this array until in the n-th computational step the fourth element of this array was set to k: `a[3] = k`. To perform this computational step according to the described approach we:

1. Create a new memory item

$$m' = \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&a[0] & int & 96 & 128 & 0 & true \\ \hline\end{array}$$

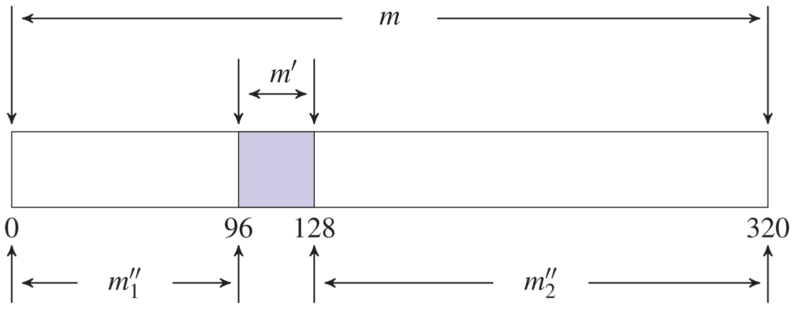


Figure 5.3: Memory item invalidation example.

2. Check the existent memory items for invalidation. We find the memory item m, since m and m' refer to the same base address and their memory areas overlap. (Figure 5.3 shows the memory areas corresponding to the participating memory items). We invalidate the memory item m:

$m =$	n_0	n	$\&a[0]$	int	$32 \cdot i$	$32 \cdot i + 32$	0	$0 \leq i \wedge i < 10$

3. For the unaffected memory area of the memory item m we create two new memory items (see Figure 5.3)

$m''_1 =$	n	∞	$\&a[0]$	int	$32 \cdot i$	$32 \cdot i + 32$	0	$0 \leq i \wedge i < 3$
$m''_2 =$	n	∞	$\&a[0]$	int	$32 \cdot i$	$32 \cdot i + 32$	0	$4 \leq i \wedge i < 10$

Now we will analyze the effect of the symbolic execution on the state space S_S formally. In the next section we define rules specifying stack variable definition and variable assignment.

5.4.1 Memory Model Initialization and Variable Assignment

First, we introduce auxiliary functions that we need for further definitions and algorithms [80]:

$\beta : Selectors \rightarrow BaseAddress$	Maps a selector to the corresponding base address.
$\tau : Selectors \rightarrow Symbols$	Maps a selector to the type of the corresponding variable.
$\omega : Selectors \rightarrow Expression$	Maps a selector to the corresponding symbolic offset expression.
`bitsizeof`: $Selectors \rightarrow \mathbb{N}$	Maps a selector to the length of selected memory in bits.
$\sigma : BaseAddress \times M \rightarrow M - Item^*$	Maps a base address to the stack, heap or global data accordingly to the current memory configuration.
$\upsilon : Expression \rightarrow \mathbb{N}$	Returns a version of the given expression.

Definitions of all auxiliary functions used in this chapter can be found in Section 5.14.

Definition 5.4. The effect of the *stack variable definition* (`typex x;`) on the state space S_s is specified as follows [80]:

$$mem' =_{def} (mem.dataSegment,\ mem.heapSegment,\ mem.stackSegment \cap \{m\}).$$

Where m is the memory item originated by the stack variable definition, which is specified in the following way:

$$m \quad =_{def} \quad \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&x & \texttt{typex} & 0 & \texttt{bitsizeof(typex)} & \texttt{Undef} & true \\ \hline\end{array}$$

Here `Undef` reflects that the value of the corresponding variable is still undefined. A stack variable definition affects only the stack segment.

To be able to handle contents of pointer inputs, as well as of pointer members in structures, the generator simulates the memory corresponding to these pointers: it creates auxiliary variables related to the input pointers and constructs associated memory items as it is shown in the following definition.

Definition 5.5. The effect of the *parameter definition* (`funcName(typex x, ..)`) on the state space S_s is specified as follows:

$$mem' =_{def} (mem.dataSegment,\ mem.heapSegment,\ mem.stackSegment \cap S).$$

Where S is the set of memory items originated by the parameter definition, which is specified in the following way:

1. `typex` is not a pointer or structure:

$$m \quad =_{def} \quad \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&x & \texttt{typex} & 0 & \texttt{bitsizeof(typex)} & x_0 & true \\ \hline\end{array}$$
$$S = \{m\}$$

   Here x_0 refers to the initial value of the parameter and reflects that this is an input.

2. `typex` is a pointer to a type `typey`, where `typey` is not a structure or union:

$$m_1 =_{def} \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&x & \texttt{typex} & 0 & \texttt{bitsizeof(typex)} & \&x@P_n & true \\ \hline\end{array}$$

$$m_2 =_{def} \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&x@P[0] & \texttt{typey[s]} & 0 & \texttt{bitsizeof(typey)}\cdot s & x@P_0 & true \\ \hline\end{array}$$
$$S = \{m_1, m_2\}$$

   Here $x@P$ is an auxiliary array of size s, which simulates the memory where the pointer x points to. The version 0 of the value of the memory item m_2 indicates that this is an input.

3. `typex` is a structure:

$$m_{1\ldots i} \quad =_{def} \begin{cases} \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&x & \texttt{typex} & l_{i-1} & l_i & x_0 & true \\ \hline\end{array} & \text{if i-th member is not a pointer} \\ \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&x & \texttt{typex} & l_{i-1} & l_i & x.m_i@P_n & true \\ \hline\end{array} & \text{otherwise} \end{cases}$$

   For all members m_i, so that $\texttt{type}_i$ is a pointer to a type $\texttt{type}'_i$:

$$S_1 \quad = \quad \bigcup_i \{\text{memory items created by definition of parameter } \texttt{type}'_i\ \texttt{x.m}_i\texttt{@P}\}$$

$$S = \{m_1, \ldots, m_i\} \cup S_1$$

Here `x.m`$_i$`@P` is an auxiliary variable, which simulates the variable where the member `x.m`$_i$ points to.

4. `typex` is a pointer to a type `typey`, where `typey` is a structure:

$$m_0 \quad =_{def} \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&x & \texttt{typex} & 0 & \texttt{bitsizeof(typex)} & \&x@P_n & true \\ \hline\end{array}$$

$$S_1 \quad = \quad \{\text{memory items created by definition of parameter } \texttt{typey x@P}\}$$

$$S = \{m_0\} \cup S_1$$

5. `typex` is a pointer to a type `typey`, where `typey` is a union:

$$m_1 =_{def} \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&x & \texttt{typex} & 0 & \texttt{bitsizeof(typex)} & \&x@P_n & true \\ \hline\end{array}$$

$$m_2 =_{def} \begin{array}{|c|c|c|c|c|c|c|c|}\hline n & \infty & \&x@P & \texttt{typey} & 0 & \texttt{bitsizeof(typey)} & x@P_0 & true \\ \hline\end{array}$$

$$S = \{m_1, m_2\}$$

Since an assignment to an array member of a union type is not supported (see Section 5.10.1), here $x@P$ is an auxiliary variable, which simulates the memory where the pointer x points to. The version 0 of the value of the memory item m_2 indicates that this is an input.

A parameter definition affects only the stack segment.

Recursive data types like lists etc. are not supported. However, as the main application field of the developed test generator is embedded systems, this limitation is not crucial, since according to the guidelines for embedded systems MISRA-C [10]: *"Dynamic heap memory allocation shall not be used."* An example with pointer parameters of structure type is discussed in Section 5.8.1.

Definition 5.6. The effect of the *global variable definition* (`typex x;`) on the state space S_s is specified as follows:

$$mem' =_{def} (mem.dataSegment \cap S,\ mem.heapSegment,\ mem.stackSegment).$$

Where S is the set of memory items originated by the global variable definition, which is specified in the same way as was defined for the parameter definition (rules 1-5). A global variable definition affects only the data segment.

Definition 5.7. The effect of the *assignment to a stack or global variable* (`sel = expr;`) on the state space S_s is specified by procedure call [80]:

$$updateByAssignment(sel,\ expr,\ n,\ mem); \quad mem' = mem;$$

where

- *sel* is an arbitrary selector that can be an identifier of an atomic variable, structure access, array element or mixed structure/array identifier, for example of the form `v.field1[i]`,
- *expr* is an expression that should be assigned to the identifier *sel*,
- *n* is the current computation step,
- *mem* is the current memory specification.

Assignment to a stack or global variable affects the stack segment or the global data segment.

```
inout: $mem$ − current memory specification
input: $sel$ − selector of the identifier
       $exp$ − expression that should be assigned to the identifier $sel$
       $n$ − current computation step

procedure updateByAssignment($sel$, $exp$, $n$, $mem$){

   // create new memory item
   $m'.v_0 = n$;
   $m'.v_1 = \infty$;
   $m'.a = \beta(sel)$;
   $m'.t = \tau(sel)$;
   $m'.o = \omega(sel)_n$;
   $m'.l = \omega(sel)_n +$ bitsizeof($sel$);
   $m'.val = exp$;
   if($\upsilon(exp)$ == $\infty$){
      $\upsilon(m'.val) = n$;
   }
   $m'.c = true$;

   // insert new memory item into the memory configuration
   insert($m', n, mem$);
}
```

Algorithm 9: Effect of the assignment on the memory specification.

Algorithm 9 shows the procedure *updateByAssignment*(), which specifies how a new memory item is created. Particularly, the validity period is defined, the base address and the type of the variable corresponding to the selector *sel* are calculated and set, offset and length of the memory location are calculated symbolically, the value is set according to the right-hand side of the expression. Symbolic expressions defining offset, length and validity constraint get a version corresponding to the current computational step. Symbolic expression defining the value of the memory item receives an exceptional handling: in case when the version of the assigned expression is not set, the version of the value is set corresponding to the current computational step like for all other symbolic expressions. Otherwise, the value keeps the version of the assigned expression. This exceptional handling is involved in the processing of undefined function calls as is discussed in Section 5.12.2.

As was discussed in Chapter 4 we perform an incremental approach of a path execution by which each non-trivial guard condition is checked for feasibility before the execution of the successive nodes is done.

This ensures the existence of the created memory item. The path constraint containing the conjunction of the already evaluated guard conditions is stored apart from the memory items and therefore the validity constraint of the created memory item can be set to *true*.

Furthermore, *updateByAssignment*() invokes the procedure *insert*() (Algorithm 10) that performs the insertion of the new memory item into the current memory configuration.

```
inout: $mem$ – current memory specification
input: $m'$ – memory item to be inserted into the memory specification
       $n$ – current computation step

procedure insert($m'$, $n$, $mem$){

  // find out corresponding segment
  $S = \sigma(m'.a, mem)$;
  $U = \emptyset$;

  foreach $m = last(S)$ downto $head(S)${

    if($m.v_1$ == $\infty$ && $m'.a$ == $m.a$){

      // check if memory items overlap
      if($\neg(m'.l \leq m.o \vee m.l \leq m'.o)$){

        // invalidate found memory item
        $m.v_1 = n$;

        // remains of the old item on the left side
        $c''_1 = m.c \wedge m'.c \wedge m.o < m'.o \wedge m'.o < m.l$;
        $m''_1 = (n, \infty, m.a, m.t, m.o, m'.o, m.val, c''_1)$;
        if($c''_1$ is feasible){
          $U = U \cup \{m''_1\}$;
        }

        // remains of the old item on the right side
        $c''_2 = m.c \wedge m'.c \wedge m.o < m'.l \wedge m'.l < m.l$;
        $m''_2 = (n, \infty, m.a, m.t, m'.l, m.l, m.val, c''_2)$;
        if($c''_2$ is feasible){
          $U = U \cup \{m''_2\}$;
        }
      }
    }
  }
  $S = S \cup U \cup \{m'\}$;
}
```

Algorithm 10: Insertion of the new memory item into the memory specification.

To insert a new memory item, the procedure *insert*() performs a loop over all matching memory items in the current memory configuration (i.e. all valid memory items with the same base address as a new memory item m'). In this loop for all memory items overlapping with the newly created item the invalidation of the found memory items and the creation of new memory items for unaffected memory areas is performed. Thereby two new memory items are created:

1. Memory item m''_1. It captures the remains of the address range of the old memory item m on the left from the address range of the new memory item m' (see also example in Figure 5.3).

2. Memory item m''_2. It captures the remains of the address range of the old memory item m on the right from the address range of the new memory item m' (see also example in Figure 5.3).

In both situations we check, if the created memory item m''_i is feasible (i.e. its validity constraint is feasible) and only in this case it is inserted into the memory configuration.

We illustrate the presented algorithm by the following example:

```
#define INDEX1 3
#define VALUE1 32
#define INDEX2 0
#define VALUE2 33
int example1(unsigned int x){
  unsigned int a[10];
  a[INDEX1] = VALUE2;
  a[INDEX2] = VALUE1;
  if(a[x] == VALUE1)
    ...
}
```

This function has the following GIMPLE representation:

```
1 int example(unsigned int x){
2   unsigned int a[10];
3   unsigned int D_1710;
4   a[3] = 33;
5   a[0] = 32;
6   D_1710 = a[x];
7   if(D_1710 == 32)
8     ...
9 }
```

Now we process this GIMPLE code line by line. First stack variable definition for all local variables and input parameters are done:

$$\begin{aligned}
m_1 &= (\texttt{1, }\infty\texttt{, \&x, 0, 32, unsigned int, } x_0\texttt{, true})\\
m_2 &= (\texttt{2, }\infty\texttt{, \&a[0], 0, 320, unsigned int, Undef, true})\\
m_3 &= (\texttt{3, }\infty\texttt{, \&D\_1710, 0, 32, unsigned int, Undef, true})
\end{aligned}$$

The memory item m_2 represents the family of memory items corresponding to the array `a[10]`. Since m_2 corresponds to a contiguous memory area and to simplify the understanding, we omit here and in all following examples the parameterized representation.

Next, the statement in line 4 is executed. The new memory item m_4 is created, valid from the current computational step. The old memory item m_2 is invalidated and two additional memory items are produced: m_5 and m_6. The memory item m_5 corresponds to the memory item m''_1 from the insertion algorithm (Algorithm 10) and represents the remains of the address region of the old memory item m_2 to the right side of the address region of the new memory item m_4. The memory item m_6 corresponds to the memory item m''_2 from the insertion algorithm and represents the remains of the address region of the old memory item m_2 to the left side of the address region of the new memory item m_4. The memory configuration is now as follows:

$$\begin{aligned}
m_1 = &\quad (1, \infty, \texttt{\& x, 0, 32, unsigned int, } x_0\texttt{, true})\\
m_2 = &\quad (\texttt{2, 3, \&a[0], 0, 320, unsigned int, Undef, true})\\
m_3 = &\quad (3, \infty, \texttt{ \&D\_1710, 0, 32, unsigned int, Undef, true})\\
m_4 = &\quad (4, \infty, \texttt{\&a[0], 96, 128, unsigned int, 33, true})\\
m_5 = &\quad (4, \infty, \texttt{\&a[0], 0, 96, unsigned int, Undef, true})\\
m_6 = &\quad (4, \infty, \texttt{\&a[0], 128, 320, unsigned int, Undef, true})
\end{aligned}$$

The statement in line 5 is proceeded in the same manner as the previous one. Though the memory configuration contains three valid items with base address `&a[0]` (m_4, m_5 and m_6), only one (m_5) corresponds to the memory area that overlaps with the newly created item m_7. Furthermore, no memory item, corresponding to the m''_1, exists, since the new memory item m_7 overlaps with the leftmost address region of the old item m_5. For this reason, only two new memory items were inserted into the memory configuration:

$$\begin{aligned}
m_1 = &\quad (1, \infty, \texttt{ \&x, 0, 32, unsigned int, } x_0\texttt{, true})\\
m_2 = &\quad (\texttt{2, 3, \&a[0], 0, 320, unsigned int, Undef, true})\\
m_3 = &\quad (3, \infty, \texttt{ \&D\_1710, 0, 32, unsigned int, Undef, true})\\
m_4 = &\quad (4, \infty, \texttt{\&a[0], 96, 128, unsigned int, 33, true})\\
m_5 = &\quad (\texttt{4, 4, \&a[0], 0, 96, unsigned int, Undef, true})\\
m_6 = &\quad (4, \infty, \texttt{\&a[0], 128, 320, unsigned int, Undef, true})\\
m_7 = &\quad (5, \infty, \texttt{\&a[0], 0, 32, unsigned int, 32, true})\\
m_8 = &\quad (5, \infty, \texttt{\&a[0], 32, 96, unsigned int, Undef, true})
\end{aligned}$$

Next, the statement in line 6 is executed. A new memory item for variable `D_1710` is created and the old one is invalidated. No further new items are created, since memory item m_9 replaces memory item m_3 completely. Afterwards, the memory configuration is as follows:

$$\begin{aligned}
m_1 = &\quad (1, \infty, \texttt{ \&x, 0, 32, unsigned int, } x_0\texttt{, true})\\
m_2 = &\quad (\texttt{2, 3, \&a[0], 0, 320, unsigned int, Undef, true})\\
m_3 = &\quad (\texttt{3, 5, \&D\_1710, 0, 32, unsigned int, Undef, true})\\
m_4 = &\quad (4, \infty, \texttt{ \&a[0], 96, 128, unsigned int, 33, true})\\
m_5 = &\quad (\texttt{4, 4, \&a[0], 0, 96, unsigned int, Undef, true})\\
m_6 = &\quad (4, \infty, \texttt{ \&a[0], 128, 320, unsigned int, Undef, true})\\
m_7 = &\quad (5, \infty, \texttt{ \&a[0], 0, 32, unsigned int, 32, true})\\
m_8 = &\quad (5, \infty, \texttt{ \&a[0], 32, 96, unsigned int, Undef, true})\\
m_9 = &\quad (6, \infty, \texttt{ \&D\_1710, 0, 32, unsigned int, a}_6\texttt{[x}_6\texttt{], true})
\end{aligned}$$

The `if` statement in line 7 specifies a guard condition that must be fulfilled in order to cover the corresponding branch. The constraint generator described in the following section constructs the corresponding path constraint.

5.5 Constraint Generator

After the statements are executed symbolically as is shown in the previous section, all necessary information for constructing the input assignment satisfying the guard condition that must be fulfilled to cover the corresponding branch is available in the memory configuration. The constraint generator is responsible for the resolution of the values of the involved memory items for the purpose of constructing a path constraint free of pointer and other values not supported by the solver [80]. To demonstrate this process, we continue with the example from the previous section and compose the path constraint to cover the branch corresponding to the *true* evaluation of the guard condition from the `if` statement in line 7.

The constraint generator operates as follows:

1. Initialize the path constraint Φ corresponding to the guard condition:
   $\Phi = (\texttt{D\_1710}_6 == 32)$.
   The variable `D_1710` becomes version 6 accordingly to the current computational step.

2. Resolve $\texttt{D\_1710}_6$. The constraint generator finds the memory item responsible for `D_1710` which is valid in computational step 6. In our example this is m_9. According to the value of m_9 $\texttt{D\_1710}_6$ is resolved to $\texttt{a}_6\texttt{[x}_6\texttt{]}$. Since $\texttt{a}_6\texttt{[x}_6\texttt{]}$ is not a constant it has to be resolved further.

3. Resolve $\texttt{a}_6\texttt{[x}_6\texttt{]}$. It matches with memory items m_8, m_7, m_6 and m_4 (memory items m_5 and m_2 are already invalid in computational step 6). For each matching memory item the constraint generator defines a conjunctive clause which resolves $\texttt{a}_6\texttt{[x}_6\texttt{]}$ and requires that the index $\texttt{x}_6$ is within the bounds of the address region of the corresponding memory item. This leads to the following constraint:
   $$\begin{aligned}&(\texttt{D\_1710}_6 == \texttt{Undef} \wedge 32\cdot x_6 < 96 \wedge 32 < 32\cdot x_6 + 32) \vee\\ &(\texttt{D\_1710}_6 == 32 \wedge 32\cdot x_6 < 32 \wedge 0 < 32\cdot x_6 + 32) \vee\\ &(\texttt{D\_1710}_6 == \texttt{Undef} \wedge 32\cdot x_6 < 320 \wedge 128 < 32\cdot x_6 + 32) \vee\\ &(\texttt{D\_1710}_6 == 33 \wedge 32\cdot x_6 < 128 \wedge 96 < 32\cdot x_6 + 32)\end{aligned}$$
   Now there is only one unresolved variable: x_6.

4. Resolve x_6. The constraint generator finds the matching memory items, in this case there is only one: m_3. The value of m_3 is x_0 and cannot be resolved further, since this is an input. This means, the constraint generator has finished and the resulting path constraint is as follows:
   $$\begin{aligned}\Phi = {}&(\texttt{D\_1710}_6 == 32) \wedge\\ &((\texttt{D\_1710}_6 == \texttt{Undef} \wedge 32\cdot x_6 < 96 \wedge 32 < 32\cdot x_6 + 32) \vee\\ &(\texttt{D\_1710}_6 == 32 \wedge 32\cdot x_6 < 32 \wedge 0 < 32\cdot x_6 + 32) \vee\\ &(\texttt{D\_1710}_6 == \texttt{Undef} \wedge 32\cdot x_6 < 320 \wedge 128 < 32\cdot x_6 + 32) \vee\\ &(\texttt{D\_1710}_6 == 33 \wedge 32\cdot x_6 < 128 \wedge 96 < 32\cdot x_6 + 32)) \wedge\\ &(x_6 == x_0)\end{aligned}$$

Notice that the final version of Φ contains only operations supported by the solver. Array accesses were replaced by offset conditions and atomic variables corresponding to their values.

After we have sketched the principle of operation of the constraint generator, we will analyze it in more detail. Algorithm 11 shows the upper-level procedure of the constraint generation process. Function *resolveConstraint*() takes a guard condition which must be resolved and the current memory specification as inputs. It initializes the path constraint with the guard condition and, as long as the path

```
input:  $g$ – guard condition
        $mem$ – current memory specification
output: $\Phi$ – resolved path constraint
function resolveConstraint($g$, $mem$){

  $\Phi = g$;
  while ($\Phi$ contains unresolved variable $var$){

    // find out corresponding segment
    $S = \sigma(\beta(var), mem)$;
    foreach $m = last(S)$ downto $head(S)${
      if ($m.v_0 \leq \upsilon(var) \wedge \upsilon(var) \leq m.v_1 \wedge m.a$ == $\beta(var)$){
        $overlap = (m.o < \omega(var) +$ bitsizeof$(sel)) \wedge (m.l > \omega(var))$;
        $c = m.c \wedge overlap$;
        if($c$ is feasible){
          resolveExp($var$, $m.val$, $c$, $mem$);
          $c_1 = c_1 \vee c$;
        }
      }
    }

    $\Phi = \Phi \wedge c_1$;
  }
  resolvePointerVars($\Phi$, $mem$);
  return $\Phi$;
}
```

Algorithm 11: Constraint resolution.

constraint contains unresolved variables, performs the following: it iterates over all memory items which correspond to the unresolved variable *var*. If the validity period of the item matches with the version of the variable *var*, feasibility constraint *c* is constructed. This feasibility constraint requires that the validity constraint of the memory item is feasible and that the address range of the memory item overlaps with the offsets specified by the variable expression. If the constructed constraint is feasible, the procedure *resolveExp*() (Algorithm 12) is invoked with variable *var* as the variable to be resolved and value of the found memory item *m.val* as its value. *resolveExp*() classifies the value *val* and invokes the corresponding subroutine (e.g. *val* can be a dereferenced pointer or structure access etc). If the detected value is a constant or of an atomic type and does not need further resolution to enable its handling by the solver, the condition requiring that *var* is equal to the value *val* is added to the feasibility constraint Φ. More complicated cases are handled by the subroutines for the resolution of specific forms of expressions. The principles of operation of these subroutines are discussed in the following sections.

After all variables participating in the passed guard condition, are resolved, the procedure *resolvePointerVars()* (Algorithm 17) is invoked, which brings the pointers in the constructed path constraint Φ in a form which can be handled by the solver. The detailed discussion about how we approach the pointer handling is given in Section 5.7.

```
input:  var – variable to be resolved
        val – detected value of the variable
        mem – current memory specification
output: res – resolution result
procedure resolveExp(var, val, res, mem){

  if(val is a constant){
    $res = res \wedge (var == val)$;
  } else if(val is a dereferenced pointer){
    resolveDerefPtr(var, val, res, mem);
  } else if(val is an array expression){
    resolveArrayExp(var, val, res, mem);
  } else if(val is a struct or union expression){
    offsetStart = $\omega(val)$;
    offsetEnd = $\omega(val)$ + bitsizeof(val);
    if(val is a struct pointer expression){
      resolveStructPtrExp(var, val, offsetStart, offsetEnd, res, mem);
    } else if(val is a union pointer expression){
      resolveUnionPtrExp(var, val, offsetStart, offsetEnd, res, mem);
    } else if(val is a struct expression){
      resolveStructExp(var, val, offsetStart, offsetEnd, res, mem);
    } else{
      resolveUnionExp(var, val, offsetStart, offsetEnd, res, mem);
    }
  } else if(val is an address operation){
    resolveAddrExp(var, val, res);
  } else {
    $res = res \wedge (var == val)$;
  }
}
```

Algorithm 12: Expression resolution.

5.6 Handling of Dereferenced Pointers

5.6.1 Assignment

Definition 5.8. The effect of the *assignment to a dereferenced pointer* (e.g. `*p = exp;` or `p->f = exp;`) on the state space S_s is specified by the procedure call [80]:

$$updateByAssignmentToDerefPtr(p,\, exp,\, n,\, mem); \quad mem' = mem;$$

where

- p is a pointer identifier,
- exp is an expression that should be assigned to the dereferenced pointer p,
- n is the current computational step,
- mem is the current memory specification.

The assignment to a dereferenced pointer may affect the stack, data or heap segment dependent on the value of pointer `p`.

Algorithm 13 shows the procedure *updateByAssignmentToDerefPtr()*, which specifies how the new memory items are created by assignment to a dereferenced pointer. First, the procedure finds all possible targets where `p` can point to. For this purpose the procedure iterates over all valid memory items referring to the pointer base address and resolves the values of these items using auxiliary function $resolvePtrVal()$ (see Algorithm 14). In case when the value of the found memory item is a structure access, the auxiliary function $resolveStructPtrVal()$ (Algorithm 20) is invoked instead of the $resolvePtrVal()$. We present this algorithm in Section 5.8 which discusses the handling of structure accesses in detail.

A pointer can point to one or more locations, depending on its value and validity constraint. *resolvePtrVal()* traverses over all these possible situations and resolves them to expressions of the form:

$$baseAddress + offset.$$

Such expressions specify which base address and offset the memory item has and, in that way, which value is modified by the assignment to the dereferenced pointer `p`. The list of these memory item specifications is returned to the *updateByAssignmentToDerefPtr()*, which creates a new memory item m' with a new value *exp* for each of them. For pointer expressions like `(*p)[i]` the offset resulting from the resolution must be corrected, so that additional offset start and offset end are calculated and added to the specified offset. For simple pointer expressions like `(*p)` the calculated additional offset is equal to zero. The insertion of the created item (and, correspondingly, invalidation of memory items conflicting with the new one) is made as specified by the procedure $insert()$ (see Algorithm 10).

We illustrate this approach by the following example:

```
input: $p$ – pointer identifier
       $exp$ – expression that should be assigned to the dereferenced pointer $p$
       $n$ – current computational step
inout: $mem$ – current memory specification

procedure updateByAssignmentToDerefPtr($p$, $exp$, $n$, $mem$){

  // find out corresponding segment
  $S = \sigma(\beta(p), mem)$;

  foreach $m = last(S)$ downto $head(S)${
    if($m.v_1$ == $\infty$ && $m.a$ == $\beta(p)$ ){

      if($m.val$ is a pointer struct access){
        $pl$ = resolveStructPtrVal($m.val, mem$);
      } else {
        $pl$ = resolvePtrVal($m.val, mem$);
      }
      foreach $m''$ in $pl${
          // create new memory item
          $m'.v_0 = n$;
          $m'.v_1 = \infty$;
          $m'.a = m''.a$;
          $m'.t = m''.t$;
          $m'.o = m''.o + \omega(p)$;
          $m'.l = m''.o + \omega(p)$ + bitsizeof($p$);
          $m'.val = exp$;
          if($\upsilon(exp)$ == $\infty$){
            $\upsilon(m'.val) = n$;
          }
          $m'.c = m.c \wedge m''.c$;

          // insert new memory item into the memory configuration
          insert($m', n, mem$);
      }
    }
  }
}
```

Algorithm 13: Effect of the assignment to a dereferenced pointer on the memory specification.

```
input: $exp$ − expression that should be resolved
       $mem$ − current memory specification
output: $el$ − set of memory items with potential target addresses and offsets

function resolvePtrVal($exp$, $mem$){

  // create new memory item
  $m = (\ldots, exp, true)$;
  $el = \{m\}$;

  while ($\exists m'$ in $el$: $m'.val$ is unresolved){

    if(calculated base address offset of $m'.val$){
      $m'.o$ = offset;
      $m'.a$ = baseAddress;
      // is resolved to baseAddress + offset, done
      continue;
    }

    x = unresolved identifier;
    $S = \sigma(\beta(x), mem)$;
    foreach $m'' = last(S)$ downto $head(S)${
      if($m''.a$ == $\beta$(x) && $m''.v_0 \leq \upsilon(x) \leq m''.v_1$){
        if($m''.val$ is a pointer struct access){
          $el = el \cup$ resolveStructPtrVal($m''.val, mem$);
        } else {
          $val_1 = m'.val$;
          in $val_1$: replace all occurrences of x by $m''.val$;
          $el = el \cup \{(\ldots, val_1, m'c \wedge m''.c)\}$;
        }
      }
    }
    // $m'$ was replaced by $m''$, erase $m'$
    $el = el \setminus \{m'\}$
  }
  return $el$;
}
```

Algorithm 14: Resolution of the pointer value to all potential base addresses and offsets.

C code	GIMPLE representation
`1  int check() {`	`1  int check() {`
`2`	`2`
`3      int z[10];`	`3      int z[10];`
`4      int *ip, *ip1;`	`4      int *ip;`
`5      int ret;`	`5      int *ip1;`
`6`	`6      int ret;`
`7      ip = z;`	`7      int D_1712;`
`8      *ip = 77;`	`8`
`9      ...`	`9      ip = &z[0];`
`10 }`	`10     *ip = 77;`
	`11     ...`
	`12 }`

Note, that symbolic execution is performed on the GIMPLE code. First the memory configuration is initialized:

$m_1 =$ `(1, ∞, &z[0], 0, 320, int, Undef, true)`
$m_2 =$ `(2, ∞, &ip, 0, 32, int*, Undef, true)`
$m_3 =$ `(3 ∞, &ip1, 0, 32, int*, Undef, true)`
$m_4 =$ `(4, ∞, &ret, 0, 32, int, Undef, true)`
$m_5 =$ `(5, ∞, &D_1712, 0, 32, int, Undef, true)`

Execution of line 9 (`ip = &z[0];`) introduces a new memory item m_6:

$m_6 =$ `(6, ∞, &ip, 0, 32, int*, &z`$_6$`[0], true)`

and invalidates the memory item m_2:

$m_2 =$ `(2, 5, &ip, 0, 32, int*, Undef, true)`

Thus, as the symbolic execution process reaches the line 10 (`*ip = 77;`), the memory has the following configuration:

$m_1 =$ `(1, ∞, &z[0], 0, 320, int, Undef, true)`
$m_2 =$ `(2, 5, &ip, 0, 32, int*, Undef, true)`
$m_3 =$ `(3 ∞, &ip1, 0, 32, int*, Undef, true)`
$m_4 =$ `(4, ∞, &ret, 0, 32, int, Undef, true)`
$m_5 =$ `(5, ∞, &D_1712, 0, 32, int, Undef, true)`
$m_6 =$ `(6, ∞, &ip, 0, 32, int*, &z`$_6$`[0], true)`

As the procedure *updateByAssignmentToDerefPtr()* is called for expression (`*ip = 77`) the resolution process is started for the memory item m_6. The function *resolvePtrVal*() resolves its value `&z`$_6$`[0]` to the base address `&z[0]` and offset `0`. A new memory item m_7 is created:

$m_7 =$ (7, ∞, &z[0], 0, 32, int, 77, true)

Its insertion causes the invalidation of the memory item m_1 and the creation of an additional memory item m_8 corresponding to the address range of the old memory item m_1 unaffected by the item m_7:

$m_1 =$ (1, 6, &z[0], 0, 320, int, Undef, true)
$m_8 =$ (7, ∞, &z[0], 32, 320, int, Undef, true)

Thus, after the symbolic execution of the expression (`*ip = 77`) the memory is configured as follows:

$m_1 =$ (1, 6, &z[0], 0, 320, int, Undef, true)
$m_2 =$ (2, 5, &ip, 0, 32, int*, Undef, true)
$m_3 =$ (3 ∞, &ip1, 0, 32, int*, Undef, true)
$m_4 =$ (4, ∞, &ret, 0, 32, int, Undef, true)
$m_5 =$ (5, ∞, &D_1712, 0, 32, int, Undef, true)
$m_6 =$ (6, ∞, &ip, 0, 32, int*, &z_6[0], true)
$m_7 =$ (7, ∞, &z[0], 0, 32, int, 77, true)
$m_8 =$ (7, ∞, &z[0], 32, 320, int, Undef, true)

In this way, despite aliasing, the value of the expression was written by the symbolic execution to the memory where it was intended to write by the program.

5.6.2 Resolution

The resolution of a dereferenced pointer is performed by the function call

$$resolveDerefPtr(var,\ p,\ c,\ mem)$$

Where

- *var* is a versioned variable identifier. It indicates the variable, that has a dereferenced pointer as a value;
- *p* is a versioned pointer identifier. It indicates the pointer, which is dereferenced;
- *c* is a constraint that holds the result of the resolution process;
- *mem* is the current memory specification.

Algorithm 15 shows the procedure *resolveDerefPtr()*. First, the algorithm resolves the value of the given dereferenced pointer *p* with help of the auxiliary procedure *resolveDerefPtrExp()* whose principle of operation we discuss later in this section. The algorithm returns a set *S* where all possible values of the dereferenced pointer associated with the pointer whose dereferenced value was resolved (here *p*) are

```
input: $var$ – variable identifier which has a dereferenced pointer as value
       $p$ – pointer identifier
       $mem$ – current memory specification
inout: $c$ – feasibility constraint
procedure resolveDerefPtr($var$, $p$, $c$, $mem$){

  $S = \emptyset$; $isInput = false$; $validFrom = 0$;
  resolveDerefPtrExp($var$, $p$, $S$, $mem$, $isInput$, $validFrom$);

  if($isInput$){

    // sort accordingly to validation period
    $S$.sort();
    // find out when the value of dereferenced pointer was overwritten
    ($resolution$, $input$, $c_2$, $validFrom$) = $head(S)$;

    // go over all inputs of pointer type and find alternative values
    foreach $input$ of pointer type{
      resolveDerefPtrExp($var$, $input$, $S$, $mem$, $true$, $validFrom$);
    }

    // sort accordingly to validation period
    $S$.sort();
  }

  foreach ($resolution$, $input$, $c_2$, $validFrom$) = $last(S)$ downto $head(S)${
    if($p \neq input$){
      $c_1 = (p == input)$;
    } else {
      $c_1 = true$;
    }
    $res = res \vee (resolution \wedge c_2 \wedge c_1 \wedge neg)$;

    if($isInput$){
      $neg = neg \wedge (!c_1 \vee !c_2)$;
    }
  }
  $c = c \wedge res$;
}
```

Algorithm 15: Resolution of a dereferenced pointer.

stored, with the feasibility constraint of this resolution and the computational step where the variable was overwritten. Furthermore, *resolveDerefPtrExp()* analyzes whether the pointer refers to an input and stores the result of this analysis in the input/output parameter *isInput*. If the generator detects that the pointer is an input and its value is not overwritten and still points to the variable simulated by the generator (see Section 5.4.1), we have to consider the situation when several input pointers point to one and the same variable like in the following example:

```
int i1 = 0;
int *p1 = &i1, *p2 = &i1;
ptr_test(p1, p2);
...
```

Thus, other possible alternative values of the dereferenced pointer are analyzed: all pointer inputs are resolved by invocation of the auxiliary procedure *resolveDerefPtrExp()*. This procedure adds all possible alternative resolution results to the set S for further analysis. The alternative value is detected if the input pointer also still points to the simulated variable and the content of this variable was overwritten later than the content of the variable where the original pointer p was pointing to. To show why it is important whether the variable where another pointer points to was overwritten before or after the dereferenced pointer under consideration, we consider the following example:

```
ptr_test(int *p1, int *p2){
  *p1 = 1;
  *p2 = 0;
  ...
}
```

Suppose, the pointer under consideration is `p1`. Then, if `p1` and `p2` point to one and the same variable, the value of `*p1` after the execution of listed code is `0` (which is the value that was written to `*p2`). However, if the pointer under consideration is `p2`, it is not significant which value was written to `*p1`, since if `p1` and `p2` point to the same variable, it would already be overwritten. And if the pointers point to different variables, `*p1` cannot affect `*p2` by any means.

After all possible values have been collected, they are sorted according to the computational step where the values of dereferenced pointers were overwritten. Subsequently, these values, beginning with the most recent one, are traversed and the constraint holding all possible resolutions of the passed dereferenced pointer p is built. This constraint requires that if the result is equal to the alternative resolution, then the feasibility constraint of this resolution must hold and the pointer under consideration must be equal to the corresponding input pointer and the more recent resolutions are infeasible, which means that either the pointers are not equal or the feasibility constraints of the resolutions are infeasible. So, if we suppose that there are n alternative resolutions, which are sorted in such a way that resolution r_n together with the corresponding feasibility constraint c_n and the corresponding input pointer p_n refer to the most recent alternative value and r_1 together with the corresponding feasibility constraint c_1 and the corresponding input pointer p_1 refers to the oldest one, the resulting constraint *res* after traversing all these resolutions has the following form:

$$\begin{array}{ll}
(r_n \wedge c_n \wedge (p == p_n)) & \vee \\
(r_{n-1} \wedge c_{n-1} \wedge (p == p_{n-1}) \wedge ((p \neq p_n) \vee !c_n)) & \vee \\
\ldots & \\
(r_1 \wedge c_1 \wedge (p == p_1) \wedge ((p \neq p_n) \vee !c_n)) \wedge \ldots \wedge ((p \neq p_2) \vee !c_2)) & \vee
\end{array}$$

The constraint requiring the equality of pointers is built only if the pointer corresponding to the current resolution is not the original pointer. The negation constraint *neg* is built only if the original pointer refers to an input. Otherwise it is redundant since all validity conditions of the resolution are already summarized in the corresponding feasibility constraints of possible resolutions.

After all resolutions are traversed the constructed constraint is conjuncted with the resulting constraint c.

Algorithm 16 shows the procedure *resolveDerefPtrExp()*. First, the algorithm detects all possible targets where `p` can point to. For this purpose it iterates over all valid memory items corresponding to the pointer base address and invokes the auxiliary function *resolvePtrVal()* (see Algorithm 14) or $resolveStructPtrVal()$ (see Algorithm 20) for the values of the found memory items. $resolvePtrVal()$ and $resolveStructPtrVal()$ resolve each value expression to the list of possible target memory items specified by the base address and offset. After the possible targets are identified, the algorithm finds out the values stored in these targets. Therefore, for each of the specified base addresses *resolveDerefPtrExp()* traverses over all matching memory items. Now we differentiate if we perform the resolution for an input pointer or not. If this is a resolution for an input pointer, the memory item is further considered only if its validity period corresponds to the validity period of the value found for the input pointer (we are interested only in more recent entries) and if the memory item refers to a simulated input. If this is not a resolution for an input pointer all found memory items are considered.

The further analysis is performed as follows: if the validity period of the found item corresponds to the version of the variable identifier *var* constraint c_1 is built, which requires, that:

1. The validity constraint of the memory item m corresponding to the pointer is valid.

2. The validity constraint of the memory item m' corresponding to the target of the pointer is valid.

3. The validity constraint of the memory item m'' corresponding to the target specification is valid.

4. The address range of m' overlaps with the address range specified by the pointer `p`.

Afterwards, if the constructed constraint is feasible, the variable identifier *var* is passed for further resolution of the value of the memory item m' to the procedure $resolveExp()$ discussed in Section 5.5. The constraint produced by this resolution is stored together with the corresponding pointer, feasibility constraint and the validity period in the resulting set of possible outcomes of the resolution process of the dereferenced pointer `p`. If it is detected that the memory item m' refers to a simulated input variable, the input/output parameter *isInput* indicating whether the dereferenced pointer still points to an input is set to $true$.

To illustrate the described approach, we first demonstrate a simple example not involving input pointers. Therefore, we extend the example from the previous section:

```
input: $var$ – variable identifier which has a dereferenced pointer as value
       $p$ – pointer identifier
       $mem$ – current memory specification
       $validFrom$ – indicates the validity period of matching memory items
inout: $R$ – set of found resolutions
       $isInput$ – indicates whether the resolution is performed for a pointer input
procedure resolveDerefPtrExp($var$, $p$, $R$, $mem$, $isInput$, $validFrom$){

  // find out corresponding segment
  $S = \sigma(\beta(p), mem)$;

  $offsetStart = \omega(p)$;
  $offsetEnd = \omega(p) +$ size($basetype(p)$);

  foreach $m = last(S)$ downto $head(S)${
    if($m.v_0 \leq \upsilon(p) \wedge \upsilon(p) \leq m.v_1 \wedge m.a$ == $\beta(p)$ ){

      if($m.val$ is a pointer struct access){
        $pl$ = resolveStructPtrVal($m.val, mem$);
      } else {
        $pl$ = resolvePtrVal($m.val, mem$);
      }

      foreach $m''$ in $pl${

        // for each memory item specification in the list
        // find all items overlapping with it
        $S_1 = \sigma(m''.a, mem)$;

        foreach $m' = last(S_1)$ downto $head(S_1)${
          if(($isInput \wedge validFrom < m'.v_0 \wedge m'$ refers to a simulated input) $\vee$ !$isInput$)){
            if($m'.v_0 \leq \upsilon(var) \wedge \upsilon(var) \leq m'.v_1 \wedge m'.a$ == $m''.a$){
              $overlap = (m'.o < m''.o + offsetEnd) \wedge (m'.l > m''.o + offsetStart)$;
              $c_1 = m.c \wedge m''.c \wedge m'.c \wedge overlap$;

              if($c_1$ is feasible){
                resolveExp($var, m'.val, c_2, mem$);
                $R$.push(($c_2$, $p$, $c_1$, $m'.v_0$));
                if($m'$ refers to a simulated input){
                  $isInput$ = $true$;
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Algorithm 16: Auxiliary procedure for the resolution of a dereferenced pointer.

C code	GIMPLE representation

```
1  int check() {
2
3      int z[10];
4      int *ip, *ip1;
5      int ret;
6
7      ip = z;
8      *ip = 77;
9      ip1 = ip;
10     if(*ip1 == 0){
11       ret = 0;
12     } else {
13       ret = 1;
14     }
15     ...
16 }
```
	```
	1  int check() {
	2
	3      int z[10];
	4      int *ip;
	5      int *ip1;
	6      int ret;
	7      int D_1712;
	8
	9      ip = &z[0];
	10     *ip = 77;
	11     ip1 = ip;
	12     D_1712 = *ip1;
	13     if(D_1712 == 0){
	14       ret = 0;
	15     } else {
	16       ret = 1;
	17     }
	18     ...
	19 }
	```

In the previous section we have shown that after the symbolic execution of the first 10 lines the memory is configured as follows:

$m_1 =$ (1, 6, &z[0], 0, 320, int, Undef, true)
$m_2 =$ (2, 5, &ip, 0, 32, int*, Undef, true)
$m_3 =$ (3 ∞, &ip1, 0, 32, int*, Undef, true)
$m_4 =$ (4, ∞, &ret, 0, 32, int, Undef, true)
$m_5 =$ (5, ∞, &D_1712, 0, 32, int, Undef, true)
$m_6 =$ (6, ∞, &ip, 0, 32, int*, &z_6[0], true)
$m_7 =$ (7, ∞, &z[0], 0, 32, int, 77, true)
$m_8 =$ (7, ∞, &z[0], 32, 320, int, Undef, true)

The next assignments `ip1 = ip` and `D_1712 = *ip1` overwrite the values of the memory items m_3 and m_5 so that the memory configuration afterwards is as follows:

$m_1 =$ (1, 6, &z[0], 0, 320, int, Undef, true)
$m_2 =$ (2, 5, &ip, 0, 32, int*, Undef, true)
$m_3 =$ (3, 7, &ip1, 0, 32, int*, Undef, true)
$m_4 =$ (4, ∞, &ret, 0, 32, int, Undef, true)
$m_5 =$ (5, 8, &D_1712, 0, 32, int, Undef, true)
$m_6 =$ (6, ∞, &ip, 0, 32, int*, &z_6[0], true)
$m_7 =$ (7, ∞, &z[0], 0, 32, int, 77, true)
$m_8 =$ (7, ∞, &z[0], 32, 320, int, Undef, true)
$m_9 =$ (8, ∞, &ip1, 0, 32, int*, ip_8, true)
$m_{10} =$ (9, ∞, &D_1712, 0, 32, int, *$ip1_9$, true)

Now we process as defined by the function *resolveConstraint*() (Algorithm 11):

1. Initialize the path constraint accordingly to the guard condition:

   $$\Phi = (\texttt{D\_1712}_9 == 0).$$

2. Resolve `D_1712` $_9$: find the memory item responsible for `D_1712`$_9$, this is m_{10}. Resolve `D_1712`$_9$ according to the value of the item found:

   $$\texttt{D\_1712}_9 \texttt{==*ip1}_9.$$

   Now the algorithm *resolveDerefPtr()* is invoked with `D_1712`$_9$ as *var*, `ip1`$_9$ as *p*, Φ as *c* and our memory configuration as *mem*. This algorithm invokes the auxiliary procedure *resolveDerefPtr-Exp()* with `D_1712`$_9$ as *var*, `ip1`$_9$ as *p*, empty set *S* as *R*, our memory configuration as *mem* and *isInput* and *validFrom* set correspondingly to *false* and `0`.

   First the offset start and offset end for `ip1`$_9$ are calculated, these are 0 and 32. Then the possible targets of `ip1`$_9$ are detected. For this purpose the value of the memory item m_9 (`ip`$_8$) is analyzed. As this is not a structure access, it is passed to the auxiliary function *resolvePtrVal*(). It produces the following specification: the base address is `&z[0]` and the offset is 0 (corresponding to the value of the memory item m_6). The internal loop iterates over the memory items corresponding to base address `&z[0]`. These are m_1, m_7 and m_8. Since *isInput* is set to *false*, all these memory items are considered, but, since the validity period of m_1 does not match the validity period of the variable `D_1712` $_9$ and the address range of the item m_8 ([32, 320)) does not overlap with the calculated offset ([0, 32)), only memory item m_7 matches. The value of m_7 is 77 and its feasibility constraint is *true*, so that the following tuple is stored in the resolution set *R*:

   $$(\texttt{D\_1712}_9 \texttt{==77}, \texttt{ip1}_9, \mathit{true}, 7)$$

   Here `D_1712`$_9$ `==77` is the resolution of `*ip1`$_9$ detected by *resolveDerefPtrExp()*, `ip1`$_9$ refers to a pointer, whose dereferenced value was resolved, *true* is the validity constraint of the found resolution and 7 refers to the computational step where the value of the dereferenced pointer was overwritten.

   Since m_7 does not refer to a simulated input, the value of *isInput* remains *false*. For this reason, back in the procedure *resolveDerefPtrExp()* no further resolutions are required. Thus, the returned set *S* is iterated and the constraint *res* is built:

   $$\texttt{D\_1712}_9 \texttt{==77}.$$

   This resolution is added to the resulting constraint Φ:

   $$\Phi = (\texttt{D\_1712}_9 == 0 \wedge \texttt{D\_1712}_9 \texttt{==77}).$$

3. No unresolved symbols exist anymore and the resolution process stops. Φ is infeasible and, since no other path goes to line 14, this line is consequently unreachable.

Now we consider an example involving input pointers.

C code	GIMPLE representation
`1 int ptr_test(int *p1, int *p2)`	`1 int ptr_test(int *p1, int *p2)`
`2 {`	`2 {`
`3   *p1 = 0;`	`3   int D_1724;`
`4   *p2 = 1;`	`4`
`5   if(*p1 == 1){`	`5   *p1 = 0;`
`6     return 1;`	`6   *p2 = 1;`
`7   } else {`	`7   D_1724 = *p1;`
`8     return 0;`	`8   if(D_1724 == 1){`
`9   }`	`9     ...`
`10 }`	`10   } else {`
	`11     ...`
	`12   }`
	`13 }`

For a better understanding of the procedure, we represent it as follows: we list the example code line by line and after each line we specify the memory items which were created by the symbolic execution of this line. The symbolic execution steps are numbered according to the line numbers of the GIMPLE representation listed above.

To set the example as clearly as possible, we do not initialize simulated auxiliary variables where the parameter `p1` and `p2` point to as arrays (as it was defined in Section 5.4.1), since (1) the algorithm for the handling of input arrays is not discussed yet – we do this in Section 5.11 – and (2) in this example this characteristic does not affect the correctness of the resolution, since here we can assume that the pointers do not point to some arrays. Thus, after the initialization of parameters the memory configuration is as follows:

```
1 int ptr_test(int *p1, int *p2)
```

$$
\begin{aligned}
m_1 &= (\texttt{1, }\infty\texttt{, \&p1, 0, 32, int*, \&p1@P}_1\texttt{, true})\\
m_2 &= (\texttt{2, }\infty\texttt{, \&p1@P, 0, 32, int, p1@P}_0\texttt{, true})\\
m_3 &= (\texttt{3, }\infty\texttt{, \&p2, 0, 32, int*, \&p2@P}_3\texttt{, true})\\
m_4 &= (\texttt{4, }\infty\texttt{, \&p2@P, 0, 32, int, p2@P}_0\texttt{, true})
\end{aligned}
$$

Subsequently, the stack initialization is done:

```
3 int D_1724;
```

$$
m_5 = (\texttt{5, }\infty\texttt{, \&D\_1724, 0, 32, int, Undef, true})
$$

After the initialization is completed, we proceed with the symbolic execution line by line:

```
5 *p1 = 0;
```

The assignment to the dereferenced pointer is proceeded as specified by the procedure *update-ByAssignmentToDerefPtr()* (see Algorithm 13), and a new memory item is created:

$m_6 =$ (6, ∞, &p1@P, 0, 32, int, 0, true)

The insertion of the memory item m_6 into the memory specification invalidates the memory item m_2, so that now m_2 is configured as follows:

$m_2 =$ (2, 5, &p1@P, 0, 32, int, p1@P_0, true)

6 `*p2 = 1;`

This assignment is proceeded similarly to the previous one:

$m_7 =$ (7, ∞, &p2@P, 0, 32, int, 1, true)

The insertion of the memory item m_7 into the memory specification invalidates memory item m_4, so that now m_4 is configured as follows:

$m_4 =$ (4, 6, &p2@P, 0, 32, int, p2@P_0, true)

7 `D_1724 = *p1;`

$m_8 =$ (8, ∞, &D_1724, 0, 32, int, *p1$_8$, true)

The insertion of the memory item m_8 into the memory specification invalidates memory item m_5, so that now m_5 is configured as follows:

$m_5 =$ (5, 7, &D_1724, 0, 32, int, Undef, true)

The next line of the example consists of an `if` statement `if(D_1724 == 1)`. This means, that the evaluation of the guard condition `(D_1724 == 1)` is necessary. Before we start with the resolution algorithm, we summarize the current memory specification:

$m_1 =$ (1, ∞, &p1, 0, 32, int*, &p1@P_1, true)
$m_2 =$ (2, 5, &p1@P, 0, 32, int, p1@P_0, true)
$m_3 =$ (3, ∞, &p2, 0, 32, int*, &p2@P_3, true)
$m_4 =$ (4, 6, &p2@P, 0, 32, int, p2@P_0, true)
$m_5 =$ (5, 7, &D_1724, 0, 32, int, Undef, true)
$m_6 =$ (6, ∞, &p1@P, 0, 32, int, 0, true)
$m_7 =$ (7, ∞, &p2@P, 0, 32, int, 1, true)
$m_8 =$ (8, ∞, &D_1724, 0, 32, int, *p1$_8$, true)

Now we continue as defined by the function *resolveConstraint*() (Algorithm 11):

1. Initialize the path constraint according to the guard condition:

   $$\Phi = (\texttt{D\_1724}_8 == 1).$$

2. Resolve `D_1724` $_8$: find the memory item responsible for `D_1724`$_8$, this is m_8. Resolve `D_1724`$_8$ according to the value of the item found:

   $$\texttt{D\_1724}_8 == {\star}\texttt{p1}_8.$$

   Now the algorithm *resolveDerefPtr()* is invoked with `D_1724`$_8$ as *var*, `p1`$_8$ as *p*, Φ as *c* and our memory configuration as *mem*. This algorithm invokes the auxiliary procedure *resolveDerefPtr-Exp()* with `D_1724`$_8$ as *var*, `p1`$_8$ as *p*, empty set *S* as *R*, our memory configuration as *mem* and *isInput* and *validFrom* set correspondingly to *false* and `0`.

   First, the possible targets of `p1`$_8$ are detected. The value of the memory item m_1 (`&p1@P`$_1$) is analyzed. This is not a structure access, so it is passed to the auxiliary function *resolvePtrVal*(). It produces the following specification: the base address is `&p1@P` and the offset is 0. The internal loop iterates over the memory items m_2 and m_6. Since *isInput* is set to *false*, all these memory items are considered, but, since the validity period of m_2 does not match the validity period of the variable `D_1724` $_8$, only memory item m_6 matches. Thus, the following tuple is stored in the resolution set *R*:

   $$(\texttt{D\_1724}_8 == \texttt{0}, \texttt{p1}_8, \textit{true}, 6)$$

   Here `D_1724`$_8$ `==0` is the resolution of ${\star}$`p1`$_8$ detected by *resolveDerefPtrExp()*, `p1`$_8$ refers to a pointer, whose dereferenced value was resolved, *true* is the validity constraint of the found resolution and 6 refers to the computational step where the value of the dereferenced pointer was overwritten.

   Since m_6 does refer to a simulated input `p1@P`, the value of *isInput* is set to *true*. For this reason further resolution process is required in the procedure *resolveDerefPtr()* . First, *validFrom* is set to 6 according to the computational step stored in the resolution set. Only values which were written in subsequent computations are relevant. Next, the algorithm iterates over all input pointers (here this is pointer parameter `p2`) and invokes *resolveDerefPtrExp()* with `D_1724`$_8$ as *var*, detected input pointer `p2`$_8$ as *p*, set *S* as *R*, our memory configuration as *mem* and *isInput* and *validFrom* set correspondingly to *true* and `6`.

   Similar to the resolution of `p1`$_8$, possible targets of `p2`$_8$ are detected. The value of the memory item m_3 (`&p2@P`$_3$) is analyzed. This is not a structure access, so it is passed to the auxiliary function *resolvePtrVal*(). It produces the following specification: the base address is `&p2@P` and the offset is 0. The internal loop iterates over the memory items m_4 and m_7. Since *isInput* is set to *true*, only m_7 is considered further as although the memory item m_4 does refer to a simulated input `p2@P`, its validity period does not conform to the value of the parameter *validFrom*. The resolution of m_7 results in the following tuple, which is stored in the resolution set *R*:

   $$(\texttt{D\_1724}_8 == \texttt{1}, \texttt{p2}_8, \textit{true}, 7)$$

   Back in the procedure *resolveDerefPtr()* the set *S* of all possible resolutions is sorted according to the validation period and after that it is as follows:

$$(\texttt{D\_1724}_8 == 0, \texttt{p1}_8, \mathit{true}, 6)$$
$$(\texttt{D\_1724}_8 == 1, \texttt{p2}_8, \mathit{true}, 7)$$

Afterwards, the set *S* is iterated beginning with the most recent entry and the constraint *res* is built step by step. First, we process the tuple $(\texttt{D\_1724}_8 == 1, \texttt{p2}_8, \mathit{true}, 7)$:

$$\mathit{res} = (\texttt{D\_1724}_8 == 1 \wedge \texttt{p1}_0 == \texttt{p2}_0).$$

After this iteration the negation constraint *neg* has the following form:

$$\mathit{neg} = (\texttt{p1}_0 \neq \texttt{p2}_0).$$

Now we process the next tuple $(\texttt{D\_1724}_8 == 0, \texttt{p1}_8, \mathit{true}, 6)$:

$$\mathit{res} = (\texttt{D\_1724}_8 == 1 \wedge \texttt{p1}_0 == \texttt{p2}_0) \vee (\texttt{D\_1724}_8 == 0 \wedge \texttt{p1}_0 \neq \texttt{p2}_0).$$

This resolution is added to the resulting constraint Φ:

$$\Phi = (\texttt{D\_1724}_8 == 1) \wedge$$
$$((\texttt{D\_1724}_8 == 1 \wedge \texttt{p1}_0 == \texttt{p2}_0) \vee (\texttt{D\_1724}_8 == 0 \wedge \texttt{p1}_0 \neq \texttt{p2}_0)).$$

3. No unresolved symbols exist anymore and the resolution process stops. The resulting path constraint Φ is feasible in case when input pointers `p1` and `p2` point to one and the same variable and the designed algorithm makes it possible to detect this.

The generated test driver as well as the other outputs produced by the test generator for this example are presented in Appendix 3.

The algorithm discussed in this section makes it possible to support the case when multiple pointer inputs point to the same variable but only for pointers of atomic types. However, the symbolic execution algorithm can be extended to support the case where the equality of pointers pointing to unions or structures is supported. For that purpose the algorithms developed for the union and structure pointer resolution (Sections 5.8.1 and 5.10.3) must be extended in a similar manner as the procedure *resolveDerefPtr()* discussed in this section.

5.7 Handling of Pointers

5.7.1 Background

A pointer is *"a variable that contains the address of a variable"* [63]. In C, the following pointer operations are valid [63]:

- Assignment of the pointers of the same type. If `p` and `p1` are pointers of the same type, then `p = p1` copies the contents of `p1` into `p`. Consequently, after this assignment `p` points to the same variable `p1` points to.

- Addition or subtraction of a pointer and an integer. If a pointer `p` points to some element of an array, then `p + 1` points to the next element and `p + i` points `i` elements after `p`. Correspondingly, `p - i` points `i` elements before. This is true despite of the type or size of the array elements and `i` is scaled corresponding to the size of the variable the pointer points to.

- Comparison of two pointers, but only if they point to the members of the same array. The behavior is undefined for arithmetic comparisons with pointers that do not point to the members of the same array. For example, `p > p1` is true, if `p` points to the element of the array behind the element the `p1` does.
- Subtraction of two pointers, but only if they point to the members of the same array. For example, `p - p1 + 1` is the number of elements from `p` to `p1` inclusive, if `p` and `p1` point to the elements of the same array and `p > p1`.
- Assignment or comparison with zero. Pointers and integers are not interchangeable, zero is the only exception.

Pointer arithmetic is consistent, it does not matter whether we deal with characters, integers or floats. The size of the variable the pointer points to is automatically considered by all pointer manipulations [63].

As we stated already, the assignment of pointers is a simple copy of contents of one variable to another and does not differ from an assignment of, for example, an integer variable. Thus, it is handled as specified by the procedure *updateByAssignment*(). The challenging part when handling pointers is their resolution. Since the solver does not support handling of constraints with pointers, the input pointer variables must be resolved to some types the solver is capable of dealing with. And though pointers are addresses, and addresses can be represented as integers, the simple exchange of pointers by integer variables in a constraint is not sufficient. The value of the pointer must be set reasonable – it must satisfy requirements of the pointer operations and it must be possible to meaningfully associate a variable to the calculated address. The method that we have developed for the pointer resolution, is described in the next section.

5.7.2 Resolution

We handle all memory areas pointed to by pointers as arrays with configurable size. By abstracting pointers to integers we achieve that constraints over pointers can be solved by a solver capable of integer arithmetics.

First, we introduce the definition and lemma that we need for the algorithm:

Definition 5.9. Each pointer p is defined by a pair of unsigned integers

$$p = (A, x),$$

where

- A corresponds to a base address of the memory area where the pointer p points into,
- x is its offset ($p = A + x$).

Lemma 5.1. *Expression*

$$(p_1 \ \omega \ p_2)$$

where

- *ω is a comparison operator,*
- *p_1 and p_2 are pointers of the form $p_i = (A_i, x_i),\ i = 1,2$*

is equivalent to the following constraint:

$$A_1 == A_2 \ \&\&\ x_1\ \omega\ x_2 \ \&\&\ 0 \le x_1 < dim(p_1) \ \&\&\ 0 \le x_2 < dim(p_2) \tag{5.1}$$

where

- *$A_1 == A_2$ ensures, that p_1 and p_2 point to the members of the same memory portion,*
- *$x_1\ \omega\ x_2$ reflects the pointer expression and*
- *$0 \le x_i < dim(p_i),\ i = 1,2,$ guarantees, that the pointer stays within the array bounds.*

Proof. First we prove, that if $(p_1\ \omega\ p_2)$, then (5.1) holds. As was stated in the previous section, the behavior of $(p_1\ \omega\ p_2)$ is defined only if p_1 and p_2 point to the members of the same array, hence $A_1 == A_2$ and $0 \le x_i < dim(p_i),\ i = 1,2$ hold. The result of comparison or subtraction of two pointers is dependent on the order of the array elements where the pointers point to. Offset x_i represents exactly the position of the element in the array where the pointer points to, hence $(x_1\ \omega\ x_2)$ holds and consequently holds (5.1).

Now we prove, that if (5.1) holds, then $(p_1\ \omega\ p_2)$ holds too. Since $A_1 == A_2$ and $0 \le x_i < dim(p_i),\ i = 1,2$ hold, it is ensured that corresponding pointers p_1 and p_2 point to the members of the same array, hence the behavior of the relation ω is defined. If $(x_1\ \omega\ x_2)$ holds and the corresponding pointers point to the members of the same array, then, based on the observation that offset x_i represents exactly the position of the element in the array where the pointer points to, $(p_1\ \omega\ p_2)$ holds. □

Algorithm 17 shows the procedure *resolvePointerVars*(). This procedure is called at the end of the function *resolveConstraint*() (see Algorithm 11) with the resolved path constraint Φ as input. The algorithm iterates over all atomic Boolean expressions of the given constraint and analyzes every variable of all these atoms. If a variable is a pointer, further analysis is required: it is distinguished between the situation when the atom is of a form `p == NULL` or `p != NULL` and all other expressions. If the pointer is compared with `NULL`, only its base address is relevant. In this case a new auxiliary variable `var.A`, corresponding to the base address of the pointer *var*, is created and the occurrence of *var* in the atomic expression *a* is replaced by it. When the atomic expression *a* is not a comparison with `NULL`, then it is a relation of pointers. This case is postponed and the variable `secondCheck`, representing if the repeated analysis of the path constraint is required, is set to *true*. This is done to ensure that the comparison of a pointer with `NULL` is not mistaken for relating to zero the base address of the pointer. After all atoms were examined for comparison with `NULL` and if the repeated analysis is required, it is iterated one more time over all atomic Boolean expressions and their variables. This time all occurrences of comparisons with `NULL` in the path constraint are already eliminated, and, consequently, if a pointer variable is detected in an atom, this variable participates in a relation of pointers. As we have stated in Lemma 5.1, the relation of two pointers is equivalent to the expression (5.1). To build this expression two auxiliary variables are created: `var.x`, corresponding to the offset of the pointer *var* and `var.A`, corresponding to its base address. The occurrence of *var* in the atomic expression *a* is replaced by the auxiliary variable responsible for the offset, and the variable related to the base address is stored in a set

```
inout: $exp$ – expression which pointer variables should be resolved
input: $mem$ – current memory specification

procedure resolvePointerVars($exp$, $mem$){

  $ptrVars = \emptyset$;
  $secondCheck = false$;
  foreach (atom $a$ in $exp$){
    foreach(variable $var$ in $a$){
      if($var$ is a pointer){
        if($a$ == ($var$ == NULL) $\vee$ $a$ == ($var$ != NULL)){
          create var.A, $\upsilon$(var.A) = $\upsilon(var)$;
          replace $var$ by var.A in atom $a$;
        } else {
          $secondCheck = true$;
        }
      }
    }
  }
  if($secondChek$){
    foreach (atom $a$ in $exp$){
      foreach(variable $var$ in $a$){
        if($var$ is a pointer){
          create var.x, $\upsilon$(var.x) = $\upsilon(var)$;
          $exp = exp \wedge$(var.x< $dim(var)$);
          replace $var$ by var.x in atom $a$;

          create var.A, $\upsilon$(var.A) = $\upsilon(var)$;
          $ptrVars = ptrVars \cup$ var.A;
        }
      }
    }
  }
  // build base address relation
  foreach(p.A in $ptrVars$){
    p1.A = next to p.A;
    $c = c \wedge$(p.A == p1.A));
  }
  $exp = exp \wedge c$;
}
```

Algorithm 17: Resolution of pointer variables.

for further processing. Since the whole expression *exp* is originated from the same guard condition, and since we are working with a GIMPLE code, where all expressions are broken down into expressions with no more than 3 operands [1], all pointers occurring in the expression *exp* are related to each other. For this reason, at the end of the procedure $resolvePointerVars()$ a constraint c is created, that requires, that all occurred pointers refer to the same array, which means that all their base addresses must be equal.

We illustrate the described approach on a simple example:

```
1  void test(char *p1,
2            char *p2){
3    if(p1 < p2){
4      ERROR;
5    }
6  }
```

To reach the line with an error, input pointers `p1` and `p2` should fulfill the guard condition `p1 < p2`. To focus on the discussed algorithm, we present the example in a simplified form. Since `p1` and `p2` are inputs, the procedure $resolveConstraint()$ cannot perform any further resolution. Thus, the following path constraint for the guard condition in line 3 is constructed:

$$p1_3 < p2_3 \quad \wedge \quad p1_3 == p1_0 \quad \wedge \quad p2_3 == p2_0$$

where $p1_i$ and $p2_i$ are pointers and therefore the solver is not capable to handle generated constraint. Now this constraint is passed to the procedure $resolvePointerVars()$ from Algorithm 17. $resolvePointerVars()$ analyzes the constraint atom by atom. First, all atoms are examined for comparison with `NULL`. As no such atom could be found, no transformations of the path constraint were performed. However, occurence of relation of pointers was detected, that is, the repeated analysis is required. During this analysis all atoms are examined one more time. First, relation atom $(p1_3 < p2_3)$ is examined. It contains two variables: $p1$ and $p2$, they are evaluated one after another. Variable $p1$ is a pointer. Thus, auxiliary variables $p1@baseAddress$ and $p1@offset$ of type `unsigned int` are created, the occurrence of $p1$ is replaced by the auxiliary variable representing its offset ($p1@offset$) and an additional constraint ensuring safety of array bounds is added. The same analysis is made for the variable $p2$. Atoms $(p1_3 == p1_0)$ and $(p2_3 == p2_0)$ are examined alike. At the end, additional constraints requiring equality of base addresses are added. The result is as follows:

$$\begin{array}{rcl}
p1@offset_3 < p2@offset_3 & \wedge & 0 \le p1@offset_3 < 10 \quad \wedge \quad 0 \le p2@offset_3 < 10 \quad \wedge \\
p1@offset_3 == p1@offset_0 & \wedge & 0 \le p1@offset_3 < 10 \quad \wedge \quad 0 \le p1@offset_0 < 10 \quad \wedge \\
p2@offset_3 == p2@offset_0 & \wedge & 0 \le p2@offset_3 < 10 \quad \wedge \quad 0 \le p2@offset_0 < 10 \quad \wedge \\
 & & p1@baseAddress_3 == p2@baseAddress_3 \quad \wedge \\
 & & p1@baseAddress_3 == p1@baseAddress_0 \quad \wedge \\
 & & p2@baseAddress_3 == p2@baseAddress_0
\end{array}$$

Here the size for the auxiliary arrays is configured equal to 10, because this memory size makes condition $p1 < p2$ feasible (any size ≥ 2 would suffice). Note, that all occurrences of pointer variables are eliminated and now all participating variables are of type `unsigned int` so that the generated constraint can be passed to the solver. The solver returns the following solution (here the different versions of the same variable are not listed for simplicity):

```
p1@baseAddress = 2147483648
p2@baseAddress = 2147483648
p1@offset = 1
p2@offset = 9
```

Now we illustrate how we interpret the obtained solution. We consider the calculated base address as a unique identifier of an auxiliary array: so, if the identifier appears for the first time, a new array is created. If the identifier is already known, the corresponding auxiliary array is taken. In our example identifier 2147483648 appears for the first time in the solution for `p1@baseAddress`, so the new array `p1__autogen_array` is created and the pointer `p1` is initialized with it. When the same identifier appears in the solution for `p2@baseAddress`, it is already known. Therefore, `p2` is also initialized with `p1__autogen_array`. Then offset values are processed and pointers are modified accordingly. It results in the following test driver:

```
char* p1, p2;
char p1__autogen_array[10];
unsigned int p1__autogen_offset;
unsigned int p2__autogen_offset;

p1 = p1__autogen_array;
p2 = p1__autogen_array;

p1__autogen_offset = 1;
p1 += p1__autogen_offset;
p2__autogen_offset = 9;
p2 += p2__autogen_offset;
```

With this test input the erroneous code in procedure `test()` is uncovered. The generated test driver for this example as well as other generator output can be found in Appendix 4.

5.7.3 Address Operation

In this section we discuss the special case of pointer resolution, namely when the value of the pointer to be resolved contains an address operation (like `p = &a + b`). To handle it we use the same concept of the pointer representation as discussed in Section 5.7.2. To define the algorithm we first introduce the following auxiliary functions:

$\alpha : Expression \rightarrow Expression$	Returns operand with address operation. For example, for expression e = `&a + b`, $\alpha(e) =$`&a`.
$\delta : Expression \rightarrow Expression$	Returns operand with offset part of the expression. For example, for expression e = `&a + b`, $\delta(e) =$`b`. If the offset part is not existent, $\delta(e) = 0$.

Algorithm 18 shows the procedure *resolveAddrExp*(). This procedure specifies how an address operation is resolved so that the resulting constraint contains only variables of atomic data types so that the solver is able to handle it. First the variable var_1: variable to which the address operator is applied is identified. For example for expression `&a + b` this is variable `a`. Next, new auxiliary variables `var.A` and $\texttt{var}_1\texttt{.A}$ are created, corresponding to the base address of the pointer *var* and to the base address

```
input: $var$ – pointer identifier which has an address expression as value
       $e$ – address expression
output: $c$ – feasibility constraint
procedure resolveAddrExp($var$, $e$, $c$){

  $a$ = $\alpha(e)$;
  $var_1$ = variable participating in expression $a$

  create var.A, $\upsilon$(var.A) = $\upsilon(var)$;
  create var$_1$.A, $\upsilon$(var$_1$.A) = $\upsilon(var_1)$;
  $c = c \wedge$(var.A == var$_1$.A);

  create var.x,  $\upsilon$(var.x) = $\upsilon(var)$;
  create var$_1$.x, $\upsilon$(var$_1$.x) = $\upsilon(var_1)$;
  $c = c \wedge$(var.x == var$_1$.x+$\omega(a) + \delta(e)$);
  $c = c \wedge$(var.x< $dim(var)$)$\wedge$(var$_1$.x< $dim(var_1)$);

  $c = c \wedge$( var$_1$.x == 0);
}
```

Algorithm 18: Resolution of address operation.

of the variable var_1 respectively. To ensure that the pointer *var* points to the memory location defined by the address operation expression, we require, that the base address of the pointer *var* and the base address of the variable var_1 are equal. Further, we process the offset part of the expression *e*. New auxiliary variables corresponding to offsets of *var* and var_1 are created: `var.x` and `var`$_1$`.x`. The offset difference between `var.x` and `var`$_1$`.x` consists of two parts: (1) defined by the selector – for example when the address expression has form `&a[3]`, this part is represented by $\omega(a)$ in the offset expression and (2) the explicit offset defined by the $\delta(e)$. The safety of array bounds is ensured and in the final step the offset of the `var`$_1$ is set to zero since this offset is already taken into account.

We illustrate the described approach on a simple example:

```
int a[10];
void test_address()
{
  int *p1 = &a[1];
  int *p2 = &a[4];
  if(p1 > p2){
    ERROR;
  }
}
```

After the symbolic execution of the first 5 lines the memory is configured as follows:

$m_1 =$ `(1, ∞, &a[0], 0, 320, int,` a_0`, true)`
$m_2 =$ `(2, 3, &p1, 0, 32, int*, Undef, true)`
$m_3 =$ `(3, 4, &p2, 0, 32, int*, Undef, true)`
$m_4 =$ `(4, ∞, &p1, 0, 32, int*, &`a_4`[1], true)`
$m_5 =$ `(5, ∞, &p2, 0, 32, int*, &`a_5`[4], true)`

Now we process as defined by the procedure *resolveConstraint*() (Section 5.5):

1. Initialize the path constraint according to the guard condition:
$$\Phi = (\texttt{p1}_5 > \texttt{p2}_5).$$
2. Resolve $\texttt{p1}_5$: find the memory item responsible for $\texttt{p1}_5$, this is m_4. Resolve $\texttt{p1}_5$ according to the value of the item found. Since the value of m_4 ($\texttt{\&a}_4\texttt{[1]}$) contains an address operation, *resolveAddrExp*() is invoked. *resolveAddrExp*() creates the auxiliary variables `p1@baseAddress`, `p1@offset` and `a@baseAddress`, `a@offset` and constructs the following constraint:

   $\texttt{p1@baseAddr}_5 == \texttt{a@baseAddr}_4 \quad \wedge$
   $\texttt{p1@offset}_5 == \texttt{a@offset}_4 + 1 \quad \wedge \quad \texttt{a@offset}_4 < 10 \quad \wedge$
   $\texttt{a@offset}_4 == 0.$

   `p1` is a local pointer. This is why its offset is not bounded.
3. Analog to $\texttt{p1}_5$ the resolution of $\texttt{p2}_5$ is performed:

   $\texttt{p2@baseAddr}_5 == \texttt{a@baseAddr}_5 \quad \wedge$
   $\texttt{p2@offset}_5 == \texttt{a@offset}_5 + 4 \quad \wedge \quad \texttt{a@offset}_5 < 10 \quad \wedge$
   $\texttt{a@offset}_5 == 0.$

   Again, `p2` is a local pointer. This is why its offset is not bounded.
4. After these steps only the resolution of $\texttt{a@offset}_i$ and $\texttt{a@baseAddr}_i$ ($i = 4,5$) to their initial values remains. To do it, the original variable $\texttt{a}_i$ ($i = 4,5$) is passed for resolution, which is resolved to $\texttt{a}_0$ according to the value of the memory item m_1. This leads to the following constraint:

   $(\texttt{p1}_5 > \texttt{p2}_5)$
   $(\texttt{p1@baseAddr}_5 == \texttt{a@baseAddr}_4 \quad \wedge$
   $\texttt{p1@offset}_5 == \texttt{a@offset}_4 + 1 \quad \wedge \quad \texttt{a@offset}_4 < 10 \quad \wedge \texttt{a@offset}_4 == 0) \quad \wedge$
   $(\texttt{p2@baseAddr}_5 == \texttt{a@baseAddr}_5 \quad \wedge$
   $\texttt{p2@offset}_5 == \texttt{a@offset}_5 + 4 \quad \wedge \quad \texttt{a@offset}_5 < 10 \quad \wedge \texttt{a@offset}_5 == 0) \quad \wedge$
   $\texttt{a}_4 == \texttt{a}_0 \quad \wedge$
   $\texttt{a}_5 == \texttt{a}_0.$

   The values of $\texttt{a@offset}_i$ and $\texttt{a@baseAddr}_i$ ($i = 4,5$) are not resolved yet, this happens after the pointer resolution is done in the next step.
5. To eliminate pointer occurrences in the generated constraint, the procedure *resolvePointerVars*() is invoked which eliminates occurrences of pointers $\texttt{p1}_5$, $\texttt{p2}_5$ and $\texttt{a}_i$ ($i = 0,4,5$). The resulting path constraint is as follows:

$$\begin{aligned}\Phi = {} & (\texttt{p1@offset}_5 > \texttt{p2@offset}_5) \wedge \\ & (\texttt{p1@baseAddr}_5 == \texttt{a@baseAddr}_4 \quad \wedge \\ & \texttt{p1@offset}_5 == \texttt{a@offset}_4 + 1 \quad \wedge \quad \texttt{a@offset}_4 < 10 \quad \wedge \texttt{a@offset}_4 == 0) \quad \wedge \\ & (\texttt{p2@baseAddr}_5 == \texttt{a@baseAddr}_5 \quad \wedge \\ & \texttt{p2@offset}_5 == \texttt{a@offset}_5 + 4 \quad \wedge \quad \texttt{a@offset}_5 < 10 \quad \wedge \texttt{a@offset}_5 == 0) \quad \wedge \\ & \texttt{a@offset}_4 == \texttt{a@offset}_0 \quad \wedge \texttt{a@baseAddr}_4 == \texttt{a@baseAddr}_0 \\ & \texttt{a@offset}_5 == \texttt{a@offset}_0 \quad \wedge \texttt{a@baseAddr}_5 == \texttt{a@baseAddr}_0\end{aligned}$$

This constraint does not contain any variables of pointer type, so it can be passed to the solver and it can be established, that the line with an error is unreachable.

The generated test driver as well as the other outputs produced by the test generator for this example are presented in Appendix 5.

5.8 Handling of Structures

While handling of an assignment to a structure is already covered by the basic algorithm discussed in 5.4.1, the resolution of an assignment of the form $var == struct.access$ is more complicated. Since the solver cannot handle structure accesses, they must be resolved further up to the value of an atomic type. When such a resolution is not possible because such a structure access is an input, an auxiliary variable of a type corresponding to the accessed field is created and the structure access is resolved to this auxiliary variable. Since the test generator operates on three-address code, we can act on the assumption that we will never have a structure access in a guard condition, i.e. the structure access can occur only on the left side of an assignment resulting from the variable resolution.

First, we introduce auxiliary functions that we need for further definitions and algorithms:

$\nu : Selectors \rightarrow Selectors$	Maps a selector to the corresponding base name. For example $\nu(x.f1.f2) = x$.
$\chi : Expression \times Expression \rightarrow Selectors$	Maps the defined memory area within a structure to the corresponding selector.
$\iota : Expression \times Expression \rightarrow Symbols$	Maps the defined memory area within a structure to the corresponding type.

The resolution of a structure assignment is performed by the procedure call

$$resolveStructExp(var,\ sel,\ offsetStart,\ offsetEnd,\ c,\ mem)$$

Where

- var is a versioned variable identifier. It indicates the variable, that has a structure access as a value.
- sel is a versioned selector of the structure access.
- $offsetStart$ is the start of the demanded memory area within the structure.
- $offsetEnd$ is the end of the demanded memory area within the structure.
- c is a constraint that holds the result of the resolution process.
- mem is the current memory specification.

The Algorithm 19 shows the procedure for the resolution of an assignment of a structure access *resolveStructExp()*. First the algorithm finds all memory items referring to the structure variable from sel. By iterating over the found items the algorithm detects by which of them the validity period correlates with the version of the structure access. If such an item is found, the overlapping of the memory

```
input: $var$ − variable identifier
       $sel$ − structure access expression
       $offsetStart$ − start of the demanded memory area within the structure
       $offsetEnd$ − end of the demanded memory area within the structure
       $mem$ − current memory specification
output: $c$ − feasibility constraint
procedure resolveStructExp($var$, $sel$, $offsetStart$, $offsetEnd$, $c$, $mem$){

  // find out corresponding segment
  $S = \sigma(\beta(sel), mem)$;

  foreach $m = last(S)$ downto $head(S)${
    if ($m.v_0 \leq \upsilon(sel) \wedge \upsilon(sel) \leq m.v_1 \wedge m.a == \beta(sel)$ ){

      $overlap = (offsetStart < m.l) \wedge (m.o < offsetEnd)$;
      $c_1 = m.c \wedge overlap$;

      if($c_1$ is feasible){

        if($m$ refers to an input){

          // create auxiliary variable
          $newVar.name = \nu(sel) + \chi(offsetStart, offsetEnd)$;
          $newVar.type = \iota(offsetStart, offsetEnd)$;
          $\upsilon(newVar) = \upsilon(m.val)$
          $c_1 = c_1 \wedge (var == newVar)$;

        } else if($offsetStart == m.o \wedge offsetEnd == m.l$) {

          resolveExp($var$, $m.val$, $c_1$, $mem$);

        } else {

          // this is nested structure access
          $newOffsetStart = offsetStart - m.o$;
          $newOffsetEnd = offsetEnd - m.o$;
          resolveStructExp($var$, $m.val$, $newOffsetStart$, $newOffsetEnd$, $c_1$, $mem$);

        }
        $c_2 = c_2 \vee c_1$;
      }
    }
  }
  $c = c \wedge c_2$
}
```

Algorithm 19: Resolution of a struct access.

segments corresponding to the memory item and to the demanded memory is examined. The overlapping condition is conjuncted with the feasibility constraint of the found memory item and is stored in constraint c_1. When overlapping occurs and the feasibility constraint is feasible (i.e. c_1 is feasible), the algorithm analyzes three possibilities to proceed:

1. The detected memory item corresponds to an input. In this case a new auxiliary variable is created. Its name is set equally to the base name corresponding to the selector *sel* plus the selector corresponding to the chosen memory area within the structure. Its type is set to the type of the accessed field. The resolution expression of the variable identifier *var* to this new variable is conjuncted with the feasibility constraint c_1.

2. The detected memory item is an exact fit, i.e. it refers to exactly the memory area corresponding to the selector *sel*. In this case the procedure *resolveExp*() (see Section 5.5) is called for further resolution of the value of the memory item *m*. The result of this resolution is stored in the constraint c_1.

3. The detected memory item is not an exact fit, i.e. the memory item describes a memory area that is greater than the memory area corresponding to the selector *sel*. This can happen in case of nested structures. To handle it, the algorithm performs a recursive call with an adjusted offset start and offset end. For example, in the following piece of code the variables `v1` and `v2` are of struct types `s1_t` and `s2_t` correspondingly:

   ```
   v1.f = v2;
   if(v1.f.f1)
     ...
   ```

   During the resolution of an expression `v1.f.f1` we will determine that the memory area corresponding to it has variable `v2` as a value. This is not sufficient and we must resolve it further. Namely find the value of the field `f1` within the variable `v2`. To do it, the offsets calculated for `v1.f.f1` must be adjusted, specifically they must correspond to the offsets of the field `f1` within structure `s2_t`. Therefore the offset of the field `f` in structure `s1_t` is subtracted from the offset start and offset end of the expression `v1.f.f1`.

The constraint produced by this approach is disjuncted with the summarizing constraint c_2, which holds all possible outcomes of the resolution process of the structure access *sel*. This constraint is conjuncted at the end of the algorithm with the resulting constraint *c*.

We illustrate the introduced algorithm by an example. First, we discuss the part of this example where the simple structure access can be resolved up to a value of an atomic type, then we continue with the part where this resolution is not possible since the structure access corresponds to an input and finally we discuss the processing of a nested structure access. First, we define the structure types used in the example:

```
typedef struct{
    int f1;
    int f2;
}nestedStructType_t;

typedef struct{
  int field1;
  int field2;
  nestedStructType_t field3;
}structType_t;
```

Consider now the function `structAccess()`:

C code	GIMPLE representation

```
1  int structAccess(structType_t p1){
2    p1.field1 = 4;
3    if(p1.field1 < 0){
4        return 1;
5    } else if(p1.field2 > 0){
6        return 2;
7    }
8
9    nestedStructType_t tmp;
10   tmp.f1 = 5;
11   tmp.f2 = 7;
12   p1.field3 = tmp;
13
14   if(p1.field3.f1 == 5){
15       return 3;
16   }
17   return 4;
18 }
```
	```
	1  int structAccess(structType_t p1){
	2    struct nestedStructType_t tmp;
	3    int D_1731;
	4    int D_1728;
	5    int D_1727;
	6    int D_1724;
	7    p1.field1 = 4;
	8    D_1724 = p1.field1;
	9    if(D_1724 < 0){
	10     D_1727 = 1
	11   } else {
	12     D_1728 = p1.field2;
	13     if(D_1728 > 0){
	14       d_1727 = 2;
	15     } else {
	16       tmp.f1 = 5;
	17       tmp.f2 = 7;
	18       p1.field3 = tmp;
	19       D_1731 = p1.field3.f1;
	20       if(D_1731 == 5){
	21         D_1727 = 3;
	22       } else {
	23         d_1727 = 4;
	24       }
	25     }
	26   }
	27   return D_1727;
	28 }
	```

Line 3 of this function contains the evaluation of a defined structure member, line 5 contains the evaluation of an undefined structure member and line 14 demonstrates a nested structure access.

Suppose, we want to reach line 4 (line 10 in GIMPLE representation). First, the memory initialization is done (we perform the analysis on the GIMPLE code):

$$m_1 = \quad (1,\ \infty,\ \texttt{\&p1},\ 0,\ 128,\ \texttt{structType\_t},\ \texttt{p1}_0,\ \texttt{true})$$
$$m_2 = \quad (2,\ \infty,\ \texttt{\&tmp},\ 0,\ 64,\ \texttt{int},\ \texttt{Undef},\ \texttt{true})$$

$m_3 =$ (3, ∞ , &D_1731, 0, 32, int, Undef, true)
$m_4 =$ (4, ∞ , &D_1728, 0, 32, int, Undef, true)
$m_5 =$ (5, ∞ , &D_1727, 0, 32, int, Undef, true)
$m_6 =$ (6, ∞ , &D_1724, 0, 32, int, Undef, true)

Execution of line 7 (`p1.field1 = 4;`), overwrites the first 32 bits of the variable `p1`. First, a memory item corresponding to these 32 bits is created:

$m_7 =$ (7, ∞, &p1, 0, 32, structType_t, 4, true)

The insertion of this memory item into the current memory specification invalidates memory item m_1:

$m_1 =$ (1, 6, &p1, 0, 128, structType_t, p1$_0$, true)

and introduces an additional memory item m_8 corresponding to the remains of the memory item m_1 unaffected by the performed operation:

$m_8 =$ (7, ∞, &p1, 32, 128, structType_t, p1$_0$, true)

Execution of line 8 (`D_1724 = p1.field1;`) overwrites the whole memory item m_6. Thus, a new memory item is created:

$m_9 =$ (8, ∞, &D_1724, 0, 32, int, p1.field1$_8$, true)

and the old one is invalidated:

$m_6 =$ (6, 7, &D_1724, 0, 32, int, Undef, true)

The next line of the function under analysis consists of an `if` statement `if(D_1724 < 0)`. This means that the evaluation of the guard condition `(D_1724 < 0)` is necessary. Before we start with the resolution algorithm we summarize the current memory specification:

$m_1 =$ (1, 6, &p1, 0, 128, structType_t, p1$_0$, true)
$m_2 =$ (2, ∞ , &tmp, 0, 64, int, Undef, true)
$m_3 =$ (3, ∞ , &D_1731, 0, 32, int, Undef, true)
$m_4 =$ (4, ∞ , &D_1728, 0, 32, int, Undef, true)
$m_5 =$ (5, ∞ , &D_1727, 0, 32, int, Undef, true)
$m_6 =$ (6, 7, &D_1724, 0, 32, int, Undef, true)
$m_7 =$ (7, ∞, &p1, 0, 32, structType_t, 4, true)
$m_8 =$ (7, ∞, &p1, 32, 128, structType_t, p1$_0$, true)
$m_9 =$ (8, ∞, &D_1724, 0, 32, int, p1.field1$_8$, true)

Now we proceed as defined by the procedure *resolveConstraint*() (Section 5.5):

1. Initialize the path constraint according to the guard condition:

$$\Phi = (\texttt{D\_1724}_8 < 0).$$

2. Resolve `D_1724`$_8$: find the memory item responsible for `D_1724`$_8$, this is m_9. Resolve `D_1724`$_8$ according to the value of the item found:

   $$\texttt{D\_1724}_8 == \texttt{p1.field1}_8.$$

   Now the algorithm *resolveStructExp*() is invoked with `D_1724`$_8$ as *var*, `p1.field1`$_8$ as *sel*, 0 for *offsetStart*, 32 for *offsetEnd*, Φ as c and our memory configuration as *mem*. The passed offsets correspond to the memory area selected by the expression `p1.field1`$_8$. The loop iterates over the memory items m_1, m_7 and m_8, but, since the validity period of m_1 does not match the validity period of `p1.field1`$_8$ and the address range of the item m_8 does not overlap with the calculated offset, only memory item m_7 matches and this match is indeed an exact fit. The procedure *resolveExp*() is invoked and the resolution results in:

   $$\Phi = (\texttt{D\_1724}_8 < 0 \wedge \texttt{D\_1724}_8 == 4).$$

3. No unresolved symbols exist anymore and the resolution process stops. Φ is infeasible, and, since no other path goes to line 10, this line is consequently unreachable.

To demonstrate the case when the resolution of a structure access is not possible because this structure field corresponds to an input, we continue with our example and aim now to cover the line 14 of the GIMPLE representation. The guard condition for the else branch in line 11 is a negation of the guard condition from the previous example, it is resolved to:

$$\Phi = (\texttt{D\_1724}_8 \geq 0 \wedge \texttt{D\_1724}_8 == 4).$$

which is obviously feasible.

We continue with the symbolic execution of line 12 `(D_1728 = p1.field2;)` which overwrites the whole memory item m_4. Thus, a new memory item is created:

$m_{10} =$ `(9, ∞, &D_1728, 0, 32, int, p1.field2`$_9$`, true)`

and the old one is invalidated:

$m_4 =$ `(4, 8, &D_1728, 0, 32, int, Undef, true)`

The next line of the function `if(D_1728 > 0)` requires the evaluation of the guard condition. Before we start with the resolution process, we summarize the current memory specification:

$m_1 =$ `(1, 6, &p1, 0, 128, structType_t, p1`$_0$`, true)`
$m_2 =$ `(2, ∞ , &tmp, 0, 64, int, Undef, true)`
$m_3 =$ `(3, ∞ , &D_1731, 0, 32, int, Undef, true)`
$m_4 =$ `(4, 8, &D_1728, 0, 32, int, Undef, true)`
$m_5 =$ `(5, ∞ , &D_1727, 0, 32, int, Undef, true)`
$m_6 =$ `(6, 7, &D_1724, 0, 32, int, Undef, true)`
$m_7 =$ `(7, ∞, &p1, 0, 32, structType_t, 4, true)`
$m_8 =$ `(7, ∞, &p1, 32, 128, structType_t, p1`$_0$`, true)`

$m_9 =$ `(8, ∞, &D_1724, 0, 32, int, p1.field1`$_8$`, true)`
$m_{10} =$ `(9, ∞, &D_1728, 0, 32, int, p1.field2`$_9$`, true)`

Now we again proceed as defined by the procedure *resolveConstraint*() (Section 5.5):

1. Initialize the path constraint according to the guard condition:
$$\Phi = (\texttt{D\_1728}_9 > 0).$$
2. Resolve `D_1728` $_9$: find the memory item responsible for `D_1728`$_9$, this is m_{10}. Resolve `D_1728`$_9$ according to the value of the item found:
$$\texttt{D\_1728}_9 == \texttt{p1.field2}_9.$$
Now the algorithm *resolveStructExp*() is invoked with `D_1728`$_9$ as *var*, `p1.field2`$_9$ as *sel*, 32 for *offsetStart*, 64 for *offsetEnd*, Φ as *c* and our memory configuration as *mem*. The passed offsets correspond to the memory area selected by the expression `p1.field2`$_9$. The loop iterates over the memory items m_1, m_7 and m_8, but, since the validity period of m_1 does not match the validity period of `p1.field2`$_9$ and the address range of the item m_7 does not overlap with the calculated offset, only memory item m_8 matches. However, memory item m_8 corresponds to an input and cannot be resolved further. Therefore, a new auxiliary variable with name `p1.field2` of type `int` is created and the resolution results in:
$$\Phi = (\texttt{D\_1728} > 0 \wedge \texttt{D\_1728}_9 == \texttt{p1.field2}_0).$$
3. No unresolved symbols exist anymore and the resolution process stops.

And finally, we consider the case of a nested structure access. We continue with our example and now aim to cover line 21 of the GIMPLE representation. The guard condition for the else branch in line 15 is a negation of the guard condition from the previous example, and it is resolved to:
$$\Phi = (\texttt{D\_1728} \leq 0 \wedge \texttt{D\_1728}_9 == \texttt{p1.field2}_0).$$
which is obviously feasible. We continue with the symbolic execution of lines 16-19. For a better understanding of the procedure, we represent it as follows: we list the example code line by line and after each line we specify the memory items which were created by the symbolic execution of this line. The symbolic execution steps are numbered according to the line numbers of the GIMPLE representation listed above.

16 `tmp.f1 = 5;`

$m_{11} =$ `(10, ∞ , &tmp, 0, 32, int, 5, true)`

The insertion of the memory item m_{11} into the memory specification invalidates the memory item m_2, so that now m_2 is configured as follows:

$m_2 =$ `(2, 9, &tmp, 0, 64, int, Undef, true)` and the creation of a memory item corresponding to the unaffected memory area of m_2:

$m_{12} =$ `(10, ∞ , &tmp, 32, 64, int, Undef, true)`

17 `tmp.f2 = 7;`

$m_{13} =$ `(11, ∞ , &tmp, 32, 64, int, 7, true)`

The insertion of the memory item m_{13} into the memory specification invalidates memory item m_{12}, so that now m_{12} is configured as follows:

$m_{12} =$ `(10, 10, &tmp, 32, 64, int, Undef, true)`

18 `p1.field3 = tmp;`

$m_{14} =$ `(12, ∞, &p1, 64, 128, structType_t,` $\texttt{tmp}_{12}$`, true)`

The insertion of the memory item m_{14} into the memory specification invalidates memory item m_8, so that now m_8 is configured as follows:

$m_8 =$ `(7, 11, &p1, 32, 128, structType_t,` $\texttt{p1}_0$`, true)`

and the creation of a memory item corresponding to the unaffected memory area of m_8:

$m_{15} =$ `(12, ∞, &p1, 32, 64, structType_t,` $\texttt{p1}_0$`, true)`

19 `D_1731 = p1.field3.f1;`

$m_{16} =$ `(13, ∞ , &D_1731, 0, 32, int,` $\texttt{p1.field3.f1}_{13}$`, true)`

The insertion of the memory item m_{16} into the memory specification invalidates memory item m_3, so that now m_3 is configured as follows:

$m_3 =$ `(3, 12, &D_1731, 0, 32, int, Undef, true)`

The next line of the function `if(D_1731 == 5)` requires the evaluation of the guard condition. Before we start with the resolution process, we summarize the current memory specification:

$m_1 =$ `(1, 6, &p1, 0, 128, structType_t,` $\texttt{p1}_0$`, true)`
$m_2 =$ `(2, 9, &tmp, 0, 64, int, Undef, true)`
$m_3 =$ `(3, 12, &D_1731, 0, 32, int, Undef, true)`
$m_4 =$ `(4, 8, &D_1728, 0, 32, int, Undef, true)`

$m_5 =$ `(5, ∞ , &D_1727, 0, 32, int, Undef, true)`
$m_6 =$ `(6, 7, &D_1724, 0, 32, int, Undef, true)`
$m_7 =$ `(7, ∞, &p1, 0, 32, structType_t, 4, true)`
$m_8 =$ `(7, 11, &p1, 32, 128, structType_t,` $\texttt{p1}_0$`, true)`
$m_9 =$ `(8, ∞, &D_1724, 0, 32, int,` $\texttt{p1.field1}_8$`, true)`
$m_{10} =$ `(9, ∞, &D_1728, 0, 32, int,` $\texttt{p1.field2}_9$`, true)`
$m_{11} =$ `(10, ∞ , &tmp, 0, 32, int, 5, true)`
$m_{12} =$ `(10, 10, &tmp, 32, 64, int, Undef, true)`
$m_{13} =$ `(11, ∞ , &tmp, 32, 64, int, 7, true)`
$m_{14} =$ `(12, ∞, &p1, 64, 128, structType_t,` $\texttt{tmp}_{12}$`, true)`
$m_{15} =$ `(12, ∞, &p1, 32, 64, structType_t,` $\texttt{p1}_0$`, true)`
$m_{16} =$ `(13, ∞ , &D_1731, 0, 32, int,` $\texttt{p1.field3.f1}_{13}$`, true)`

Now we again process as defined by the procedure *resolveConstraint*() (Section 5.5):

1. Initialize the path constraint according to the guard condition:
$$\Phi = (\texttt{D\_1731}_{13} == 5).$$
2. Resolve $\texttt{D\_1731}_{13}$: find the memory item responsible for $\texttt{D\_1731}_{13}$, this is m_{16}. Resolve $\texttt{D\_1731}_{13}$ according to the value of the item found:
$$\texttt{D\_1731}_{13} == \texttt{p1.field3.f1}_{13}.$$
Now the algorithm *resolveStructExp*() is invoked with $\texttt{D\_1731}_{13}$ as *var*, $\texttt{p1.field3.f1}_{13}$ as *sel*, 64 for *offsetStart*, 96 for *offsetEnd*, Φ as *c* and our memory configuration as *mem*. The passed offsets correspond to the memory area selected by the expression $\texttt{p1.field3.f1}_{13}$. The loop iterates over the memory items m_1, m_7, m_8, m_{14} and m_{15}, but only the memory item m_{14} matches. This is neither an input nor an exact fit – the memory area corresponding to the m_{14} is bigger than selected by $\texttt{p1.field3.f1}_{13}$. Further resolution is necessary. The algorithm performs a recursive call with $\texttt{D\_1731}_{13}$ as *var*, $\texttt{tmp}_{12}$ as *sel* and adjusted offset start (0) and offset end (32) - this is in fact the memory area that corresponds to the field `f1` within the structure `nestedStructType_t`.

   This call determines the memory item m_{11} as a perfect fit and, since the value of m_{11} is 5, the resolution results in:
$$\Phi = (\texttt{D\_1731}_{13} == 5 \wedge \texttt{D\_1731}_{13} == 5).$$
3. No unresolved symbols exist anymore and the resolution process stops.

The generated test driver as well as the other outputs produced by the test generator for this example are presented in Appendix 6.

5.8.1 Pointers and Structures

Usually the pointers to structures are used and not the structures directly. Therefore, it is important to be able to handle the resolution of pointers to structures. We distinguish two cases: assignment to a dereferenced structure pointer and resolution of a structure pointer (i.e. when a dereferenced structure pointer is used in a guard condition). We discuss the algorithms which were developed to handle these cases in the following two sections.

Assignment

The assignment to a dereferenced pointer is discussed in detail in Section 5.6. The only aspect that we have postponed was the case when the value of a pointer is a structure access. Algorithm 20 shows an auxiliary function that was used by Algorithms 13 and 14. A pointer can point to one or more locations, depending on its value and validity constraint. *resolveStructPtrVal*() finds the memory items corresponding to the memory locations where the given pointer can point to. The algorithm takes the found value expression *exp* for the dereferenced pointer, which is a structure access, and the current memory configuration as input. First, the given structure expression is reduced to the structure variable and resolved to the base address and offset by calling the auxiliary function *resolvePtrVal*() (see Algorithm 14). For example, if the given expression is `p->f`, at first the expression `p` is resolved.

The structure access refers only to a part of the pointed memory area and, therefore, it is necessary to filter the found memory items to minimize the effort of the generation process. Thus, all memory items which correspond to the found base address and whose validity period matches the version of the given expression are analyzed if their memory area overlaps with the memory area corresponding to the structure access. If this is the case, the found value is either added to the list of the possible resolutions or is resolved further.

We illustrate the described approach by the following example:

C code	GIMPLE representation
struct birthday_t	int test(){
{	struct person_t * p;
unsigned int day;	struct person_t p1;
unsigned int month;	struct birthday_t bd;
unsigned int year;	struct birthday_t * D_1790;
};	
struct person_t {	p = &p1;
int weight;	p->birthday = &bd;
int height;	D_1790 = p->birthday;
bool isMale;	D_1790->day = 1;
birthday_t *birthday;	...
};	}
int test(){	
person_t *p, p1;	
birthday_t bd;	
p = &p1;	
p->birthday = &bd;	
p->birthday->day = 1;	
...	
}	

We perform the symbolic execution on the GIMPLE code and for a better understanding of the procedure we represent it as follows: we list the example code line by line and after each line we specify the

```
input: $mem$ – current memory specification
       $exp$ – expression that should be resolved
output: $el$ – set of memory items with potential target addresses and offsets
function resolveStructPtrVal($exp$, $mem$){

  $pl$ = resolvePtrVal($\beta(exp)$, $mem$);
  $offsetStart$ = $\omega(exp)$;
  $offsetEnd$ = $\omega(exp)$ + bitsizeof($exp$);
  $el = \emptyset$;

  foreach $m$ in $pl${

    // find out corresponding segment
    $S = \sigma(m.a, mem)$;

    foreach $m' = last(S)$ downto $head(S)${
      if($m'.a$ == $m.a$ && $m'.v_0 \leq \upsilon(exp) \leq m'.v_1$){

        $overlap = (m'.o < offsetEnd + m.o) \wedge (m'.l > offsetStart + m.o)$;
        if($overlap$ is feasible){

          if($m'.val$ is an input){

            // no further resolution is possible
            // create new memory item
            $newOffset$ = $\omega(m'.val) + m'.o$;
            $m'' = (m'.a, newOffset, \ldots, m'.val, m'.c \wedge overlap)$;
            $el = el \cup m''$;

          } else if($m'.val$ is a pointer struct access){
            $el1$ = resolveStructPtrVal($m'.val$, $mem$);
          } else {
            $el1$ = resolvePtrVal($m'.val$, $mem$);
          }

          if($overlap \neq true$){
            foreach $m''$ in $el1${
              $m''.c = m''c \wedge overlap$;
            }
          }
          $el = el \cup el1$;
        }
      }
    }
  }
  return $el$;
}
```

Algorithm 20: Resolution of the pointer structure access to all potential base addresses and offsets.

memory items which were created by the symbolic execution of this line. The symbolic execution steps are numbered according to the line numbers of the GIMPLE representation listed above.

First, the memory is initialized:

```
2 struct person_t * p;
```

$m_1 =$ (1, ∞, &p, 0, 32, person_t*, Undef, true)

```
3 struct person_t p1;
```

$m_2 =$ (2, ∞, &p1, 0, 128, person_t, Undef, true)

```
4 struct birthday_t bd;
```

$m_3 =$ (3, ∞, &bd, 0, 96, birthday_t, Undef, true)

```
5 struct birthday_t * D_1790;
```

$m_4 =$ (4, ∞, &D_1790, 0, 32, birthday_t*, Undef, true)

Then the assignments are processed:

```
7 p = &p1;
```

is processed as was shown in procedure *updateByAssignment*() (Algorithm 9), a new memory item is created:

$m_5 =$ (5, ∞, &p, 0, 32, person_t*, &p1$_5$, true)

The insertion of the created memory item invalidates the memory item m_1:

$m_1 =$ (1, 4, &p, 0, 32, person_t*, Undef, true)

```
8 p->birthday = &bd;
```

is processed as was shown in procedure *updateByAssignmentToDerefPtr()* (Algorithm 13), the dereferenced pointer `p` is first resolved to the base address `&p1`, a new memory item is created:

$m_6 =$ (6, ∞, &p1, 96, 128, person_t, &bd$_6$, true)

The insertion of the new memory item into the memory configuration invalidates the memory item m_2:

$m_2 =$ (`2, 5, &p1, 0, 128, person_t, Undef, true`)

and introduces a new memory item corresponding to the remains of the memory item m_2 not affected by the assignment:

$m_7 =$ (`6,` ∞, `&p1, 0, 96, person_t, Undef, true`)

9 `D_1790 = p->birthday;`

is processed again according to the procedure *updateByAssignment*() (Algorithm 9), a new memory item is created:

$m_8 =$ (`7,` ∞, `&D_1790, 0, 32, birthday_t*,` `p->birthday`$_7$ `, true`)

The insertion of the created memory item invalidates the memory item m_4:

$m_4 =$ (`4, 6, &D_1790, 0, 32, birthday_t*, Undef, true`)

10 `D_1790->day = 1;`

At first sight, this assignment does not differ from the assignment in line 8, but the value of the dereferenced pointer `D_1790` is a pointer struct access (memory item m_8). Therefore, the procedure *resolveStructPtrVal* (Algorithm 20) is invoked with expression `p->birthday`$_7$ as *exp* and our memory configuration as *mem*. First, the pointer `p` is resolved to the base address by the procedure *resolvePtrVal*() (Algorithm 14), and the result contains only one auxiliary memory item with base address `&p1` and offset 0. The loop over corresponding memory items from the memory configuration iterates over m_2, m_6 and m_7. The validity period of the item m_2 does not match with the validity period of `p->birthday`$_7$, and the memory area of m_7 ([0, 96)) does not overlap with the memory area selected by `p->birthday` ([96, 128)). The only item that is left is m_6. Its value (`&bd`$_6$) is not an input and not a structure access. Therefore, it is resolved to the base address according to the procedure *resolvePtrVal*() (Algorithm 14), and the result (base address: `&bd`, offset: 0) is added to the set of memory items of potential target addresses. In this way, the generator detected that the value `1` should be written on the base address `&bd` with offsets [0, 32). So the resulting item is created:

$m_9 =$ (`8,` ∞, `&bd, 0, 32, birthday_t, 1, true`)

Its insertion into the memory configuration invalidates the memory item m_3:

$m_3 =$ (`3, 7, &bd, 0, 96, birthday_t, Undef, true`)

and introduces a new memory item corresponding to the remains of the memory item m_3 not affected by the assignment:

$m_{10} =$ `(8,` ∞`, &bd, 32, 96, birthday_t, Undef, true)`

All statements from our example are processed. We summarize the memory configuration:

$m_1 =$ `(1, 4, &p, 0, 32, person_t*, Undef, true)`
$m_2 =$ `(2, 5, &p1, 0, 128, person_t, Undef, true)`
$m_3 =$ `(3, 7, &bd, 0, 96, birthday_t, Undef, true)`
$m_4 =$ `(4, 6, &D_1790, 0, 32, birthday_t*, Undef, true)`
$m_5 =$ `(5,` ∞`, &p, 0, 32, person_t*, &p1`$_5$` , true)`
$m_6 =$ `(6,` ∞`, &p1, 96, 128, person_t, &bd`$_6$` , true)`
$m_7 =$ `(6,` ∞`, &p1, 0, 96, person_t, Undef, true)`
$m_8 =$ `(7,` ∞`, &D_1790, 0, 32, birthday_t*, p->birthday`$_7$` , true)`
$m_9 =$ `(8,` ∞`, &bd, 0, 32, birthday_t, 1, true)`
$m_{10} =$ `(8,` ∞`, &bd, 32, 96, birthday_t, Undef, true)`

Resolution

The resolution of a structure pointer access (e.g `a == p->m1`) is performed by the procedure call

$$resolveStructPtrExp(var, sel, offsetStart, offsetEnd, c, mem)$$

Where

- *var* is a versioned variable identifier. It indicates the variable, that has a structure access as a value.
- *sel* is a versioned selector of the pointer structure access.
- *offsetStart* is the start of the demanded memory area within the structure.
- *offsetEnd* is the end of the demanded memory area within the structure.
- *c* is a constraint that holds the result of the resolution process.
- *mem* is the current memory specification.

Algorithm 21 shows the procedure for the resolution of an assignment of a pointer structure access $resolveStructPtrExp()$. First, the algorithm performs the resolution of the pointer, which was dereferenced to access the members of the structure. This is done by the auxiliary procedure *resolveStructPtrVal()* (Algorithm 20) in case if the value of the pointer is again a pointer structure access, or by the procedure $resolvePtrVal()$ (Algorithm 14) otherwise. These procedures resolve pointers until the memory where the respective pointer points to is found, and return all possible resolutions. After this is done, the problem reduces itself to the resolution of a structure access, which is performed by the procedure $resolveStructExp()$ (Algorithm 19).

We illustrate our approach by the following example:

```
input: $var$ – variable identifier which has a dereferenced pointer as value
       $sel$ – structure access expression
       $offsetStart$ – start of the demanded memory area within the structure
       $offsetEnd$ – end of the demanded memory area within the structure
       $mem$ – current memory specification
output: $c$ – feasibility constraint
procedure resolveStructPtrExp($var$, $sel$, $offsetStart$, $offsetEnd$, $c$, $mem$){

  // find out corresponding segment
  $S = \sigma(\beta(sel), mem)$;

  foreach $m = last(S)$ downto $head(S)${
    if($m.v_0 \leq \upsilon(sel) \wedge \upsilon(sel) \leq m.v_1 \wedge m.a$ == $\beta(sel)$ ){
      if($m.val$ is a pointer struct access){
        $pl$ = resolveStructPtrVal($m.val$, $mem$);
      } else {
        $pl$ = resolvePtrVal($m.val$, $mem$);
      }

      foreach $m''$ in $pl${

        $offsetStart_1 = offsetStart + m''.o$;
        $offsetEnd_1 = offsetEnd + m''.o$;

        resolveStructExp($var$, $m''.a$, $offsetStart_1$, $offsetEnd_1$, $c_1$, $mem$);
        $c_2 = c_2 \vee c_1$;
      }
    }
    $c = c \wedge c_2$
  }
}
```

Algorithm 21: Resolution of a struct pointer access.

C code	GIMPLE representation
`1 unsigned int CURRENT_MONTH;`	`1 unsigned int CURRENT_MONTH;`
`2 int getAge2(person_t *p)`	`2 int getAge2(person_t *p){`
`3 {`	`3   unsigned int CURRENT_MONTH_0;`
`4   int age;`	`4   unsigned int D_1772;`
`5`	`5   struct birthday_t *D_1771;`
`6   if(CURRENT_MONTH > p->birthday->month &&`	`6   ...`
`7      CURRENT_DAY > p->birthday->day){`	`7   D_1771 = p->birthday;`
`8     age = CURRENT_YEAR -`	`8   D_1772 = D_1771->month;`
`9           p->birthday->year;`	`9   CURRENT_MONTH_0 = CURRENT_MONTH;`
`10   } else {`	`10`
`11    age = CURRENT_YEAR -`	`11   if(D_1772 < CURRENT_MONTH_0){`
`12          p->birthday->year - 1;`	`12     D_1775 = p->birthday;`
`13   }`	`13     ...`
`14`	`14   }`
`15   return age;`	`15 }`
`16 }`	

Where the types `birthday_t` and `person_t` are defined as follows:

```
struct birthday_t
{
  unsigned int day;
  unsigned int month;
  unsigned int year;
};
struct person_t {
  int weight;
  int height;
  bool  isMale;
  birthday_t *birthday;
};
```

We do not demonstrate the whole GIMPLE representation (and symbolic execution) here, but only the evaluation of the first guard condition (the first clause in the `if` statement in line 6 of C code). The complete generator output for this example is presented in Appendix 7.

Nevertheless, for a better understanding of the procedure, we represent it as follows: we list the example code line by line and after each line we specify the memory items which were created by the symbolic execution of this line. The symbolic execution steps are numbered according to the line numbers of the GIMPLE representation listed above.

First, the memory is initialized:

1 `unsigned int CURRENT_MONTH;`

$m_1 =$ `(1, ∞, &CURRENT_MONTH, 0, 32, unsigned int, CURRENT_MONTH`$_0$`, true)`

2 `int getAge2(person_t *p){`

$m_2 =$ (`2`, ∞, `&p, 0, 32, person_t*, &p@P`$_2$`, true`)
$m_3 =$ (`2`, ∞, `&p@P, 0, 96, person_t, p@P`$_0$`, true`)
$m_4 =$ (`2`, ∞, `&p@P, 96, 128, person_t, &p@P.birthday@P`$_2$`, true`)
$m_5 =$ (`2`, ∞, `&p@P.birthday@P, 0, 96, birthday_t, p@P.birthday@P`$_0$, `true`)

Since parameter `p` is of a pointer type, to be able to resolve this pointer and to reason about its contents, an auxiliary variable `p@P` is created and corresponding memory items (m_3, m_4) are constructed. The value of the memory item m_2 is set to the address of the created auxiliary variable. Memory item m_3 corresponds to non-pointer members of the structure `person_t`, the item m_4 corresponds to the member `birthday` of type `birthday_t*`. Since this is a pointer, a new auxiliary variable `p@P.birthday@P` of type `birthday_t` is created, which simulates the variable where the member `birthday` points to. The value of the item m_4 is set to the address of this variable and an item corresponding to it is created (m_5). Since structure `birthday_t` has no pointer members, no further auxiliary variables or memory items are created. In the generator the process of expanding of the structure members is bounded by a parameter so that it does not result in an endless recursion.

```
3 unsigned int CURRENT_MONTH_0;
```

$m_6 =$ (`3`, ∞, `&CURRENT_MONTH_0, 0, 32, unsigned int, Undef, true`)

```
4 unsigned int D_1772;
```

$m_7 =$ (`4`, ∞, `&D_1772, 0, 32, unsigned int, Undef, true`)

```
5 struct birthday_t *D_1771;
```

$m_8 =$ (`5`, ∞, `&D_1771, 0, 32, birthday_t*, Undef, true`)

The assignments are processed:

```
7 D_1771 = p->birthday;
```

$m_9 =$ (`6`, ∞, `&D_1771, 0, 32, birthday_t*, p->birthday`$_6$`, , true`)

Invalidates the memory item m_8:

$m_8 =$ (`5, 5, &D_1771, 0, 32, birthday_t*, Undef, true`)

```
8 D_1772 = D_1771->month;
```

$m_{10} =$ (7, ∞, &D_1772, 0, 32, unsigned int, D_1771->month$_7$, , true)

Invalidates the memory item m_7:

$m_7 =$ (4, 6, &D_1772, 0, 32, unsigned int, Undef, true)

9 CURRENT_MONTH_0 = CURRENT_MONTH;

$m_{11} =$ (8, ∞, &CURRENT_MONTH_0, 0, 32, unsigned int, CURRENT_MONTH$_8$, true)

Invalidates the memory item m_6:

$m_6 =$ (3, 7, &CURRENT_MONTH_0, 0, 32, unsigned int, Undef, true)

Before we start with the resolution of the guard condition in line 11, we summarize the memory configuration:

$m_1 =$ (1, ∞, &CURRENT_MONTH, 0, 32, unsigned int, CURRENT_MONTH$_0$, true)
$m_2 =$ (2, ∞, &p, 0, 32, person_t*, &p@P$_2$, true)
$m_3 =$ (2, ∞, &p@P, 0, 96, person_t, p@P$_0$, true)
$m_4 =$ (2, ∞, &p@P, 96, 128, person_t, &p@P.birthday@P$_2$, true)
$m_5 =$ (2, ∞, &p@P.birthday@P, 0, 96, birthday_t, p@P.birthday@P$_0$, true)
$m_6 =$ (3, 7, &CURRENT_MONTH_0, 0, 32, unsigned int, Undef, true)
$m_7 =$ (4, 6, &D_1772, 0, 32, unsigned int, Undef, true)
$m_8 =$ (5, 5, &D_1771, 0, 32, birthday_t*, Undef, true)
$m_9 =$ (6, ∞, &D_1771, 0, 32, birthday_t*, p->birthday$_6$, , true)
$m_{10} =$ (7, ∞, &D_1772, 0, 32, unsigned int, D_1771->month$_7$, , true)
$m_{11} =$ (8, ∞, &CURRENT_MONTH_0, 0, 32, unsigned int, CURRENT_MONTH$_8$, true)

Now we process as defined by the function *resolveConstraint*() (Algorithm 11):

1. Initialize the path constraint according to the guard condition:

$$\Phi = (\texttt{D\_1772}_8 < \texttt{CURRENT\_MONTH\_0}_8).$$

2. Resolve D_1772$_8$: find the memory item responsible for D_1772$_8$, this is m_{10}. Resolve D_1772$_8$ according to the value of the item found:

$$\texttt{D\_1772}_8 == \texttt{D\_1771->month}_7.$$

Now the algorithm *resolveStructPtrExp*() is invoked with $\texttt{D\_1772}_8$ as *var*, $\texttt{D\_1771->month}_7$ as *sel*, 32 as *offsetStart*, 64 as *offsetEnd*, Φ as *c* and our memory configuration as *mem*. The following items were found for base address `&D_1771`: m_9 and m_8. The validity period of m_8 does not fit the version of *sel*. Thus, only item m_9 matches.

The value of m_9 is $\texttt{p->birthday}_6$, this is structure pointer access, so the auxiliary function *resolveStructPtrVal*() is called. First, another auxiliary function *resolvePtrVal*() for the expression `p` is called. It produces the following specification: the base address is `&p@P` and the offset is 0. The loop iterates over the memory items m_3 and m_4, but only item m_4 overlaps with the memory selected by the expression $\texttt{p->birthday}_6$. The value of m_4 is $\texttt{\&p@P.birthday@P}_2$ this is not an input and not a pointer structure offset, therefore the function *resolvePtrVal*() is invoked again which produces the following specification: the base address is `&p@P.birthday@P` and the offset is 0. This specification is passed to the procedure *resolveStructPtrExp*(), which calls *resolveStructExp*() with $\texttt{D\_1772}_8$ as *var*, $\texttt{\&p@P.birthday@P}_2$ as *sel*, [32, 64) as offsets, Φ as *c* and our memory configuration as *mem*. *resolveStructExp*() determines, that given *sel* corresponds to the memory item m_5, whose value is an input, so it creates an auxiliary variable `p@P.birthday@P.month` of type `unsigned int` and produces the following resolution:

$$\texttt{D\_1772}_8 == \texttt{p@P.birthday@P.month}_0.$$

3. Resolve $\texttt{CURRENT\_MONTH\_0}_8$:

$$\texttt{CURRENT\_MONTH\_0}_8 == \texttt{CURRENT\_MONTH}_0.$$

4. No unresolved symbols exist anymore. Thus, the resolution process stops. The result is as follows:

$$\Phi = (\texttt{D\_1772}_8 < \texttt{CURRENT\_MONTH\_0}_8) \wedge$$
$$(\texttt{D\_1772}_8 == \texttt{p@P.birthday@P.month}_0) \wedge$$
$$(\texttt{CURRENT\_MONTH\_0}_8 == \texttt{CURRENT\_MONTH}_0).$$

Constraint Φ now only contains variables of atomic types, thus the solver is able to reason about it.

The generated test driver as well as the other outputs produced by the test generator for this example are presented in Appendix 7.

5.9 Handling of Bitfields

Since the solver underlying the generation process is capable of handling bitfields, the bitfields processing is mostly done by the solver. However, the preprocessing step that collects all needed information is required. In this section we discuss how this preprocessing step is performed and give an example to illustrate it.

During the preprocessing of the C code to the GIMPLE representation not only are all expressions broken down to expressions with no more than three operands, but also other modifications are made. Amongst them is the transformation of the bitfield evaluation. We observe this in the following example:

C code	GIMPLE representation
`1  typedef struct bitfield_t {`	`1  bitfield_t globalBF;`
`2      uint8_t bit1:1;`	`2  int test()`
`3      uint8_t bit2:1;`	`3  {`
`4      ...`	`4    int retval;`
`5      uint8_t bit12:1;`	`5    unsigned char D_1729;`
`6  } bitfield_t;`	`6    unsigned char D_1730;`
`7`	`7    ...`
`8  bitfield_t globalBF;`	`8    retval = 0;`
`9  int test()`	`9    globalBF.bit11 = 1;`
`10 {`	`10   globalBF.bit5 = 0;`
`11   int retval = 0;`	`11   D_1729 =`
`12   globalBF.bit11 = 1;`	`12     BIT_FIELD_REF <globalBF, 8, 8>;`
`13   globalBF.bit5 = 0;`	`13   D_1730 = D_1729 & 4;`
`14   if(globalBF.bit11 && ...){`	`14   if (D_1730 != 0){`
`15     ...`	`15     ...`
`16   }`	`16   }`
`17   ...`	`17   ...`
`18 }`	`18 }`

In this example a bitfield `bitfield_t` is defined with 12 fields of length 1. Lines 12-13 of the C code demonstrate an assignment to a bitfield and line 14 shows the evaluation of the bitfield. While an assignment to a bit field is also handled in GIMPLE as an assignment to an ordinary structure member, the evaluation looks rather different: first, the content of the byte where the accessed bitfield belongs to is stored in an auxiliary variable (line 11), then the status of the accessed bit is stored in another auxiliary variable (line 13) and, finally, it is evaluated. Since the bitfield expression `BIT_FIELD_REF(var, size, start)` (where `var` is the name of the bitfield, `size` is the size of the extracted segment in bits and `start` is the bit number where the extracted segment starts) does not define a single member of a bitfield but a segment that can contain multiple fields, the values of all these fields must be identified, composed and stored in the auxiliary variable on the right-hand side.

Definition 5.10. The effect of the *assignment of a bitfield* on the state space S_s is specified by the procedure call:

$$updateByBitFieldAssignment(var, exp, n, mem); \quad mem' = mem;$$

where

- *var* is a variable identifier where the bitfield is assigned to,
- *exp* is a bitfield expression that should be assigned,
- *n* is the current computational step,
- *mem* is the current memory specification.

Assignment of a bitfield expression affects only the stack segment.

Algorithm 22 shows the procedure *updateByBitFieldAssignment*(), which specifies how the assignment of the bit field affects the memory specification. First, the procedure calculates offset start and

```
inout: $mem$ – current memory specification
input: $var$ – variable identifier
       $exp$ – bitfield expression
       $n$ – current computational step
procedure updateByBitFieldAssignment($var$, $exp$, $n$, $mem$){

  $offsetStart$ = $exp.start$;
  $offsetEnd$ = $exp.start + exp.size$;

  // find out corresponding segment
  $S = \sigma(\beta(exp.var), mem)$;

  foreach $m = last(S)$ downto $head(S)${
    if ($m.v_1$ == $\infty \wedge m.a$ == $\beta(exp.var) \wedge m.val$ is not an input){
      if ($m.o < offsetEnd \wedge m.o \geq offsetStart \wedge m.l \leq offsetEnd \wedge m.l > offsetStart$){

        MASK = BITMASK($m.l - m.o$) << ($m.o - offsetStart$);
        $c = c \wedge$(($var$ & MASK) == ($m.val$ << ($m.o - offsetStart$)));
      }
    }
  }

  $newExp$ = rttExtract($exp.var$, $offsetEnd - 1$, $offsetStart$);
  $\upsilon(newExp) = \upsilon(exp)$;

  updateByAssignment($var$, $newExp$, $n$, $mem$);

  foreach $m$ new in $mem$ {
    $m.c = m.c \wedge c$;
  }
}
```

Algorithm 22: Effect of the assignment of a bitfield on the memory specification.

offset end of the assigned bitfield based on the given bitfield expression. Next, all valid memory items referring to the base address corresponding to the bitfield name and not referring to an input are found. The memory items referring to an input can be ignored since their values can be set arbitrarily and do not restrict the solver by assigning any values to the bits corresponding to these memory items. If the memory area of the found item correlates with the memory area defined by the bitfield expression, a bit mask is built which corresponds exactly to the bits defined by the current memory item (here with `BITMASK(n)` we denote a bit mask of length `n`, e.g. `BITMASK(3)` is 111 in binary representation). Next, a constraint is built, that reflects that the bits of the variable *var* corresponding to the memory area defined by the found item must be equal to the value of this item. After such constraints are built for all fitting memory items, a bitfield expression, understandable by the solver, is composed, its version is set equal to the version of the bitfield expression *exp* and the procedure *updateByAssignment*() is called where on the left-hand side of the assignment there is a variable identifier *var* where the bitfield is assigned to, and on the right-hand side is the bitfield expression in the solver-required form. After the procedure *updateByAssignment*() is finished, all memory items created by this procedure receive an additional feasibility constraint that characterizes all already defined bits amongst the extracted bits of the bitfield.

We illustrate the described approach by the example function `test()` from the beginning of this section. For a better understanding of the procedure, we represent it as follows: we list the example code line by line and after each line we specify the memory items which were created by the symbolic execution of this line. The symbolic execution steps are numbered according to the line numbers of the GIMPLE representation listed above.

First, the memory is initialized:

```
1 bitfield_t globalBF;
```

$m_1 =$ (1, ∞, `&globalBF`, 0, 16, `bitfield_t`, `globalBF`$_0$, `true`)

```
4 int retval;
```

$m_2 =$ (2, ∞, `&retval`, 0, 32, `int`, `Undef`, `true`)

```
5 unsigned char D_1729;
```

$m_3 =$ (3, ∞, `&D_1729`, 0, 8, `unsigned char`, `Undef`, `true`)

```
6 unsigned char D_1730;
```

$m_4 =$ (4, ∞, `&D_1730`, 0, 8, `unsigned char`, `Undef`, `true`)

Then the assignments are processed:

```
8 retval = 0;
```

$m_5 =$ (5, ∞, &retval, 0, 32, int, 0, true)

Invalidates the memory item m_2:

$m_2 =$ (2, 4, &retval, 0, 32, int, Undef, true)

9 `globalBF.bit11 = 1;`

$m_6 =$ (6, ∞, &globalBF, 10, 11, bitfield_t, 1, true)

Invalidates the memory item m_1:

$m_1 =$ (1, 5, &globalBF, 0, 16, bitfield_t, globalBF$_0$, true)

and introduces additional memory items for remains of m_1 unaffected by the assignment:

$m_7 =$ (6, ∞, &globalBF, 0, 10, bitfield_t, globalBF$_0$, true)
$m_8 =$ (6, ∞, &globalBF, 11, 16, bitfield_t, globalBF$_0$, true)

10 `globalBF.bit5 = 0;`

$m_9 =$ (7, ∞, &globalBF, 4, 5, bitfield_t, 0, true)

Invalidates the memory item m_7:

$m_7 =$ (6, 6, &globalBF, 0, 10, bitfield_t, globalBF$_0$, true)

and introduces additional memory items for the remains of m_7 unaffected by the assignment:

$m_{10} =$ (7, ∞, &globalBF, 0, 4, bitfield_t, globalBF$_0$, true)
$m_{11} =$ (7, ∞, &globalBF, 5, 10, bitfield_t, globalBF$_0$, true)

Before we proceed with the next statement, we summarize the current memory specification:

$m_1 =$ (1, 5, &globalBF, 0, 16, bitfield_t, globalBF$_0$, true)
$m_2 =$ (2, 4, &retval, 0, 32, int, Undef, true)
$m_3 =$ (3, ∞, &D_1729, 0, 8, unsigned char, Undef, true)
$m_4 =$ (4, ∞, &D_1730, 0, 8, unsigned char, Undef, true)
$m_5 =$ (5, ∞, &retval, 0, 32, int, 0, true)
$m_6 =$ (6, ∞, &globalBF, 10, 11, bitfield_t, 1, true)
$m_7 =$ (6, 6, &globalBF, 0, 10, bitfield_t, globalBF$_0$, true)
$m_8 =$ (6, ∞, &globalBF, 11, 16, bitfield_t, globalBF$_0$, true)

$m_9 =$ `(7,` ∞`, &globalBF, 4, 5, bitfield_t, 0, true)`
$m_{10} =$ `(7,` ∞`, &globalBF, 0, 4, bitfield_t,` $\texttt{globalBF}_0$`, true)`
$m_{11} =$ `(7,` ∞`, &globalBF, 5, 10, bitfield_t,` $\texttt{globalBF}_0$`, true)`

11 `D_1729 = BIT_FIELD_REF <globalBF, 8, 8>;`

Line 11 contains an assignment of 8 bits of the bitfield `globalBF` beginning with bit 8 to the variable `D_1729`, so that the procedure *updateByBitFieldAssignment*() is invoked with `D_1729` as *var*, `BIT_FIELD_REF <globalBF, 8, 8>` as *exp*, *n* equal to 8 and our memory configuration. First, offsets are calculated: *offsetStart* is 8 and *offsetEnd* is 16. Then the procedure iterates over all memory items referring to the base address `&globalBF`, but only memory items m_6 and m_8-m_{11} are still valid, and of these memory items only m_6 correlates with the memory defined by the bitfield expression. (Items m_8 and m_{11} refer to inputs, so that the bits, corresponding to these items are still undefined and can have all possible values. Therefore, m_8 and m_{11} do not restrict the solver by assigning any values to the bits corresponding to these items, and, consequently, m_8 and m_{11} can be ignored. Items m_9 and m_{10} do not overlap with the memory defined by the bitfield expression.)

First, we process the memory item m_6: its value is 1 and the created bit mask is 4 (`1 << 2`), the value of the memory item (1) is shifted two bits to the left and this results in the following constraint:

$c =$ `( (D_1729 & 4) == 4)`

Now the bitfield expression is built in the form required by the solver:

exp `=rttExtract(globalBF, 15, 8)`

and the procedure *updateByAssignment*() for assignment

`D_1729 = rttExtract(globalBF, 15, 8)`

is called. This call produces one new memory item:

$m_{12} =$ `(8,` ∞`, &D_1729, 0, 8, unsigned char, rttExtract(`$\texttt{globalBF}_8$`,15,8), true)`

The feasibility constraint of this memory item is amended by the constraint *c*, characterizing the defined bits of the variable `D_1729`:

$m_{12} =$ `(8,` ∞`, &D_1729, 0, 8, unsigned char, rttExtract(`$\texttt{globalBF}_8$`,15,8), (`$\texttt{D\_1729}_8$` & 4 == 4))`

In addition to the creation of a new memory item m_{12}, the procedure *updateByAssignment*() invalidated the memory item m_3:

$m_3 =$ `(3, 7, &D_1729, 0, 8, unsigned char, Undef, true)`

13 `D_1730 = D_1729 & 4;`

$m_{13} =$ (9, ∞, &D_1730, 0, 8, unsigned char, (D_1729$_9$ & 4), true)

Invalidates the memory item m_4:

$m_4 =$ (4, 8, &D_1730, 0, 8, unsigned char, Undef, true)

Before we start with the resolution of a guard condition in line 14, we summarize the memory configuration:

$m_1 =$ (1, 5, &globalBF, 0, 16, bitfield_t, globalBF$_0$, true)
$m_2 =$ (2, 4, &retval, 0, 32, int, Undef, true)
$m_3 =$ (3, 7, &D_1729, 0, 8, unsigned char, Undef, true)
$m_4 =$ (4, 8, &D_1730, 0, 8, unsigned char, Undef, true)
$m_5 =$ (5, ∞, &retval, 0, 32, int, 0, true)
$m_6 =$ (6, ∞, &globalBF, 10, 11, bitfield_t, 1, true)
$m_7 =$ (6, 6, &globalBF, 0, 10, bitfield_t, globalBF$_0$, true)
$m_8 =$ (6, ∞, &globalBF, 11, 16, bitfield_t, globalBF$_0$, true)
$m_9 =$ (7, ∞, &globalBF, 4, 5, bitfield_t, 0, true)
$m_{10} =$ (7, ∞, &globalBF, 0, 4, bitfield_t, globalBF$_0$, true)
$m_{11} =$ (7, ∞, &globalBF, 5, 10, bitfield_t, globalBF$_0$, true)
$m_{12} =$ (8, ∞, &D_1729, 0, 8, unsigned char, rttExtract(globalBF$_8$,15,8), (D_1729$_8$ & 4 == 4))
$m_{13} =$ (9, ∞, &D_1730, 0, 8, unsigned char, (D_1729$_9$ & 4), true)

Now we start the resolution process for the guard condition `D_1730 != 0` as defined by the function *resolveConstraint*() (Algorithm 11):

1. Initialize the path constraint according to the guard condition:

$$\Phi = (\texttt{D\_1730}_9 \,!= 0).$$

2. Resolve D_1730$_9$: find the memory item responsible for D_1730$_9$, this is m_{13}. Resolve D_1730$_9$ according to the value of the item found and add it to the constraint Φ:

$$\Phi = (\texttt{D\_1730}_9 \,!= 0) \wedge (\texttt{D\_1730}_9 == (\texttt{D\_1729}_9 \texttt{ \& 4})).$$

3. Resolve D_1729$_9$: find the memory item responsible for D_1729$_9$, this is m_{12}. Resolve D_1729$_9$ according to the value and the feasibility constraint of the item found and add it to the constraint Φ:

$$\Phi = (\texttt{D\_1730}_9 \,!= 0) \wedge (\texttt{D\_1730}_9 == (\texttt{D\_1729}_9 \texttt{ \& 4})) \wedge$$
$$(\texttt{D\_1729}_9 == \texttt{rttExtract(globalBF}_8\texttt{, 15, 8)} \wedge (\texttt{D\_1729}_8 \texttt{ \& 4 == 4}))$$

4. Resolve D_1729$_8$:

$$\begin{aligned}\Phi = (\texttt{D\_1730}_9 \,!= 0) \wedge (\texttt{D\_1730}_9 == (\texttt{D\_1729}_9 \ \texttt{\&}\ 4)) \wedge \\ (\texttt{D\_1729}_9 == \texttt{rttExtract(globalBF}_8\texttt{, 15, 8)} \wedge (\texttt{D\_1729}_8 \ \texttt{\&}\ 4 == 4)) \wedge \\ (\texttt{D\_1729}_8 == \texttt{rttExtract(globalBF}_8\texttt{, 15, 8)} \wedge (\texttt{D\_1729}_8 \ \texttt{\&}\ 4 == 4))\end{aligned}$$

5. Symbol $\texttt{globalBF}_8$ does not need any resolution, since it was already resolved by the procedure *updateByBitFieldAssignment*(). No unresolved symbols exist anymore and the resolution process stops.

The complete test driver for this example as well as the other generator output is listed in Appendix 8.

5.10 Handling of Unions

The difference between structures and unions is that in a structure each member has its own separate chunk of memory, while in a union each member is allocated at the same piece of storage [13]. All members of a union start at the same address and, depending on the size of a member, it can be completely or partially overwritten by storing a value of another member. Consider the following example:

```
typedef union {
  unsigned short  c2u16;
  unsigned char c2u8[2];
} union_u16;
union_u16 v;
int test_sym1(unsigned short x){
  v.c2u16 = x;
  v.c2u8[0] = 0;
  ...
```

In this example union `union_u16` contains two members: `c2u16` of type `unsigned short`, occupying 16 bits and an array `c2u8` of two elements of type `unsigned char`, so that each of the elements of the array occupies 8 bits. By assignment `v.c2u16 = x`, where `x` is an input parameter of type `unsigned short`, all bits of the variable `v` are overwritten according to the value of the parameter `x`. Dependent on whether the most significant byte of a word is stored at the lowest (big endian) or highest (little endian) memory address, the next assignment `v.c2u8[0] = 0` will overwrite the highest or lowest 8 bits of the variable `v`. Suppose, we are working on a little-endian machine so that the least significant 8 bits of `v` where overwritten with 0. If we now extract `v.c2u16` we get `(x & 0xff00)` as a value.

In the following sections we discuss the algorithms which enable the generator to keep track of the values stored and retrieved from the union.

5.10.1 Assignment

Before we present the algorithm for processing an assignment to a union member, we demonstrate how it operates on our example line by line. First the memory configuration is initialized:

$$\begin{aligned} m_1 &= (\texttt{1, }\infty\texttt{, \&v, 0, 16, union\_u16, v}_0\texttt{, true}) \\ m_2 &= (\texttt{2, }\infty\texttt{, \&x, 0, 16, unsigned short, x}_0\texttt{, true}) \end{aligned}$$

The first assignment `v.c2u16 = x` overwrites the whole memory item m_1 and afterwards the memory configuration is as follows:

$m_1 =$ `(1, 1, &v, 0, 16, union_u16,` v_0`, true)`
$m_2 =$ `(2,` ∞`, &x, 0, 16, unsigned short,` x_0`, true)`
$m_3 =$ `(3,` ∞`, &v, 0, 16, union_u16,` x_3`, true)`

The next assignment `v.c2u8[0] = 0` overwrites only the first 8 bits (we suppose we use a little-endian machine) of the variable `v`. First, a memory item corresponding to these 8 bits is created:

$m_4 =$ `(4,` ∞`, &v, 0, 8, union_u16, 0, true)`

Now we must identify the value of the remaining 8 bits. To do this, we shift the old value (value of the memory item m_3) 8 bits to the right – this way we cut off the overwritten bits – and store the new value in the item m_5 which corresponds to the right-side remains of the memory item m_3:

$m_5 =$ `(4,` ∞`, &v, 8, 16, union_u16,` x_3 `>> 8, true)`

The insertion of the memory items m_4 and m_5 into the memory specification invalidates the memory item m_3. Thus, after processing of the assignment `v.c2u8[0] = 0` the memory is configured as follows:

$m_1 =$ `(1, 1, &v, 0, 16, union_u16,` v_0`, true)`
$m_2 =$ `(2,` ∞`, &x, 0, 16, unsigned short,` x_0`, true)`
$m_3 =$ `(3, 3, &v, 0, 16, union_u16,` x_3`, true)`
$m_4 =$ `(4,` ∞`, &v, 0, 8, union_u16, 0, true)`
$m_5 =$ `(4,` ∞`, &v, 8, 16, union_u16,` x_3 `>> 8, true)`

In this way, we know the values of the bits in range [0; 8) and in range [8; 16) and can reconstruct the values of the union members.

After we have sketched the principle of proceeding of the algorithm, we analyze it in more detail. To be able to handle assignment to a union member, we expand the procedure *insert*() (Algorithm 10) as is shown in Algorithm 23. This is conditioned by the fact that the assignment of a new value to a union member can modify the value of another union member, as we have shown in the example above. The only difference to the earlier version is that after the new memory items for unaffected memory areas are created, their values must be corrected in case when the memory item is of a union type (this part of the algorithm is highlighted in light gray). Depending on whether the generator runs on a big-endian or a little-endian machine, the value of the memory item corresponding to the remains of the old memory item on the left side (m_1) or on the right side (m_2) is adapted.

First we consider the case when the most significant byte is stored on the highest memory address. In this case the memory item m_1 represents the value of the old memory item in the range $[m.o, m'.o)$ (where m is the old memory item and m' is the new memory item created by the assignment to the union member). Since the resolution algorithm for union access, which we will discuss in the next section, builds a bit mask corresponding to the length of the memory area of the item and extracts the values

```
inout: $mem$ – current memory specification
input: $m'$ – memory item to be inserted into the memory specification
       $n$ – current computation step

procedure insert($m'$, $n$, $mem$){

  // find out corresponding segment
  $S = \sigma(m'.a, mem)$;
  $U = \emptyset$;

  foreach $m = last(S)$ downto $head(S)${
    if($m.v_1$ == $\infty$ && $m'.a$ == $m.a$){

      // invalidate found memory item
      $m.v_1 = n$;

      // check if memory items overlap
      if($\neg(m'.l \leq m.o \vee m.l \leq m'.o)$){

        // remains of the old item on the left side
        $c''_1 = m.c \wedge m'.c \wedge m.o < m'.o \wedge m'.o < m.l$;
        $m''_1 = (n, \infty, m.a, m.t, m.o, m'.o, m.val, c''_1)$;
        if($c''_1$ is feasible){
          $U = U \cup \{m''_1\}$;
        }

        // remains of the old item on the right side
        $c''_2 = m.c \wedge m'.c \wedge m.o < m'.l \wedge m'.l < m.l$;
        $m''_2 = (n, \infty, m.a, m.t, m'.l, m.l, m.val, c''_2)$;
        if($c''_2$ is feasible){
          $U = U \cup \{m''_2\}$;
        }
        if($m.t$ is a union){
          if(_LITTLE_ENDIAN_){
            $newValue$ = $m''_2.v$ >> $m''_2.o - m.o$;
            $\upsilon(newValue) = \upsilon(m''_2.v)$;
            $m''_2.v$ = $newValue$;
          } else {
            $newValue$ = $m''_1.v$ >> $m.l - m''_1.l$;
            $\upsilon(newValue) = \upsilon(m''_1.v)$;
            $m''_1.v$ = $newValue$;
          }
        }
      }
    }
  }
  $S = S \cup U \cup \{m'\}$;
}
```

Algorithm 23: Insertion of the new memory item into the memory specification.

from the memory items, which correspond to their bit length, the value of m_1 must not be adapted. The memory item m_2 represents the value of the old memory item in the range $[m'.l, m.l)$. To ensure that this item holds the right value, the least significant bits $[m.o, m'.l)$ of the old value must be cut off, which is done by shifting to the right for $(m'.l - m.o)$ bits.

Now consider the case when the most significant byte is stored on the lowest memory address. In this case the value of m_2 must not be adapted since during the resolution process a mask is built to extract the right value. The memory item m_1 represents the value of the old memory item in the range $[m.o, m'.o)$. Consequently, the least significant bits $[m'.o, m.l)$ must be cut off, which is done by shifting to the right for $(m.l - m'.o)$ bits.

The discussed algorithm supports assignment to a union variable but only when this variable is not a member of an array. However, the presented algorithm can be expanded to support also an assignment to an array member of a union type. For that purpose the memory area which corresponds to the affected array member of a union type should be extracted into the separate memory item and this memory item can be handled in the same manner is it was discussed in the introduced algorithm.

5.10.2 Resolution

In this section we discuss the procedure for the resolution of a union access. Before we present the algorithm, we demonstrate how it proceeds on two simple examples: the first example demonstrates access to a smaller union member after assignment of a bigger one and the second example demonstrates access to a bigger member after the assignment of small ones. In both examples we use the union `union_u16` defined in the previous example. Now we consider the following example:

C code	GIMPLE representation
`1 union_u16 globalV;`	`1 union_u16 globalV;`
`2 int test_sym1(unsigned short x)`	`2 int test_sym1(unsigned short x){`
`3 {`	`3   unsigned char D_1723;`
`4   globalV.c2u16 = x;`	`4   ...`
`5   if(globalV.c2u8[0] == 0xff &&`	`5   globalV.c2u16 = x;`
`6      globalV.c2u8[1] == 85){`	`6   D_1723 = globalV.c2u8[0];`
`7        return 1;`	`7   if(D_1723 == 255){`
`8   }`	`8     ...`
`9   return 0;`	`9 }`
`10 }`	

Assignment `globalV.c2u16 = x` overwrites all bits of the global variable `globalV`, while the following `if` statement performs the evaluation of the smaller members. In this example we analyze only the processing of the first condition of the `if` statement in line 5 of the C code, which performs the evaluation of a member `c2u8[0]` of the union, which corresponds only to a part of the variable `globalV`. We perform symbolic execution on the GIMPLE code. To set the example as simple as possible, we disregard auxiliary variables introduced by GIMPLE and not used in evaluation of the first guard condition, since they are not relevant for our illustration. The complete generator output for this example is presented in Appendix 9.

For a better understanding of the procedure, we represent it as follows: we list the example code line by line and after each line we specify the memory items which were created by the symbolic execution of this line. The symbolic execution steps are numbered according to the line numbers of the GIMPLE representation listed above.

First the memory is initialized. Initialization of globals and parameters:

```
1 union_u16 globalV;
```

$$m_1 = \quad (\texttt{1, }\infty\texttt{, \&globalV, 0, 16, union\_u16, globalV}_0\texttt{, true})$$

```
2 int test_sym1(unsigned short x)
```

$$m_2 = \quad (\texttt{2, }\infty\texttt{, \&x, 0, 16, unsigned short, x}_0\texttt{, true})$$

Subsequently, the stack initialization is done:

```
3 unsigned char D_1723;
```

$$m_3 = \quad (\texttt{3 ,}\infty\texttt{, \&D\_1723, 0, 8, unsigned char, Undef, true})$$

After the initialization is completed, we proceed with the symbolic execution line by line:

```
5 globalV.c2u16 = x;
```

$$m_4 = \quad (\texttt{4, }\infty\texttt{, \&globalV, 0, 16, union\_u16, x}_4\texttt{, true})$$

The insertion of the memory item m_4 into the memory specification invalidates the memory item m_1, so that now m_1 is configured as follows:

$$m_1 = \quad (\texttt{1, 3, \&globalV, 0, 16, union\_u16, globalV}_0\texttt{, true})$$

```
6 D_1723 = globalV.c2u8[0];
```

$$m_5 = \quad (\texttt{5, }\infty\texttt{, \&D\_1723, 0, 8, unsigned char, globalV.c2u8}_5\texttt{[0], true})$$

The insertion of the memory item m_5 into the memory specification invalidates the memory item m_3:

$$m_3 = \quad (\texttt{3, 4, \&D\_1723, 0, 8, unsigned char, Undef, true})$$

The next line of the example consists of the `if` statement `if(D_1723 == 255)`. This means that the evaluation of the guard condition `(D_1723 == 255)` is necessary. Before we start with the resolution algorithm, we summarize the current memory specification:

$$\begin{aligned}
m_1 &= \texttt{(1, 3, \&globalV, 0, 16, union\_u16, globalV}_0\texttt{, true)}\\
m_2 &= \texttt{(2, }\infty\texttt{, \&x, 0, 16, unsigned short, x}_0\texttt{, true)}\\
m_3 &= \texttt{(3, 4, \&D\_1723, 0, 8, unsigned char, Undef, true)}\\
m_4 &= \texttt{(4, }\infty\texttt{, \&globalV, 0, 16, union\_u16, x}_4\texttt{, true)}\\
m_5 &= \texttt{(5, }\infty\texttt{, \&D\_1723, 0, 8, unsigned char, globalV.c2u8}_5\texttt{[0], true)}
\end{aligned}$$

Now we start the resolution process for the guard condition `D_1723 == 255` as defined by the function $resolveConstraint()$ (Algorithm 11):

1. Initialize the path constraint according to the guard condition:

$$\Phi = (\texttt{D\_1723}_5 \texttt{==255}).$$

2. Resolve $\texttt{D\_1723}_5$: find the memory item responsible for $\texttt{D\_1723}_5$, this is m_5. The value of the found memory item is a union member access, so the procedure *resolveUnionExp*() (see Algorithm 24) with $\texttt{D\_1723}_5$ as *var*, $\texttt{globalV.c2u8}_5\texttt{[0]}$ as *sel*, 0 as *offsetStart*, 8 as *offsetEnd* and our memory configuration as *mem*. The passed offsets correspond to the memory area selected by the expression $\texttt{globalV.c2u8}_5\texttt{[0]}$. The loop iterates over the memory items m_1 and m_4, but, since the validity period of m_1 does not match the validity period of $\texttt{globalV.c2u8}_5\texttt{[0]}$, only memory item m_4 matches. The memory area of m_4 overlaps with the memory selected by $\texttt{globalV.c2u8}_5\texttt{[0]}$. Therefore, the auxiliary function *calculateValueForUnion*() is invoked. This function determines that the size of the member is less than the size of the memory area corresponding to the memory item, so that this is the case when a smaller value should be extracted from the bigger one. Suppose, we are working on a little-endian machine. In this case we want to access the lowest 8 bits and the calculated shift is therefore zero. The bit mask for the lowest 8 bits is `0xff` and the calculated value is consequently $\texttt{(0xff \& x}_4\texttt{)}$. Since no other matching memory items exist, the *totalVal* calculated for the union access is $\texttt{(unsigned char)(0xff \& x}_4\texttt{)}$ and the path constraint is now as follows:

$$\begin{gathered}\Phi = (\texttt{D\_1723}_5 \texttt{==255}) \wedge\\ (\texttt{D\_1723}_5 \texttt{ == (unsigned char)(0xff \& x}_4\texttt{))}.\end{gathered}$$

3. Resolve $\texttt{x}_4$: find the responsible memory item, this is m_2. The value of the found item refers to an input atomic variable $\texttt{x}_0$. The resolution expression is built and added to the path constraint Φ:

$$\begin{gathered}\Phi = (\texttt{D\_1723}_5 \texttt{==255}) \wedge\\ (\texttt{D\_1723}_5 \texttt{ == (unsigned char)(0xff \& x}_4\texttt{))} \wedge\\ (\texttt{x}_4 \texttt{==x}_0).\end{gathered}$$

4. No unresolved symbols exist anymore and the resolution process stops.

Now we discuss the case when the bigger union member is accessed after assignment of the small ones:

C code

```
 1  union_u16 globalV;
 2  int test_sym2(unsigned char x,
 3                unsigned char y)
 4  {
 5    globalV.c2u8[0] = x;
 6    globalV.c2u8[1] = y;
 7
 8    if(globalV.c2u16 == 0x5555){
 9      return 1;
10    }
11    return 0;
12  }
```

GIMPLE representation

```
 1  union_u16 globalV;
 2  int test_sym2(unsigned char x,
 3                unsigned char y)
 4  {
 5    short unsigned int D_1727;
 6    ...
 7    globalV.c2u8[0] = x;
 8    globalV.c2u8[1] = y;
 9    D_1727 = globalV.c2u16;
10    if(D_1727 == 21845){
11      ...
12    }
13    ...
14  }
```

Again, to set the example as simple as possible, we disregard auxiliary variables introduced by GIMPLE and not used in evaluation of the first guard condition, since they are not relevant for our illustration. The complete generator output for this example is presented in Appendix 10.

For a better understanding of the procedure, we represent it as follows: we list the example code line by line and after each line we specify the memory items which were created by the symbolic execution of this line. The symbolic execution steps are numbered according to the line numbers of the GIMPLE representation listed above.

First the memory is initialized. Initialization of globals and parameters:

1 `union_u16 globalV;`

$$m_1 = \quad (\texttt{1, } \infty\texttt{, \&globalV, 0, 16, union\_u16, globalV}_0\texttt{, true})$$

2 `int test_sym1(unsigned char x, unsigned char y)`

$$m_2 = \quad (\texttt{2, } \infty\texttt{, \&x, 0, 8, unsigned char, x}_0\texttt{, true})$$
$$m_3 = \quad (\texttt{3, } \infty\texttt{, \&y, 0, 8, unsigned char, y}_0\texttt{, true})$$

Subsequently, the stack initialization is done:

5 `short unsigned int D_1727;`

$$m_4 = \quad (\texttt{4,}\infty\texttt{, \&D\_1727, 0, 16, short unsigned int, Undef, true})$$

After the initialization is completed, we proceed with the symbolic execution line by line:

7 `globalV.c2u8[0] = x;`

$m_5 =$ (5, ∞, &globalV, 0, 8, union_u16, x_5, true)

The insertion of the memory item m_5 into the memory specification invalidates the memory item m_1, so that now m_1 is configured as follows:

$m_1 =$ (1, 4, &globalV, 0, 16, union_u16, $\texttt{globalV}_0$, true)

Furthermore, the insertion of the memory item m_5 provokes the construction of the new memory item m_6 corresponding to the remains of the memory area of the memory item m_1 not overlapping with the memory area of m_5. The value of the new memory item is shifted 8 bits to the right in order to store the value corresponding to the memory area of the item:

$m_6 =$ (5, ∞, &globalV, 8, 16, union_u16, $\texttt{globalV}_0$ >> 8, true)

8 `globalV.c2u8[1] = y;`

$m_7 =$ (6, ∞, &globalV, 8, 16, union_u16, y_6, true)

The insertion of the memory item m_7 into the memory specification invalidates the memory item m_6, so that now m_6 is configured as follows:

$m_6 =$ (5, 5, &globalV, 8, 16, union_u16, $\texttt{globalV}_0$ >> 8, true)

9 `D_1727 = globalV.c2u16;`

$m_8 =$ (7, ∞, &D_1727, 0, 16, short unsigned int, $\texttt{globalV.c2u16}_7$, true)

The insertion of the memory item m_8 into the memory specification invalidates the memory item m_4:

$m_4 =$ (4, 6, &D_1727, 0, 16, short unsigned int, Undef, true)

The next line of the example consists of an `if` statement `if(D_1727 == 21845)`. This means that the evaluation of the guard condition `(D_1727 == 21845)` is necessary. Before we start with the resolution algorithm we summarize the current memory specification:

$m_1 =$ (1, 4, &globalV, 0, 16, union_u16, $\texttt{globalV}_0$, true)
$m_2 =$ (2, ∞, &x, 0, 8, unsigned char, x_0, true)
$m_3 =$ (3, ∞, &y, 0, 8, unsigned char, y_0, true)
$m_4 =$ (4, 6, &D_1727, 0, 16, short unsigned int, Undef, true)
$m_5 =$ (5, ∞, &globalV, 0, 8, union_u16, x_5, true)

$m_6 =$ `(5, 5, &globalV, 8, 16, union_u16,` $\texttt{globalV}_0$ `>> 8, true)`
$m_7 =$ `(6, ∞, &globalV, 8, 16, union_u16,` $\texttt{y}_6$`, true)`
$m_8 =$ `(7, ∞, &D_1727, 0, 16, short unsigned int,` $\texttt{globalV.c2u16}_7$`, true)`

Now we start the resolution process for the guard condition `D_1727 == 21845` as defined by the function *resolveConstraint*() (Algorithm 11):

1. Initialize the path constraint according to the guard condition:

$$\Phi = (\texttt{D\_1727}_7 \texttt{==21845}).$$

2. Resolve $\texttt{D\_1727}_7$: find the memory item responsible for $\texttt{D\_1727}_7$, this is m_8. The value of the found memory item is a union member access, so the procedure *resolveUnionExp*() with $\texttt{D\_1727}_7$ as *var*, $\texttt{globalV.c2u16}_7$ as *sel*, 0 as *offsetStart*, 16 as *offsetEnd* and our memory configuration as *mem*. The passed offsets correspond to the memory area selected by the expression $\texttt{globalV.c2u16}_7$. The loop iterates over the memory items m_1, m_5, m_6 and m_7, but, since the validity periods of m_1 and m_6 do not match the validity period of $\texttt{globalV.c2u16}_7$, only the memory items m_5 and m_7 match. The memory areas of m_5 and m_7 overlap both with the memory selected by $\texttt{globalV.c2u16}_7$, so that this is the case when a value should be constructed from values of multiple memory items. The accessed union member allocates bits in range [0, 16), while the memory item m_7 corresponds to the bits in range [8, 16). This means that the bits of the value of the memory item m_7 must be first brought to the right position by shifting to the right for 8 bits. Further, the casting ensures that the constructed values are in the range of the type of the accessed union member and, therefore, the value of the union access $\texttt{globalV.c2u16}_7$ can be resolved to

$$\texttt{(short unsigned int)(x}_5\texttt{) | (short unsigned int)(y}_6\texttt{ << 8)}$$

and the path constraint is now as follows:

$$\begin{aligned}\Phi = (\texttt{D\_1727}_7 \texttt{==21845}) \wedge \\ (\texttt{D\_1727}_7 \texttt{ == (short unsigned int)(x}_5\texttt{) |} \\ \texttt{(short unsigned int)(y}_6\texttt{ << 8))}.\end{aligned}$$

3. Resolve $\texttt{x}_5$ and $\texttt{y}_6$: find responsible memory items, these are m_2 and m_3 respectively. The values of found items refer both to input atomic variables, so $\texttt{x}_5$ is resolved to $\texttt{x}_0$ and $\texttt{y}_6$ is resolved to $\texttt{y}_0$. The resolution expressions are built and added to the path constraint:

$$\begin{aligned}\Phi = (\texttt{D\_1727}_7 \texttt{==21845}) \wedge \\ (\texttt{D\_1727}_7 \texttt{ == (short unsigned int)(x}_5\texttt{) |} \\ \texttt{(short unsigned int)(y}_6\texttt{ << 8))} \wedge \\ (\texttt{x}_5 \texttt{==x}_0) \wedge (\texttt{y}_6 \texttt{==y}_0).\end{aligned}$$

4. No unresolved symbols exist anymore and the resolution process stops.

After we have sketched the principle of proceeding of the algorithm, we will analyze it in more detail.

```
input: var – variable identifier
       sel – structure access expression
       offsetStart – start of the demanded memory area within the union
       offsetEnd – end of the demanded memory area within the union
       mem – current memory specification
output: c – feasibility constraint
procedure resolveUnionExp(var, sel, offsetStart, offsetEnd, c, mem){

  // find out corresponding segment
  $S = \sigma(\beta(sel), mem)$;

  foreach $m = last(S)$ downto $head(S)${
    if ($m.v_0 \leq \upsilon(sel) \wedge \upsilon(sel) \leq m.v_1 \wedge m.a$ == $\beta(sel)$ ){

      $overlap = (offsetStart < m.l) \wedge (m.o < offsetEnd)$;
      $c_1 = m.c \wedge overlap$;

      if($c_1$ is feasible){
        val = calculateValueForUnion(m, offsetStart, offsetEnd, $c_1$);
        totalVal = totalVal | val;
      }
    }
  }

  $c = c \wedge (var == totalVal)$;
}
```

Algorithm 24: Resolution of a union access.

The resolution of a union assignment is performed by the procedure call

$$resolveUnionExp(var, sel, offsetStart, offsetEnd, c, mem)$$

Where

- *var* is a versioned variable identifier. It indicates the variable, that has a union access as a value.
- *sel* is a versioned selector of the union access.
- *offsetStart* is the start of the demanded memory area within the union.
- *offsetEnd* is the end of the demanded memory area within the union.
- *c* is a constraint that holds the result of the resolution process.
- *mem* is the current memory specification.

Algorithm 24 shows the procedure for the resolution of an assignment of a union access *resolveUnion-Exp()*. First, the algorithm finds all memory items referring to the union variable from the selector *sel*. By iterating over the found items the algorithm detects by which of them the validity period correlates with the version of the union access. As far as such an item is found, the overlapping of the memory segments corresponding to the memory item and to the demanded memory is examined. The overlapping condition is conjuncted with the feasibility constraint of the found memory item and stored in the constraint c_1. When overlapping occurs and the feasibility constraint is feasible (i.e. c_1 is feasible), the algorithm invokes an auxiliary function $calculateValueForUnion()$ (Algorithm 25). This auxiliary function calculates the contribution made by a memory item to the value of the accessed union member. To do this, $calculateValueForUnion()$ first creates an auxiliary variable if the value of the memory item refers to an input like we have already done by processing a structure access in Section 5.8. Further, the algorithm analyzes the following three possibilities:

1. The accessed member has exactly the same size as the value of the memory item. Since we have already ascertained that the accessed memory area and the memory area corresponding to the memory item overlap, the memory item corresponds to exactly the same bits as are selected by the selector and, therefore, we do not have to perform any further calculations.
2. The size of the accessed member is greater than the size of the memory area corresponding to the memory item *m*. This is the case, when the value must be constructed from smaller values. In the last example:

   ```
   globalV.c2u8[0] = x;
   globalV.c2u8[1] = y;
   if(v.c2u16 == 21845){ ...
   ```

   the value of `globalV.c2u16` consists of values of the memory items m_5 and m_7. In order to determine the contribution of each of these items, $calculateValueForUnion()$ performs shifting to the right to bring the bits to the right position and fill the less significant bits with zeros.

```
output: $val$ – calculated value
input: $m$ – memory item, whose contribution must be calculated
        $offsetStart$ – start of the demanded memory area within the union
        $offsetEnd$ – end of the demanded memory area within the union
        $c$ – feasibility constraint
        $mem$ – current memory specification
function calculateValueForUnion($m$, $offsetStart$, $offsetEnd$, $c$, $mem$){

  $baseType$ = $\iota(offsetStart, offsetEnd)$,
  if($m$ refers to an input){
    // create auxiliary variable
    $newVar.name$ = $\nu(m.val) + \chi(offsetStart, offsetEnd)$;
    $newVar.type$ = $\iota(offsetStart, offsetEnd)$;
    $\upsilon(newVar) = \upsilon(m.val)$
    $start$ = $newVar$;
  } else {
    $start$ = $m.val$;
  }

  if($m.l - m.o$ == size($baseType$)){
    $val$ = $start$;
  } else if($m.l - m.o$ < size($baseType$)){
    // a bigger value is constructed from smaller ones
    if(_LITTLE_ENDIAN_){
      $shift$ = $m.o$ – $offsetStart$;
    } else {
      $shift$ = $offsetEnd$ – $m.l$;
    }
    $val$ = $start$ << $shift$;
  } else {
    // a smaller value is extracted from a bigger one
    if(_LITTLE_ENDIAN_){
      $shift$ = $offsetStart$ – $m.o$;
    } else {
      $shift$ = $m.l$ – $offsetEnd$;
    }
    $val$ = $start$ >> $shift$;
  }

  $val$ = BITMASK($offsetEnd$ – $offsetStart$) & $val$;
  $val$ = ($baseType$)($val$);
  if($c$ is not always true){
    $val$ = rttIte(c, val, 0);
  }

  return $val$;
}
```

Algorithm 25: Effect of the assignment to a union member on the memory specification.

3. The size of the accessed member is less than the size of the memory area corresponding to the memory item *m*. This is the case, when the value must be extracted from the bigger value. To determine the value, *calculateValueForUnion*() performs shifting to the right to cut off the bits, that do not participate in the accessed member.

After the value is built in the discussed manner, we must ensure that only relevant bits participate in the result (e.g. for the case when the memory item holds the remains of the bigger value). This is done by a bit mask of length equal to the length of the member. Subsequently, the calculated value is casted to the type of the accessed member and, in case if the passed constraint *c* is not trivially *true*, a conditional assignment understood by the solver is built: `rttIte(c, val, 0)`. This assignment means that if the constraint *c* is evaluated to *true*, the value is equal to *val* and zero otherwise.

The final value for the union member is built from values calculated for each matching memory item by bitwise OR. The resolution expression of the variable *var* to this final value is conjuncted at the end of the algorithm with the resulting constraint *c*.

5.10.3 Pointers and Unions

The handling of unions and pointers consists of two cases: assignment to a dereferenced union pointer and resolution of a union pointer (i.e. when a dereferenced union pointer is used in a guard condition). The assignment to a dereferenced union pointer is covered by the procedure *updateByAssignmentToDerefPtr()* (Algorithm 13) due to expanded *insert*() procedure (Algorithm 23). In this section we present the resolution algorithm for union pointer handling.

The resolution of a union pointer access (e.g `a == p->m1`) is performed by the procedure call

$$resolveUnionPtrExp(var, sel, offsetStart, offsetEnd, c, mem)$$

Where

- *var* is a versioned variable identifier. It indicates the variable, that has a union access as a value.
- *sel* is a versioned selector of the pointer union access.
- *offsetStart* is the start of the demanded memory area within the union.
- *offsetEnd* is the end of the demanded memory area within the union.
- *c* is a constraint that holds the result of the resolution process.
- *mem* is the current memory specification.

Algorithm 26 shows the procedure for the resolution of an assignment of a pointer union access *resolveUnionPtrExp*(). First, the algorithm performs the resolution of the pointer, which was dereferenced to access the members of the union. This is done by the auxiliary procedure *resolveStructPtrVal*() (Algorithm 20) in case if the value of the pointer is a pointer structure access, or by the procedure *resolvePtrVal*() (Algorithm 14) otherwise. These procedures resolve pointers until the memory where the respective pointer points to is found, and return all possible resolutions. After this is done, the

```
input: $var$ – variable identifier which has a dereferenced pointer as value
       $sel$ – union access expression
       $offsetStart$ – start of the demanded memory area within the union
       $offsetEnd$ – end of the demanded memory area within the union
       $mem$ – current memory specification
output: $c$ – feasibility constraint
procedure resolveUnionPtrExp($var$, $sel$, $offsetStart$, $offsetEnd$, $c$, $mem$){

  // find out corresponding segment
  $S = \sigma(\beta(sel), mem)$;

  foreach $m = last(S)$ downto $head(S)${
    if ($m.v_0 \leq \upsilon(sel) \wedge \upsilon(sel) \leq m.v_1 \wedge m.a$ == $\beta(sel)$ ){

      if($m.val$ is a pointer struct access){
        $pl$ = resolveStructPtrVal($m.val, mem$);
      } else {
        $pl$ = resolvePtrVal($m.val, mem$);
      }

      foreach $m''$ in $pl${

        $offsetStart_1 = offsetStart + m''.o$;
        $offsetEnd_1 = offsetEnd + m''.o$;

        resolveUnionExp($var$, $m''.a$, $offsetStart_1$, $offsetEnd_1$, $c_1$, $mem$);
        $c_2 = c_2 \vee c_1$;
      }
    }
    $c = c \wedge c_2$;
  }
}
```

Algorithm 26: Resolution of a union pointer access.

problem reduces itself to the resolution of a union access, which is performed by the procedure *resolve-UnionExp()* (Algorithm 24).

We illustrate our approach by the following example:

C code

```
1   union_u16 globalV;
2   union_u16 *globalP = &globalV;
3   int union_ptr1(unsigned short x)
4   {
5     globalP->c2u16 = x;
6     if(globalP->c2u8[0] == 0xff &&
7        globalP->c2u8[1] == 85){
8        return 1;
9     }
10    return 0;
11  }
```

GIMPLE representation

```
1   union_u16 globalV;
2   union_u16 *globalP = &globalV;
3   int union_ptr1(unsigned short x) {
4     unsigned char D_1754;
5     union union_u16 *globalP_3;
6     union union_u16 *globalP_2;
7
8     globalP_2 = globalP;
9     globalP_2->c2u16 = x;
10    globalP_3 = globalP;
11    D_1754 = globalP_3->c2u8[0];
12    if(d_1754 == 255){
13      ...
14  }
```

In this example we use the same union `union_u16` as was used in the previous examples demonstrating the union handling. We perform symbolic execution on the GIMPLE code. To set an example as simple as possible, we disregard auxiliary variables introduced by GIMPLE and not used in evaluation of the first guard condition, since they are not relevant for our illustration. The complete generator output for this example is presented in Appendix 11.

For a better understanding of the procedure, we represent it as follows: we list the example code line by line and after each line we specify the memory items which were created by the symbolic execution of this line. The symbolic execution steps are numbered according to the line numbers of the GIMPLE representation listed above.

First the memory is initialized. Initialization of globals and parameters:

1 `union_u16 globalV;`

$$m_1 = \quad (\texttt{1, } \infty\texttt{, \&globalV, 0, 16, union\_u16, globalV}_0\texttt{, true})$$

2 `union_u16 *globalP;`

$$m_2 = \quad (\texttt{2, } \infty\texttt{, \&globalP, 0, 32, union\_u16*, \&globalP@P}_2\texttt{, true})$$
$$m_3 = \quad (\texttt{2, } \infty\texttt{, \&globalP@P, 0, 16, union\_u16, globalP@P}_0\texttt{, true})$$

Although the global variable `globalP` is initialized in the example, it is still possible to manipulate its value in a test procedure. To make this possible, the auxiliary variable `globalP@P` is created, which simulates the memory where the `globalP` points to. However, if it is required to use the defined initial values, the generator can be forced to do it via a modifiable parameter.

```
3 int test_sym1(unsigned short x)
```

$m_4 =$ `(3, ∞, &x, 0, 16, unsigned short,` $\mathtt{x}_0$`, true)`

Subsequently, the stack initialization is done:

```
4 unsigned char D_1754;
```

$m_5 =$ `(4, ∞, &D_1754, 0, 8, unsigned char, Undef, true)`

```
5 union union_u16 * globalP_3;
```

$m_6 =$ `(5, ∞, &globalP_3, 0, 32, union_u16*, Undef, true)`

```
6 union union_u16 * globalP_2;
```

$m_7 =$ `(6, ∞, &globalP_2, 0, 32, union_u16*, Undef, true)`

After the initialization is completed, we proceed with the symbolic execution line by line:

```
8 globalP_2 = globalP;
```

$m_8 =$ `(7,∞, &globalP_2, 0, 32, union_u16*,` $\mathtt{globalP}_7$`, true)`

The insertion of the memory item m_8 into the memory specification invalidates the memory item m_7, so that now m_7 is configured as follows:

$m_7 =$ `(6, 6, &globalP_2, 0, 32, union_u16*, Undef, true)`

```
9 globalP_2->c2u16 = x;
```

This assignment is proceeded according to the procedure *updateByAssignmentToDerefPtr()* (Algorithm 13). First the pointer `globalP_2` is resolved by the procedure *resolvePtrVal*() (Algorithm 14) to the base address where this pointer points to. This is `&globalP@P` with offset 0. The member `c2u16` of the union `union_u16` has offsets [0, 16). Therefore, the following memory item is created:

$m_9 =$ `(8, ∞, &globalP@P, 0, 16, union_u16,` $\mathtt{x}_8$`, true)`

The insertion of the memory item m_8 into the memory specification invalidates the memory item m_3, so that now m_3 is configured as follows:

$m_3 =$ `(2, 7, &globalP@P, 0, 16,union_u16,` $\mathtt{globalP@P}_0$`,true)`

10 `globalP_3 = globalP;`

$m_{10} =$ `(9,∞, &globalP_3, 0, 32, union_u16*, globalP`$_9$`, true)`

The insertion of the memory item m_{11} into the memory specification invalidates the memory item m_6, so that now m_6 is configured as follows:

$m_6 =$ `(5, 8, &globalP_3, 0, 32, union_u16*, Undef, true)`

11 `D_1754 = globalP_3->c2u8[0];`

is processed as is shown in procedure *updateByAssignment*() (Algorithm 9), a new memory item is created:

$m_{11} =$ `(10, ∞, &D_1754, 0, 8,unsigned char, globalP_3->c2u8[0]`$_{10}$`,`
`true)`

The insertion of the created memory item invalidates the memory item m_5:

$m_5 =$ `(4, 9, &D_1754, 0, 8, unsigned char, Undef, true)`

Before we start with the resolution of the guard condition in line 12, we summarize the memory configuration:

$m_1 =$ `(1, ∞, &globalV, 0, 16, union_u16, globalV`$_0$`, true)`
$m_2 =$ `(2, ∞, &globalP, 0, 32, union_u16*, &globalP@P`$_2$`, true)`
$m_3 =$ `(2, 7, &globalP@P, 0, 1600,union_u16, globalP@P`$_0$`,true)`
$m_4 =$ `(3, ∞, &x, 0, 16, unsigned short, x`$_0$`, true)`
$m_5 =$ `(4, 9, &D_1754, 0, 8, unsigned char, Undef, true)`
$m_6 =$ `(5, 8, &globalP_3, 0, 32, union_u16*, Undef, true)`
$m_7 =$ `(6, 6, &globalP_2, 0, 32, union_u16*, Undef, true)`
$m_8 =$ `(7,∞, &globalP_2, 0, 32, union_u16*, globalP`$_7$`, true)`
$m_9 =$ `(8, ∞, &globalP@P, 0, 16, union_u16, x`$_8$`, true)`
$m_{10} =$ `(9,∞, &globalP_3, 0, 32, union_u16*, globalP`$_9$`, true)`
$m_{11} =$ `(10, ∞, &D_1754, 0, 8,unsigned char,`
`globalP_3->c2u8[0]`$_{10}$`,true)`

Now we process as defined by the function *resolveConstraint*() (Algorithm 11):

1. Initialize the path constraint according to the guard condition:

$$\Phi = (\texttt{D\_1754}_{10} == 255).$$

2. Resolve `D_1754`$_{10}$: find the memory item responsible for `D_1754`$_{10}$, this is m_{11}. Resolve `D_1754`$_{10}$ according to the value of the item found:

$$\texttt{D\_1754}_{10} == \texttt{globalP\_3->c2u8[0]}_{10}.$$

Now the algorithm *resolveUnionPtrExp*() is invoked with $\texttt{D\_1754}_{10}$ as *var*, $\texttt{globalP\_3->c2u8[0]}_{10}$ as *sel*, 0 as *offsetStart*, 8 as *offsetEnd*, Φ as *c* and our memory configuration as *mem*. The following items were found for the base address `&globalP_3`: m_{10} and m_6. The validity period of m_6 does not fit the version of *sel*. Thus, only item m_{10} matches.

The value of m_{10} is $\texttt{globalP}_9$, this is not a structure pointer access, so the auxiliary function *resolvePtrVal()* is called. It produces the following specification: the base address is `&globalP@P` and the offset is 0. Back in the procedure *resolveUnionPtrExp*(), the procedure *resolveUnionExp()* is called with $\texttt{D\_1754}_{10}$ as *var*, $\texttt{\&globalP@P}_9$ as *sel*, [0, 8) as offsets, Φ as *c* and our memory configuration as *mem*. *resolveUnionExp*() determines, that given *sel* corresponds to the memory items m_3 and m_9, but the validity period of m_3 does not match the validity period of $\texttt{\&globalP@P}_9$. Thus, only memory item m_9 remains. The auxiliary function *calculateValueForUnion()* is invoked with the detected memory item and the defined offsets as inputs. This function determines, that the size of the member is less than the size of the memory area corresponding to the memory item. This is the case when a smaller value should be extracted from the bigger one. Suppose, we are working on a little-endian machine, so that we want to access the lowest 8 bits and the calculated shift is therefore zero. The bit mask for the lowest 8 bits is `0xff` and the calculated value is consequently $\texttt{(0xff \& x}_8\texttt{)}$. Since no other matching memory items exist, the *totalVal* calculated for the union access is $\texttt{(unsigned char)(0xff \& x}_8\texttt{)}$ and the path constraint is now as follows:

$$\Phi = (\texttt{D\_1754}_{10} == 255) \wedge \\ (\texttt{D\_1754}_{10} \texttt{ == (unsigned char)(0xff \& x}_8\texttt{))}.$$

3. Resolve $\texttt{x}_8$: find the responsible memory item, this is m_4. The value of the found item refers to an input atomic variable $\texttt{x}_0$. The resolution expression is built and added to the path constraint Φ:

$$\Phi = (\texttt{D\_1754}_{10} == 255) \wedge \\ (\texttt{D\_1754}_{10} \texttt{ == (unsigned char)(0xff \& x}_8\texttt{))} \wedge \\ (\texttt{x}_8 == \texttt{x}_0).$$

4. No unresolved symbols exist anymore. Thus, the resolution process stops.

5.11 Handling of Arrays

In GIMPLE arrays have two different representations:

1. The input parameters of the array type are represented as pointers and array reads as a dereferenced pointer after the addition of a corresponding offset. For example, the array access `a[x]`, where `a` is an integer array, has the following GIMPLE representation:

```
D_1763 = x * 4;
D_1764 = a + D_1763;
D_1765 = *D_1764;
```

2. All remaining occurrences like global variables of array type or array within a structure type stay unchanged.

The underlying solver SONOLAR [82] that calculates solutions for the generated path constraints is capable of array theories. Therefore, the generator is supported by the solver when handling array inputs. The theory of arrays implemented in SONOLAR has the signature $\sum_A : \{read, write, =\}$ [21]. The function $read(a, i)$ returns the value of the i-th element of array a. The function $write(a, i, e)$ returns array a overwritten on index i with value e while all other array elements remain unchanged. The predicate $=$ can be applied only to array elements, not to arrays. The set of axioms of the theory of arrays is defined as follows [21]:

$$\begin{array}{llll}
\text{(A1)} & i = j & \Rightarrow & read(a,i) = read(a,j) \\
\text{(A2)} & i = j & \Rightarrow & read(write(a,i,e),j) = e \\
\text{(A3)} & i \neq j & \Rightarrow & read(write(a,i,e),j) = read(a,j) \\
\text{(A4)} & a = b & \Leftrightarrow & \forall i(read(a,i) = read(b,i))
\end{array}$$

Furthermore, the equality in the theory of arrays is reflexive, symmetric and transitive.

For the handling of input arrays we use the function $read()$, which has the following notation in SONOLAR: `rttArrayRead(a, i)`. While this function accepts only one-dimensional arrays, this is no limitation, since an array with n dimensions $dim_0, dim_2, \ldots dim_{n-1}$ can be represented as a one-dimensional array with dimension dim, where $dim = dim_0 \cdot dim_2 \cdots dim_{n-1}$. However, only arrays of atomic types can be handled, since SONOLAR does not support structure or union types.

In the two following sections we discuss first the case when an input parameter of an array type must be analyzed (Section 5.11.1) and then all remaining cases of array occurrences (Section 5.11.2).

5.11.1 Handling of Arrays as Input Parameters

As we have already mentioned, the input parameters of array type are represented by GIMPLE as pointers and array reads as a dereferenced pointer after the addition of a corresponding offset. The handling of dereferenced pointers is discussed in Section 5.6. This approach is working well for cases when the dereferenced pointer can be resolved up to a concrete array element, so that only its atomic value has to be determined. However, when the dereferenced pointer refers to an input array and the offset depends on an input, the discussed algorithm is not sufficient. In this section we will, therefore, extend it, so that it will be able to handle array inputs with input indices.

First, we illustrate by an example where the algorithm discussed in Section 5.6 reaches its limits. The module under test `test()` compares two elements of the integer array `a` and returns $true$ if the element `a[x]` is greater then the element `a[y]` and $false$ otherwise. To ensure that passed values of `x` and `y` are within the array bounds, we restrict their range by a precondition.

C code	GIMPLE representation
`1 #define N 2`	`1 int test(int *a, int x,`
`2 typedef int my_array[N];`	`2         int y){`
`3 int test(my_array a,`	`3   int D_1768;`
`4          unsigned int x,`	`4   int *D_1767;`
`5          unsigned int y)`	`5   unsigned int D_1766;`
`6 {`	`6   int D_1765;`
`7     __rtt_precondition(x < N && y < N);`	`7   int *D_1764;`
`8`	`8   unsigned int D_1763;`
`9     int retval = 0;`	`9   ...`
`10    if(a[x] > a[y]){`	`10  D_1763 = x * 4;`
`11        retval = 1;`	`11  D_1764 = a + D_1763;`
`12    } else {`	`12  D_1765 = *D_1764;`
`13        retval = 0;`	`13  D_1766 = y * 4;`
`14    }`	`14  D_1767 = a + D_1766;`
`15    return retval;`	`15  D_1768 = *D_1767;`
`16 }`	`16  if (D_1765 > D_1768){`
	`17    ...`
	`18  }`
	`19 }`

We demonstrate here not the whole GIMPLE representation (and symbolic execution) but only the evaluation of the guard condition of the `if` statement in line 10 of the C code and the symbolic execution of the relevant statements (the execution of statements involving the variable `retval` as well as of the precondition is omitted). The complete generator output for this example is presented in Appendix 12.

For a better understanding of the procedure, we represent it as follows: we list the example code line by line and after each line we specify the memory items which were created by the symbolic execution of this line. The symbolic execution steps are numbered according to the line numbers of the GIMPLE representation listed above.

First the memory is initialized. Initialization of parameters:

```
1 int test(int *a, int x, int y)
```

$$m_1 = \quad (\texttt{1}, \infty, \texttt{\&a, 0, 32, int*, \&a@P[0]}_1, \texttt{true})$$
$$m_2 = \quad (\texttt{1}, \infty, \texttt{\&a@P[0], 0, 3200, int[100], a@P}_0\texttt{, true})$$
$$m_3 = \quad (\texttt{2}, \infty, \texttt{\&x, 0, 32, unsigned int, x}_0, \texttt{true})$$
$$m_4 = \quad (\texttt{3}, \infty, \texttt{\&y, 0, 32, unsigned int, y}_0, \texttt{true})$$

For the parameter `a` two memory items were created: m_1 and m_2, where m_2 corresponds to an auxiliary array `a@P`, which simulates the memory were the pointer parameter `a` points to.

Subsequently, the stack initialization is done:

```
3 int D_1768;
```

$$m_5 = \quad (\texttt{4}, \infty, \texttt{\&D\_1768, 0, 32, int, Undef, true})$$

4 `int *D_1767;`

$m_6 =$ (`5, ∞, &D_1767, 0, 32, int*, Undef, true`)

5 `unsigned int D_1766;`

$m_7 =$ (`6, ∞, &D_1766, 0, 32, unsigned int, Undef, true`)

6 `int D_1765;`

$m_8 =$ (`7, ∞, &D_1765, 0, 32, int, Undef, true`)

7 `int *D_1764;`

$m_9 =$ (`8, ∞, &D_1764, 0, 32, int*, Undef, true`)

8 `unsigned int D_1763;`

$m_{10} =$ (`9, ∞, &D_1763, 0, 32, unsigned int, Undef, true`)

After the initialization is completed, we proceed with the symbolic execution line by line:

10 `D_1763 = x * 4;`

$m_{11} =$ (`10, ∞, &D_1763, 0, 32, unsigned int,` $\texttt{x}_{10} \cdot 4$`, true`)

The insertion of the memory item m_{11} into the memory specification invalidates the memory item m_{10}, so that now m_{10} is configured as follows:

$m_{10} =$ (`9, 9, &D_1763, 0, 32, unsigned int, Undef, true`)

11 `D_1764 = a + D_1763;`

$m_{12} =$ (`11, ∞, &D_1764, 0, 32, int*,` $\texttt{a}_{11} + \texttt{D\_1763}_{11}$`, true`)

The insertion of the memory item m_{12} into the memory specification invalidates the memory item m_9:

$m_9 =$ (`8, 10, &D_1764, 0, 32, int*, Undef, true`)

12 `D_1765 = *D_1764;`

$m_{13} =$ (12, ∞, &D_1765, 0, 32, int, *D_1764$_{12}$, true)

The insertion of the memory item m_{13} into the memory specification invalidates the memory item m_8:

$m_8 =$ (7, 11, &D_1765, 0, 32, int, Undef, true)

13 D_1766 = y * 4;

$m_{14} =$ (13, ∞, &D_1766, 0, 32, unsigned int, y$_{13}\cdot$4, true)

The insertion of the memory item m_{14} into the memory specification invalidates the memory item m_7:

$m_7 =$ (6, 12, &D_1766, 0, 32, unsigned int, Undef, true)

14 D_1767 = a + D_1766;

$m_{15} =$ (14, ∞, &D_1767, 0, 32, int*, a$_{14}$+D_1766$_{14}$, true)

The insertion of the memory item m_{15} into the memory specification invalidates the memory item m_6:

$m_6 =$ (7, 13, &D_1767, 0, 32, int*, Undef, true)

15 D_1768 = *D_1767;

$m_{16} =$ (15, ∞, &D_1768, 0, 32, int, *D_1767$_{15}$, true)

The insertion of the memory item m_{16} into the memory specification invalidates the memory item m_5:

$m_5 =$ (4, 14, &D_1768, 0, 32, int, Undef, true)

The next line of the example consists of an if statement if(D_1765 > D_1768). This means that the evaluation of the guard condition (D_1765 > D_1768) is necessary. Before we start with the resolution algorithm, we summarize the current memory specification:

$m_1 =$ (1, ∞, &a, 0, 32, int*, &a@P[0]$_1$, true)
$m_2 =$ (1, ∞, &a@P[0], 0, 3200, int[100], a@P$_0$, true)
$m_3 =$ (2, ∞, &x, 0, 32, unsigned int, x$_0$, true)

$m_4 =$ (3, ∞, &y, 0, 32, unsigned int, y_0, true)
$m_5 =$ (4, 14, &D_1768, 0, 32, int, Undef, true)
$m_6 =$ (5, 13, &D_1767, 0, 32, int*, Undef, true)
$m_7 =$ (6, 12, &D_1766, 0, 32, unsigned int, Undef, true)
$m_8 =$ (7, 11, &D_1765, 0, 32, int, Undef, true)
$m_9 =$ (8, 10, &D_1764, 0, 32, int*, Undef, true)
$m_{10} =$ (9, 9, &D_1763, 0, 32, unsigned int, Undef, true)
$m_{11} =$ (10, ∞, &D_1763, 0, 32, unsigned int, $x_{10} \cdot 4$, true)
$m_{12} =$ (11, ∞, &D_1764, 0, 32, int*, a_{11}+D_1763$_{11}$, true)
$m_{13} =$ (12, ∞, &D_1765, 0, 32, int, *D_1764$_{12}$, true)
$m_{14} =$ (13, ∞, &D_1766, 0, 32, unsigned int, $y_{13} \cdot 4$, true)
$m_{15} =$ (14, ∞, &D_1767, 0, 32, int*, a_{14}+D_1766$_{14}$, true)
$m_{16} =$ (15, ∞, &D_1768, 0, 32, int, *D_1767$_{15}$, true)

Now we process as defined by the function *resolveConstraint*() (Algorithm 11):

1. Initialize the path constraint according to the guard condition:
$$\Phi = (\texttt{D\_1765}_{15} > \texttt{D\_1768}_{15}).$$
2. Resolve D_1765$_{15}$: find the memory item responsible for D_1765$_{15}$, this is m_{13}. Resolve D_1765$_{15}$ according to the value of the item found:
$$\texttt{D\_1765}_{15} == *\texttt{D\_1764}_{12}.$$
Now the algorithm *resolveDerefPtr()* (see Algorithm 15) is invoked with D_1765$_{15}$ as *var*, *D_1764$_{12}$ as *p*, Φ as *c* and our memory configuration as *mem*. It passes further resolution to the algorithm *resolveDerefPtrExp()* (see Algorithm 16). The resolution of the memory item m_{12} by the function *resolvePtrVal*() produces the following specification: the base address is &a@P[0] and the offset is $x_0 \cdot 4$ (the value of the memory item m_{11} is ($x_{10} \cdot 4$), x_{10} is resolved up to the input). The internal loop finds matching memory item m_2. Now the algorithm *resolveDerefPtrExp()* invokes the subroutine *resolveExp*() (Algorithm 12), but the value (a@P$_0$) of the found memory item m_2 is an input and cannot be resolved further. However, neither this value can be passed to the solver in an expression like (D_1765$_{15}$ ==a@P$_0$), since it is of an array type, so that the corresponding expression makes no sense. To be able to handle this situation, we must first extend the algorithm *resolveDerefPtrExp()*.

Algorithm 27 shows the extended function *resolveDerefPtrExp()* (the unextended version is shown in Algorithm 16). The part of the algorithm that was introduced for the handling of input arrays is highlighted in light gray. As input arrays are represented by GIMPLE as pointers and the information, if this is a pointer or an array, is lost, we represent all input pointers (except of pointers to union types) as arrays of a modifiable size. Since all array elements which have the same value can be represented as a single memory item, this generates merely a slight overhead. Now, if a pointer refers to an input, we can act on the assumption, that this pointer refers to an input array. In this case the array expression resolving the variable *var* to the element of array $m'.val$ at index *i* by invocation of rttArrayRead() is built.

```
input: $var$ − variable identifier which has a dereferenced pointer as value
       $p$ − pointer identifier
       $mem$ − current memory specification
       $validFrom$ − indicates the validity period of matching memory items
inout: $R$ − set of found resolutions
       $isInput$ − indicates whether the resolution is performed for a pointer input
procedure resolveDerefPtrExp($var$, $p$, $R$, $mem$, $isInput$, $validFrom$){

  // find out corresponding segment
  $S = \sigma(\beta(p), mem)$;
  $offsetStart = \omega(p)$;
  $offsetEnd = \omega(p)$ + size(basetype($p$));
  foreach $m = last(S)$ downto $head(S)${
    if($m.v_0 \leq \upsilon(p) \wedge \upsilon(p) \leq m.v_1 \wedge m.a$ == $\beta(p)$ ){

      if($m.val$ is a pointer struct access){
        $pl$ = resolveStructPtrVal($m.val, mem$);
      } else {
        $pl$ = resolvePtrVal($m.val, mem$);
      }
      foreach $m''$ in $pl${

        // for each memory item specification in the list
        // find all items overlapping with it
        $S_1 = \sigma(m''.a, mem)$;

        foreach $m' = last(S_1)$ downto $head(S_1)${
          if(($isInput \wedge validFrom < m'.v_0 \wedge m'$ refers to a simulated input)$\vee$ !$isInput$)){
            if($m'.v_0 \leq \upsilon(var) \wedge \upsilon(var) \leq m'.v_1 \wedge m'.a$ == $m''.a$){
              if($m'.val$ is not an input) {
                $overlap = (m'.o < m''.o + offsetEnd) \wedge (m'.l > m''.o + offsetStart)$;
                $c_1 = m.c \wedge m''.c \wedge m'.c \wedge overlap$;
                if($c_1$ is feasible){
                  resolveExp($var, m'.val, c_2, mem$);
                  $R$.push(($c_2, p, c_1, m'.v_0$));
                  if($m'$ refers to a simulated input) $isInput$ = $true$;
                }
              } else {
                $\upsilon(i) = \upsilon(p)$;
                $idxExp = ((i == (m''.o \cdot 8)/$size(basetype($p$)))$\wedge (m'.o \leq i \cdot$size(basetype
                    ($p$))$< m'.l$));
                $arrayExp = (var ==$rttArrayRead($m'.val$, $i$));
                $c_1 = m.c \wedge m''.c \wedge m'.c$;
                if($c_1$ is feasible){
                  $c_2 = arrayExp \wedge idxExp$;
                  $R$.push(($c_2, p, c_1, m'.v_0$));
                  if($m'$ refers to a simulated input) $isInput$ = $true$;
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Algorithm 27: Resolution of a dereferenced pointer, extended.

The index expression holds the resolution expression for the index where the array read is performed. To build an index expression an auxiliary variable `i` is introduced. The version of this auxiliary variable is set equal to the version of the pointer identifier p, since the array read is made in the computational step corresponding to this version. The value of the index is required to be equal to the scaled offset expression calculated by the auxiliary function `resolveStructPtrVal()` or `resolvePtrVal()` depending on the value of the memory item m (`basetype()` is here an auxiliary function that maps the pointer selector to its base type, see Section 5.14). Furthermore, it is ensured that the index is within the bounds of the fitting memory item m'.

After the index and array expressions are constructed, the constraint c_1 is built, which requires that the validity constraints of all participating memory items m, m' and m'' are valid. If this constraint is feasible, constraint c_2 is built, which represents the resolution of the dereferenced pointer and requires that the index and array expressions are valid. This constraint is stored together with the corresponding pointer, feasibility constraint and the validity period in the resulting set of possible outcomes of the resolution process of the dereferenced pointer `p`. If it is detected that the memory item m' refers to a simulated input variable, the input/output parameter *isInput* indicating whether the dereferenced pointer still points to an input is set to *true*.

After we have defined how the resolution of the input array is handled, we are able to proceed with our example. We continue with step 2 of the resolution process:

2. The dereferenced pointer $\texttt{*D\_1764}_{12}$ was already resolved to the base address `&a@P[0]` and offset $\texttt{x}_0 \cdot 4$. The matching memory item m_2 was found. Since the value of m_2 is an array, the new part of the algorithm *resolveDerefPtrExp()* is now invoked. The index and array expressions are built:

$$\begin{aligned} \mathit{idxExp} &= (\texttt{idx0}_{12}\ \texttt{==}\ \texttt{x}_0 \wedge 0 \le \texttt{idx0}_{12} \cdot 32 < 3200) \\ \mathit{arrayExp} &= (\texttt{D\_1765}_{15}\ \texttt{==rttArrayRead(a@P}_0\texttt{, idx0}_{12}\texttt{))} \end{aligned}$$

   The value of the index $\texttt{idx0}_{12}$ was scaled according to the size of array elements.

   Now the following tuple is stored in the resolution set R:

$$(\mathit{idxExp} \wedge \mathit{arrayExp}, \texttt{D\_1764}_{12}, \mathit{true}, 1).$$

   Although the memory item m_2 refers to a simulated input `a@P`, no further input pointers are detected and the resolution of $\texttt{D\_1765}_{15}$ results in:

$$\begin{aligned} \Phi = (\texttt{D\_1765}_{15} > \texttt{D\_1768}_{15}) \wedge (\texttt{D\_1765}_{15}\ \texttt{==rttArrayRead(a@P}_0\texttt{, idx0}_{12}\texttt{))} \wedge \\ (\texttt{idx0}_{12}\ \texttt{==}\ \texttt{x}_0 \wedge 0 \le \texttt{idx0}_{12} \cdot 32 < 3200). \end{aligned}$$

3. Resolve $\texttt{D\_1768}_{15}$: the resolution proceeds similar to the resolution of $\texttt{D\_1765}_{15}$ and results in:

$$\begin{aligned} \Phi = (\texttt{D\_1765}_{15} > \texttt{D\_1768}_{15}) \wedge (\texttt{D\_1765}_{15}\ \texttt{==rttArrayRead(a@P}_0\texttt{, idx0}_{12}\texttt{))} \wedge \\ (\texttt{idx0}_{12}\ \texttt{==}\ \texttt{x}_0 \wedge 0 \le \texttt{idx0}_{12} \cdot 32 < 3200) \wedge \\ (\texttt{D\_1768}_{15}\ \texttt{==rttArrayRead(a@P}_0\texttt{, idx0}_{15}\texttt{))} \wedge \\ (\texttt{idx0}_{15}\ \texttt{==}\ \texttt{y}_0 \wedge 0 \le \texttt{idx0}_{15} \cdot 32 < 3200). \end{aligned}$$

4. No unresolved symbols exist anymore. Thus, the resolution process stops and the constructed path constraint is passed to the solver.

The solver determines Φ as feasible and returns the following solution (we omit here the listing of the calculated values for local and auxiliary variables, since they do not affect the generated test case):

```
a@P[0] = −513
a@P[1] = −1
x = 1
y = 0
```

Based on the calculated solution, the generator constructs the following test driver:

```
int *a;
unsigned int x;
unsigned int y;
int a_rtt_array[100];
int __rtt_return;
@rttBeginTestStep;
{
  y = 0;
  x = 1;
  a_rtt_array[0] = −513;
  a = a_rtt_array;
  a_rtt_array[1] = −1;
  a = a_rtt_array;

  @rttCall(__rtt_return = test(a, x, y));
}
@rttEndTestStep;
```

As was already mentioned, CTGEN generates tests in RT-Tester syntax [44]. To make settings in the input array, the auxiliary array `a_rtt_array` is created. Its values are set according to the calculated by the solver and the input parameter `a` is set to this array. The values of `x` and `y` are set appropriately to the solution. These settings satisfy the guard condition `(a[x] > a[y])` and, consequently, this test will cover the intended branch. The complete test script as well as the other outputs produced by the generator for the discussed example can be observed in Appendix 12.

5.11.2 Handling of Arrays in remaining Cases

In this section we discuss the algorithm for array handling in cases when a variable of array type is not altered by GIMPLE to a variable of pointer type but remains as it was. This occurs in all cases when the array variable is not an input parameter, e.g if it is a global or local variable.

Before we present the algorithm for the resolution of an array expression, we demonstrate on a simple example, how it proceeds, line by line. The module under test `example()` has the following inputs: a global integer array `aG` and an integer parameter `x`. It returns *true*, if the element at index `x` is equal to 2 and *false* otherwise.

C code

```
1  unsigned int aG[10];
2  int example(unsigned int x){
3    aG[0] = 1;
4    aG[3] = 2;
5    if(aG[x] == 2)
6      return 1;
7    return 0;
8  }
```

GIMPLE representation

```
 1  unsigned int aG[10];
 2  int example(unsigned int x){
 3    unsigned int D_1714;
 4    ...
 5    aG[0] = 1;
 6    aG[3] = 2;
 7    D_1714 = aG[x];
 8    if(D_1714 == 2){
 9      ...
10    }
11  }
```

We demonstrate here not the complete GIMPLE representation (and symbolic execution) but only the evaluation of the guard condition of the `if` statement in line 6 of C code and the symbolic execution of the relevant statements. The complete generator output for this example is presented in Appendix 13.

For a better understanding of the procedure, we represent it as follows: we list the example code line by line and after each line we specify the memory items which were created by the symbolic execution of this line. The symbolic execution steps are numbered according to the line numbers of the GIMPLE representation listed above.

First, the memory is initialized. Initialization of globals and parameters:

1 `unsigned int aG[10];`

$$m_1 = \texttt{(1, } \infty\texttt{, \&aG[0], 0, 320, int[10], aG}_0\texttt{, true)}$$

2 `int example(unsigned int x)`

$$m_2 = \texttt{(2, } \infty\texttt{, \&x, 0, 32, unsigned int, x}_0\texttt{, true)}$$

Subsequently, the stack initialization is done:

3 `unsigned int D_1714;`

$$m_3 = \texttt{(3, } \infty\texttt{, \&D\_1714, 0, 32, unsigned int, Undef, true)}$$

After the initialization is completed we proceed with the symbolic execution line by line:

5 `aG[0] = 1;`

$$m_4 = \texttt{(4, } \infty\texttt{, \&aG[0], 0, 32, int[10], 1, true)}$$

The insertion of the memory item m_4 into the memory specification invalidates the memory item m_1, so that now m_1 is configured as follows:

$m_1 =$ (1, 3, &aG[0], 0, 320, int[10], aG$_0$, true)

Furthermore, the insertion of the memory item m_4 triggers the construction of the new memory item m_5 corresponding to the remains of the memory area of the memory item m_1 not overlapping with the memory area of m_4:

$m_5 =$ (4, ∞, &aG[0], 32, 320, int[10], aG$_0$, true)

6 aG[3] = 2;

$m_6 =$ (5, ∞, &aG[0], 96, 128, int[10], 2, true)

The insertion of the memory item m_6 into the memory specification invalidates the memory item m_5, so that now m_5 is configured as follows:

$m_5 =$ (4, 4, &aG[0], 32, 320, int[10], aG$_0$, true)

Furthermore, the insertion of the memory item m_6 triggers the construction of the new memory items m_7 and m_8 corresponding to the remains of the memory area of the memory item m_5 not overlapping with the memory area of m_6:

$m_7 =$ (5, ∞, &aG[0], 32, 96, int[10], aG$_0$, true)
$m_8 =$ (5, ∞, &aG[0], 128, 320, int[10], aG$_0$, true)

7 D_1714 = aG[x];

$m_9 =$ (6, ∞, &D_1714, 0, 32, unsigned int, aG$_6$[x$_6$], true)

The insertion of the memory item m_9 into the memory specification invalidates the memory item m_3:

$m_3 =$ (3, 5, &D_1714, 0, 32, unsigned int, Undef, true)

The next line of the example consists of an `if` statement `if(D_1714 == 2)`. This means that the evaluation of the guard condition `(D_1714 == 2)` is necessary. Before we start with the resolution algorithm, we summarize the current memory specification:

$m_1 =$ (1, 3, &aG[0], 0, 320, int[10], aG$_0$, true)
$m_2 =$ (2, ∞, &x, 0, 32, unsigned int, x$_0$, true)
$m_3 =$ (3, 5, &D_1714, 0, 32, unsigned int, Undef, true)
$m_4 =$ (4, ∞, &aG[0], 0, 32, int[10], 1, true)
$m_5 =$ (4, 4, &aG[0], 32, 320, int[10], aG$_0$, true)
$m_6 =$ (5, ∞, &aG[0], 96, 128, int[10], 2, true)

$$\begin{array}{ll} m_7 = & (5, \infty, \texttt{\&aG[0], 32, 96, int[10], aG}_0\texttt{, true}) \\ m_8 = & (5, \infty, \texttt{\&aG[0], 128, 320, int[10], aG}_0\texttt{, true}) \\ m_9 = & (6, \infty, \texttt{\&D\_1714, 0, 32, unsigned int, aG}_6\texttt{[x}_6\texttt{], true}) \end{array}$$

Now we process as defined by the function *resolveConstraint*() (Algorithm 11):

1. Initialize the path constraint according to the guard condition:

$$\Phi = (\texttt{D\_1714}_6\texttt{==}2).$$

2. Resolve `D_1714`$_6$: find the memory item responsible for `D_1714`$_6$, this is m_9. Resolve `D_1714`$_6$ according to the value of the item found:

$$\texttt{D\_1714}_6 \texttt{==aG}_6\texttt{[x}_6\texttt{]}.$$

Now the algorithm *resolveArrayExp*() (see Algorithm 28) is invoked with `D_1714`$_6$ as *var*, `aG`$_6$`[x`$_6$`]` as *val*, Φ as c and our memory configuration as *mem*.

First the values of the auxiliary variables *offsetStart* and *offsetEnd* are calculated:

$$\begin{array}{lcl} offsetStart & = & 32 \cdot x_6 \\ offsetEnd & = & 32 \cdot x_6 + 32 \end{array}$$

Then the loop iterates over all memory items with the matching base address. These are the following: m_8, m_7, m_6, m_5, m_4 and m_1. The validity period of the memory items m_5 and m_1 does not match the validity period of `aG`$_6$`[x`$_6$`]`. Thus, only m_8, m_7, m_6 and m_4 remain. We analyze them one by one:

- m_8: first the overlapping constraint is built

$$overlap = (128 < 32 \cdot \texttt{x}_6 + 32) \wedge (320 > 32 \cdot \texttt{x}_6)$$

Since the value m_8 refers to an input array, the array expression is built:

$$arrayExp = (\texttt{D\_1714}_6 \texttt{==rttArrayRead(aG}_0\texttt{, x}_6\texttt{))}$$

The array expression and overlapping constraint are summarized:

$$c_1 = (\texttt{D\_1714}_6 \texttt{==rttArrayRead(aG}_0\texttt{, x}_6\texttt{))} \wedge \\ (128 < 32 \cdot \texttt{x}_6 + 32) \wedge (320 > 32 \cdot \texttt{x}_6)$$

- m_7: first the overlapping constraint is built

$$overlap = (32 < 32 \cdot \texttt{x}_6 + 32) \wedge (96 > 32 \cdot \texttt{x}_6)$$

Since the value m_7 refers to an input array, the array expression is built:

$$arrayExp = (\texttt{D\_1714}_6 \texttt{==rttArrayRead(aG}_0\texttt{, x}_6\texttt{))}$$

The array expression and overlapping constraint are summarized:

$$c_1 = (\texttt{D\_1714}_6 \texttt{==rttArrayRead(aG}_0\texttt{, x}_6\texttt{))} \wedge (32 < 32 \cdot \texttt{x}_6 + 32) \wedge (96 > 32 \cdot \texttt{x}_6)$$

- m_6: first the overlapping constraint is built

  $$overlap = (96 < 32\cdot\mathtt{x}_6 + 32) \wedge (128 > 32\cdot\mathtt{x}_6)$$

  Since the value m_8 does not refer to an input, the procedure *resolveExp*() (Algorithm 12) is invoked. It produces the following resolution:

  $$c_1 = (\mathtt{D\_1714}_6 == 2) \wedge (96 < 32\cdot\mathtt{x}_6 + 32) \wedge (128 > 32\cdot\mathtt{x}_6)$$

- m_4: analog to the memory item m_6 results in

  $$c_1 = (\mathtt{D\_1714}_6 == 1) \wedge (0 < 32\cdot\mathtt{x}_6 + 32) \wedge (32 > 32\cdot\mathtt{x}_6)$$

Now we summarize all possible resolutions of `aG`$_6$`[x`$_6$`]` and thus the resolution results in:

$$\begin{gathered}\Phi = (\mathtt{D\_1714}_6{==}2) \wedge \\ ((\mathtt{D\_1714}_6 ==\mathtt{rttArrayRead(aG}_0\mathtt{,\ x}_6\mathtt{))} \wedge (128 < 32\cdot\mathtt{x}_6 + 32) \wedge (320 > 32\cdot\mathtt{x}_6) \vee \\ (\mathtt{D\_1714}_6 ==\mathtt{rttArrayRead(aG}_0\mathtt{,\ x}_6\mathtt{))} \wedge (32 < 32\cdot\mathtt{x}_6 + 32) \wedge (96 > 32\cdot\mathtt{x}_6) \vee \\ (\mathtt{D\_1714}_6 == 2) \wedge (96 < 32\cdot\mathtt{x}_6 + 32) \wedge (128 > 32\cdot\mathtt{x}_6) \vee \\ (\mathtt{D\_1714}_6 == 1) \wedge (0 < 32\cdot\mathtt{x}_6 + 32) \wedge (32 > 32\cdot\mathtt{x}_6)).\end{gathered}$$

3. Resolve $\mathtt{x}_6$: find the memory item responsible for $\mathtt{x}_6$, this is m_2. Resolve $\mathtt{x}_6$ according to the value of the item found:

   $$\mathtt{x}_6 == \mathtt{x}_0.$$

   Add the resolution result to the path constraint:

   $$\begin{gathered}\Phi = (\mathtt{D\_1714}_6{==}2) \wedge \\ ((\mathtt{D\_1714}_6 ==\mathtt{rttArrayRead(aG}_0\mathtt{,\ x}_6\mathtt{))} \wedge (128 < 32\cdot\mathtt{x}_6 + 32) \wedge (320 > 32\cdot\mathtt{x}_6) \vee \\ (\mathtt{D\_1714}_6 ==\mathtt{rttArrayRead(aG}_0\mathtt{,\ x}_6\mathtt{))} \wedge (32 < 32\cdot\mathtt{x}_6 + 32) \wedge (96 > 32\cdot\mathtt{x}_6) \vee \\ (\mathtt{D\_1714}_6 == 2) \wedge (96 < 32\cdot\mathtt{x}_6 + 32) \wedge (128 > 32\cdot\mathtt{x}_6) \vee \\ (\mathtt{D\_1714}_6 == 1) \wedge (0 < 32\cdot\mathtt{x}_6 + 32) \wedge (32 > 32\cdot\mathtt{x}_6)) \wedge \\ (\mathtt{x}_6 == \mathtt{x}_0).\end{gathered}$$

4. No unresolved symbols exist anymore. Thus, the resolution process stops and the constructed path constraint is passed to the solver.

After we have sketched the principle of proceeding of the algorithm, we will analyze it in more detail. The resolution of an array assignment is performed by the procedure call

$$resolveArrayExp(var,\ val,\ c,\ mem)$$

Where

- *var* is a versioned variable identifier. It indicates the variable, that has an array access as a value.
- *val* is a versioned array expression.
- *c* is a constraint that holds the result of the resolution process.

```
input: $var$ − variable identifier which has an array expression as value
       $val$ − array expression that must be resolved
       $mem$ − current memory specification
inout: $c$ − feasibility constraint
procedure resolveArrayExp($var$, $val$, $c$, $mem$){

  $offsetStart = \omega(val)$
  $offsetEnd = \omega(val)$ + bitsize($val$);

  // find out corresponding segment
  $S = \sigma(\beta(val), mem)$;

  foreach $m = last(S)$ downto $head(S)${
    if ($m.v_0 \leq \upsilon(val) \wedge \upsilon(val) \leq m.v_1 \wedge m.a$ == $\beta(val)$ ){

      $overlap = (m.o < offsetEnd) \wedge (m.l > offsetStart)$;
      $c_1 = m.c \wedge overlap$;

      if($c_1$ is feasible){
        if($m.val$ is not an input array){
          resolveExp($var$, $m'.val$, $c_1$, $mem$);
        } else {
          $arrayExp = (var ==$rttArrayRead($m.val$, getIdx($val$)));
          $c_1 = c_1 \wedge arrayExp$;
        }
        $c_2 = c_2 \vee c_1$;
      }
    }
  }
  $c = c \wedge c_2$
}
```

Algorithm 28: Resolution of an array expression.

- *mem* is the current memory specification.

The Algorithm 28 shows the procedure for the resolution of an assignment of an array expression *resolveArrayExp()*. First the algorithm determines offset start and offset end of the memory area selected by the array expression. Then the algorithm finds all memory items referring to the array variable from the array expression *val*. By iterating over the found items the algorithm detects by which of them the validity period correlates with the version of the array access. As far as such an item is found, the overlapping of the memory segments corresponding to the memory item and to the array expression is examined. The overlapping condition is conjuncted with the feasibility constraint of the found memory item and is stored in constraint c_1. When overlapping occurs and the feasibility constraint is feasible (i.e. c_1 is feasible), the algorithm examines the value of the found memory item and, if the value does not refer to an input array, invokes the procedure *resolveExp*() (Algorithm 12). This procedure continues with the resolution process of the found value and stores the result in constraint c_1. Otherwise, if the value of the item refers to an input array, the array expression resolving the variable *var* to the element of array *m.val* at index corresponding to the index of the array expression *val* by invocation of `rttArrayRead()` is built. This array expression is added to the feasibility constraint c_1. The constraint c_2 holds the disjunction of all constructed c_1 constraints and thus all possible resolutions of the array expression *val*.

At the end of the algorithm, after all memory items with matching base address were explored, the resulting constraint c_1 is added to the constraint c, the overall outcome of the resolution process.

Now we return to the presented example and observe the test driver that was generated for the built path constraint Φ. The solver has determined Φ as feasible and returned the following solution (here we omit the listing of the calculated values for local and auxiliary variables since they do not affect the generated test case):

```
aG[1] = 2
x = 1
```

Based on the calculated solution, the generator constructs the following test driver:

```
extern unsigned int aG[10];
unsigned int x;
int __rtt_return;
@rttBeginTestStep;
{
  x = 1;
  aG[1] = 2
  @rttCall(__rtt_return = example(x));
}
@rttEndTestStep;
```

As was already mentioned, CTGEN generates tests in RT-Tester syntax [44]. The values of parameter `x` and the global array `aG[]` are set according to the solution. These settings satisfy the guard condition `(aG[x] == 2)` and, consequently, this test will cover the intended branch. The complete test script as well as the other outputs produced by the generator for the discussed example are listed in Appendix 13.

5.12 Function Calls

There is a number of approaches to deal with function calls in symbolic execution and concolic testing. For defined functions the most common and straight forward approach is the inlining of the body of the called function [26, 54]. Yet this method increases the number of paths to be investigated rapidly. For the sake of scalability CUTE [93] suggests to concretize the return values of the function, but, since in this way the return value of the function is fixed, it can be impossible to cover some following branches although they are coverable. To deal with the problem of scalability when performing bottom-up unit testing Pathcrawler [18] performs analysis only of all feasible paths of the called function. This information is available since the called function was investigated before the calling function.

In this dissertation we use inlining to process defined function calls. Nevertheless, since many of the paths introduced by the called function cover the same branches in the calling function, we do not aim to achieve 100% coverage of the inlined function. Instead we search only for a path through this function which can lead to new covered branches. And, if the called function is too complex, there is still a possibility to switch to the approach introduced for processing undefined functions.

For the handling of functions whose code is not available, a number of approaches exist. The most common one is to substitute symbolic variables with concrete values [47, 93, 23]. Another approach is used by PathFinder [85] – *mixed concrete-symbolic solving*, where the external function is not interpreted first and is concretized later with consideration of solvable constraints in the path condition, but through concretization the generation process falls back to the random testing, with all its limitations. KLEE [22] uses a modeled library, which allows interactions with the environment. However, the implementation of such a library is time consuming and this method still has limitations in case when there is no implemented model for the invoked external function. Pathcrawler suggests another approach [18] - formal specification of the called function by the use of pre- and postconditions. In the analysis of the caller function the external function is replaced by the constraint, specifying its behavior. However, this approach requires manual intervention.

In this dissertation we propose the automatic generation of mock objects for the handling of external function calls. These mock objects replace external functions by test stubs with the same signature. The return data, output parameters and global variables which can be modified by the stubbed function are set according to the calculated values in order to fulfill a path condition. This also allows to simulate exceptional behavior of stubbed functions, which often is not easy to stimulate in practice. It is possible but not required to customize stub behavior by using the annotation language.

In the following sections we discuss approaches used in this dissertation for handling defined and undefined function calls in more detail.

5.12.1 Processing defined Function Calls

To handle defined function calls we apply inlining. Before STCT initialization the CFG of the function under investigation is examined and all nodes containing function calls are replaced by the CFGs of the corresponding functions. Afterwards, the test generation process is undertaken as usual with a difference, that the objective is not to cover all branches of the resulting CFG but only the branches of the original one.

We illustrate our approach by the following example:

```
int foo(int a, int b)
{
    switch(a){
    case 0:
        return b;
    case 1:
        return -b;
    case 2:
        return 0;
    default:
        return -1;
    }
    return -1;
}

int test(int a, int b)
{
    if(foo(a, b) > 0){
        return 1;
    }
    return 0;
}
```

Suppose, our goal is to generate 100% branch coverage for the function `test()`. The CFG of the function `test()` in three address code is shown in Figure 5.4 on the left. The node holding a call to function `foo()` is drawn gray. Before the analysis for test generation starts, this node is replaced by its CFG – the result is shown in Figure 5.4 on the right. The part of the CFG corresponding to the function `foo()` is drawn gray.

The generation process produces two test cases corresponding to traces (for simplicity we list here only nodes of the traces)

- $<$ (a$_{foo}$ =a), (b$_{foo}$ =b), (D_1725= -1), (return$_{foo}$ = D_1725), (D_1719 = return$_{foo}$), (retval.0 = (D_1719 > 0)), (D_1722 = 1), (return = D_1722) $>$
- $<$ (a$_{foo}$ =a), (b$_{foo}$ =b), (D_1725=b$_{foo}$), (return$_{foo}$ = D_1725), (D_1719 = return$_{foo}$), (retval.0 = (D_1719 > 0)), (D_1722 = 1), (return = D_1722) $>$

The nodes that belong to these traces are drawn blue in Figure 5.5. The nodes (D_1725= $-$b$_{foo}$) and (D_1725= 0) are still uncovered, but their coverage is not necessary, since all branches of the CFG corresponding to the function `test()` where examined by these two traces. The test script and all the other outputs produced by the generator for this example can be observed in Appendix 14.

5.12.2 Processing undefined Function Calls

When an external function call appears on the path under consideration, the return value of this external function, its output parameters and all global variables allowed for modification are handled as symbolic *stub variables*. These symbolic stub variables can possibly be modified by this call. A stub variable holds the information about the stub function to which it belongs and – if it corresponds to the return value – the output parameter or global variable, changed by this stub.

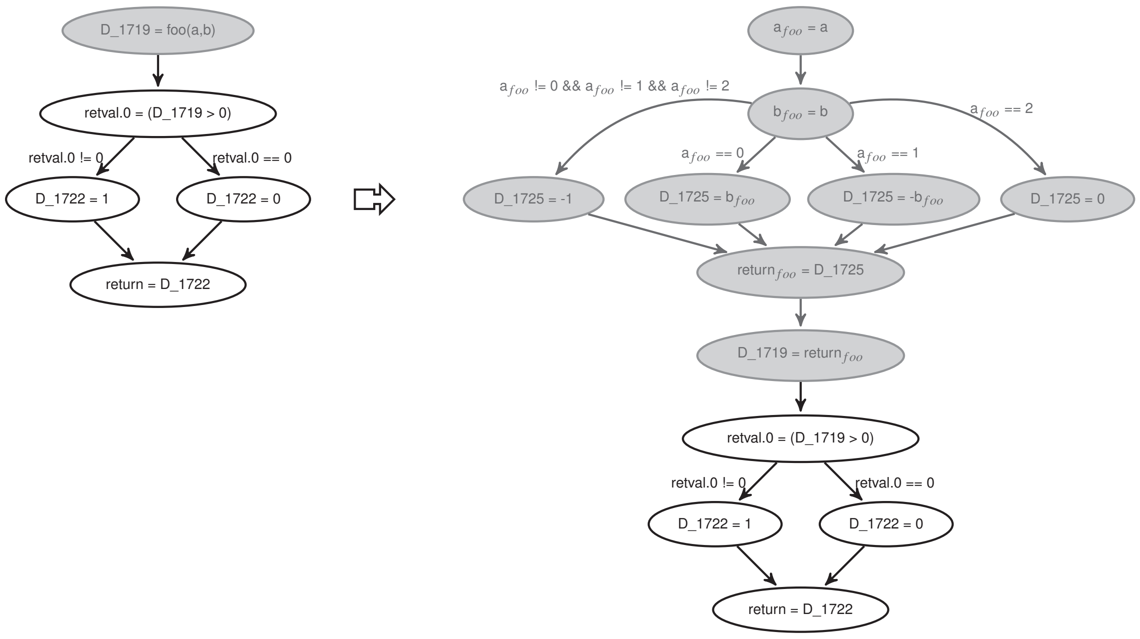


Figure 5.4: Processing defined functions: inlining.

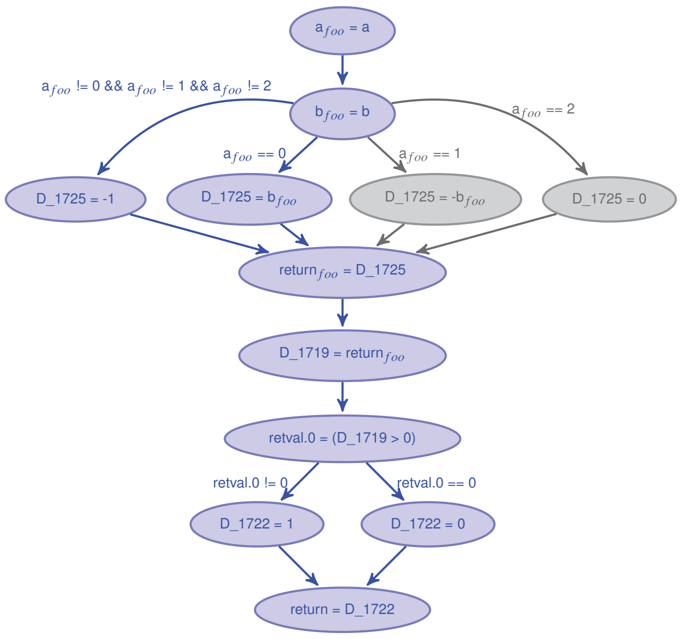


Figure 5.5: Processing defined functions: selection.

Definition 5.11. *Stub variable sv* is defined as the following structure:

$$sv \quad =_{def} \quad \boxed{sv.name \mid sv.type \mid sv.stubType \mid sv.func \mid sv.parameter}$$

where

- *sv.name* is the name of the stub variable,
- *sv.type* is the type of the stub variable (integer, char etc),
- *sv.stubType* is the stub type of the stub variable – it indicates whether this variable corresponds to a global variable (type is *stubGlobal*), return value of the stub function (type is *stubReturn*) or its output parameter (type is *stubOutputParameter*),
- *sv.func* is the name of the corresponding stub function,
- *sv.parameter* is the name of the corresponding parameter in case if the type is *stubOutputParameter* and empty otherwise.

First we introduce some auxiliary functions that we need for the definition of the algorithm for handling undefined function calls:

$\varphi : Expression \rightarrow SymbolTable$	Maps an expression to the corresponding symbol table entry – variable or function according to the expression.
$\phi : Expression \rightarrow SymbolTable$	Maps a parameter from a function expression to the corresponding signature entry containing symbol table information.
$\eta : Expression \rightarrow \mathbb{N}$	Returns the stub counter for the given function, i.e. how many times this function was already called on the path under investigation.

Definition 5.12. The effect of the *assignment of an undefined function call to a variable*

```
v = func(...);
```

on the state space S_s is specified by the procedure call:

$$updateByFctAssignment(sel,\ func,\ n,\ mem); \quad mem' = mem;$$

where

- *sel* is a selector of the variable identifier,
- *func* is an undefined function expression,
- *n* is the current computational step,
- *mem* is the current memory specification.

```
input: $sel$ – selector of the identifier
       $func$ – function expression that should be assigned to the identifier $sel$
       $n$ – current computational step
inout: $mem$ – current memory specification

procedure updateByFctAssignment($sel$, $func$, $n$, $mem$){

  handleOutputParameters($func$, $n$, $mem$);
  handleGlobals($func$, $n$, $mem$);

  // create a stub variable corresponding to the return value of the function
  sv.name = $\varphi(func)$.name + ‘‘@RETURN’’;
  sv.type = $\varphi(func)$.type;
  sv.stubType = stubReturn;
  sv.func = $\varphi(func)$.name;

  // set the version of the stub variable expression corresponding to
  // the stub counter
  $\upsilon(sv)$ = $\eta(func)$;

  updateByAssignment($sel$, $sv$, $n$, $mem$);
}
```

Algorithm 29: Effect of the assignment of an undefined function call on the memory specification.

Assignment of an undefined function call may affect the stack or data segment dependent on the signature of the function *func* and existence of the global variables in the module under test configuration.

Algorithm 29 shows the procedure *updateByFctAssignment*(). This procedure specifies the algorithm for processing undefined function calls. First, symbolic stub variables for output parameters of the observed function are created by the auxiliary procedure *handleOutputParameters*(). Next, symbolic stub variables for all global variables, which can possibly be changed by the observed function call, are created by the auxiliary procedure *handleGlobals*(). And, finally, a stub variable corresponding to the return value of the observed function is created: its name is composed from the function name and `RETURN` identifier, the type is set corresponding to the return type of the function, the stub type is set to *stubReturn* to characterize this stub variable as corresponding to the return value and, last, the name of the function, to which this stub variable refers, is set. The version of the expression with this stub variable is set according to the stub counter of the called function and an effect on the assignment of the created stub variable to the left-hand side of the function assignment expression is computed. The version of the stub variable expression is important here, since it is used by the further test driver generation to identify by which stub call which values for the stub variables must be set.

To illustrate this procedure consider the simple example: the following function call is performed on the path under investigation: `i = func_ext()`, where the signature of the called function is `int func_ext()`. In this case a new stub variable `func_ext@RETURN` of type `int` is created. Its stub type is set to *stubReturn* and the corresponding stub function is set to `func_ext`. Suppose, this is the first call to function `func_ext()`, hence the version of the corresponding expression is set to zero. Finally, the effect of an assignment $\texttt{i = func\_ext@RETURN}_0$ on the memory specification is computed.

```
input: func − function expression whose output parameters should be analyzed
       n − current computational step
inout: mem − current memory specification

procedure handleOutputParameters(func, n, mem){

  foreach(parameter expression p in func.parameters){
    sp = $\phi(p)$;

    if(sp.type is a pointer && sp.refType is not const){

      // create a stub variable corresponding to the output parameter
      // of the function
      sv.name = sp.name + ``@'' + $\varphi(func)$.name + ``@outputParam'';
      sv.type = sp.refType;
      sv.stubType = stubOutputParameter;
      sv.func = $\varphi(func)$.name;
      // set the version of the stub variable expression corresponding to
      // the stub counter
      $\upsilon(sv) = \eta(func)$;

      if(p contains address operation){
        v = variable participating in expression p
        updateByAssignment(v, sv, n, mem);
      } else {
        updateByAssignment(*p, sv, n, mem);
      }
    }
  }
}
```

Algorithm 30: Undefined function calls: handling of output parameters.

The auxiliary routine for the handling of the output parameters is shown in Algorithm 30. The procedure receives the function expression, the current computational step and the current memory specification as input. For each expression that was passed to the function call as a parameter, the following analysis is performed: if the corresponding function parameter is a pointer and its content is not protected by the `const` qualifier, a new stub variable is created and an effect of the assignment of this variable to the variable where the pointer parameter points to is computed. For example, the following function call is performed on the path under investigation: `func_ext(&a)` where the signature of the called function is `func_ext(int *p)`. In this case, the pointer does not point to a constant, so that a new stub variable `a@func_ext@outputParam` of type `int` is created. Its stub type is set to `stubOutputParameter` and the corresponding stub function is set to `func_ext`. Suppose, this is the first call to function `func_ext()`, hence the version of the corresponding expression is set to zero. Finally, the effect of an assignment `a = a@func_ext@outputParam`$_0$ on the memory specification is computed.

The auxiliary routine for the handling of the global variables is shown in Algorithm 31. The procedure receives the function expression, the current computational step and the current memory specification as input. For all memory items from the data segment following analysis is performed: if the memory item is still valid, it does not refer to a constant and its modification is allowed (for the specification of global variables whose modification is permitted see Chapter 3), this item is invalidated and a new stub variable is created. A new memory item, a copy of the invalidated item but with the difference that the value is set to the stub variable expression, is created.

Definition 5.13. The effect of an *undefined procedure call*

```
proc(...);
```

on the state space S_s is specified by the procedure call:

$$updateByProcedureCall(proc,\, n,\, mem); \quad mem' = mem;$$

where

- *proc* is a procedure expression,
- *n* is the current computational step,
- *mem* is the current memory specification.

An undefined procedure call may affect the stack or data segment dependent on the signature of the procedure *proc* and existence of the global variables in the module under test configuration.

Algorithm 32 shows procedure *updateByProcedureCall*(), which specifies the algorithm for processing undefined procedure calls. This procedure is a simpler version of the procedure *updateByFctAssignment()*, since a procedure has no return value. In that way merely the output parameters and global variables must be considered.

To illustrate how the algorithm for processing undefined function calls works, we use the following example:

```
input: func − function expression whose call is analyzed
       n − current computational step
inout: mem − current memory specification

procedure handleGlobals(func, n, mem){

  $S = \emptyset$;

  foreach m = head(dataSegment) upto last(dataSegment){

    if($m.v_1$ == $\infty$ &&
       m.t is not const &&
       $\varphi(m.a)$ is not prohibited){

      // invalidate found memory item
      $m.v_1 = n$;

      // create a stub variable corresponding to the global var
      sv.name = $\varphi(m.a)$.name + ``@'' + $\varphi(func)$.name;
      sv.type = $\varphi(m.a)$.type;
      sv.stubType = stubGlobal;
      sv.func = $\varphi(func)$.name;
      // set the version of the stub variable expression corresponding to
      // the stub counter
      $\upsilon(sv)$ = $\eta(func)$;

      // create a new memory item
      $m'.v_0 = n$;
      $m'.v_1 = \infty$;
      $m'.a = m.a$;
      $m'.t = m.t$;
      $m'.o = m.o$;
      $m'.l = m.l$;
      $m'.val = sv$;

      $S = S \cup \{m'\}$;
    }
  }
  dataSegment = dataSegment $\cup$ S;
}
```

Algorithm 31: Undefined function calls: handling of global variables.

```
input: proc − procedure expression that should be assigned to the identifier sel
       n − current computational step
inout: mem − current memory specification

procedure updateByProcedureCall(proc, n, mem){

  handleOutputParameters(proc, n, mem);
  handleGlobals(proc, n, mem);

}
```

Algorithm 32: Effect of the assignment of an undefined procedure call on the memory specification.

```
extern int func_ext(int a);
int globalVar;

void test(int p1, int p2){
  __rtt_modifies(globalVar);

  globalVar = -p2;
  if(func_ext(p1) > p2){
    if(func_ext(p2) == p1 && globalVar == p2){
      ERROR;
    }
  }
}
```

In the procedure `test()` the external function `func_ext()` is called twice. To reach the line with an error, `func_ext()` must return a value that is greater than the value of the parameter `p2` by the first call. Furthermore, by the second call it must return a value that is equal to the value of the parameter `p1`. The symbolic interpreter analyzes what could possibly be altered by `func_ext()` and creates the stub variables `func_ext@RETURN` and `globalVar@func_ext`. The constraint generator generates the following path constraint (the path constraint is listed here in a simplified form for better understanding; the complete path constraint for this example is listed in Appendix 15):

```
func_ext@RETURN@0 > p2 &&
func_ext@RETURN@1 == p1 &&
globalVar@func_ext@1 == p2
```

The occurrences of the stub variables are versioned corresponding to the running number of the calls of the external function. Here `func_ext@RETURN@0` corresponds to the return value of the first call and `func_ext@RETURN@1` to the return value of the second one. The solver determines the path constraint as feasible and computes the following solution:

```
func_ext@RETURN@0 = 2147483647
func_ext@RETURN@1 = 0
globalVar@func_ext@1 = -1
p1 = 0
p2 = -1
```

Now consider the generated test driver:

```
extern unsigned int func_ext_STUB_testCaseNr;
extern unsigned int func_ext_STUB_retID;
extern int func_ext_STUB_retVal[2];

int p1, p2;

/***** STUB func_ext *****/
func_ext_STUB_testCaseNr = 0;
func_ext_STUB_retID = 0;

/* set values for return */
func_ext_STUB_retVal[0] = 2147483647;
func_ext_STUB_retVal[1] = 0;
/***** end STUB func_ext *****/

p1 = 0;
p2 = −1;
@rttCall(test(p1, p2));
```

and the generated stub:

```
int func_ext(int a){
  @GLOBAL:
    unsigned int func_ext_STUB_testCaseNr;
    unsigned int func_ext_STUB_retID;
    int func_ext_STUB_retVal[2];
  @BODY:
    func_ext_RETURN =
      func_ext_STUB_retVal[func_ext_STUB_retID%2];
    if(func_ext_STUB_testCaseNr == 0){
      if(func_ext_STUB_retID == 1){
        globalVar = −1;
      }
    }
    func_ext_STUB_retID++;
};
```

CTGEN generates tests in RT-Tester syntax. An array `func_ext_STUB_retVal` of size two (corresponding to the number of calls of the function `func_ext()`) is created to hold the calculated return values. These values are stored by the test driver according to their version. The variable `func_ext_STUB_retID` corresponds to the running number of the stub call. It is reset by the test driver before each call of the UUT and incremented by the corresponding stub each time it is called. Since one test driver can hold many test cases, the variable `func_ext_STUB_testCaseNr`, that corresponds to the number of the test case, is created. This variable is set by the test driver. The value of the global variable `globalVar` is set by the stub if the number of the stub call and the test case number match the calculated ones for this global variable.

The complete test script, stub and all other outputs produced by the generator for this example are listed in Appendix 15.

5.12.3 Processing undefined Function Calls with Stub Specification

In the previous section we presented how we process undefined function calls. However, the behavior of a stub function generated this way can deviate from the behavior of the real function. To approach this problem, we introduce a specification of stubs by means of the annotation language presented in Chapter 3. It is possible to define the range of stub parameters, global variables as well as return values over pre- and postconditions.

To illustrate this technique, we consider the example from the previous section:

```
extern int func_ext(int a);
int globalVar;

void test(int p1, int p2){
  __rtt_modifies(globalVar);

  globalVar = -p2;
  if(func_ext(p1) > p2){
    if(func_ext(p2) == p1 && globalVar == p2){
      ERROR;
    }
  }
}
```

The solver computed the following solution to reach the line with an error:

```
func_ext@RETURN@0 = 2147483647
func_ext@RETURN@1 = 0
globalVar@func_ext@1 = -1
p1 = 0
p2 = -1
```

Suppose, the return value of the real function `func_ext()` cannot be greater or equal to 20 and that the `globalVar` is modified by `func_ext()` in such a way, that afterwards the value of the variable `globalVar` is always greater than 17. In this case the first calculated return value for the stub (2147483647) as well as the calculated value for modification of the variable `globalVar` (-1) are impossible. To take this fact into account, we expand the code under test with a dummy function which has the same signature as `func_ext()`:

```
int func_ext(int a)
{
    __rtt_extern();
    __rtt_postcondition(__rtt_return < 20 && globalVar > 17);

    return 0;

}
```

This dummy function contains:

1. The auxiliary function `__rtt_extern()` which identifies the defined function as a stub specification for an external function call.

2. A postcondition which specifies that return values of the corresponding generated stub must be

less than 20 and that after the execution of the stub the global variable `globalVar` must have a value greater than 17.

Suppose further, that the parameter `a` of the function `func_ext()` must always be greater than zero and the global variable `globalVar` is accepted only in range $(-20, 20)$. To take this into account, we expand the specification of the stub with a precondition:

```
int func_ext(int a)
{
    __rtt_extern();
    __rtt_precondition(a > 0 && -20 < globalVar && globalVar < 20);
    __rtt_postcondition(__rtt_return < 20 && globalVar > 17);

    return 0;

}
```

To process specified pre- and postconditions, we apply inlining like in the case of processing defined functions. The difference is, that to activate the stub generation the call to the function is not replaced by its body but remains in the CFG. The nodes corresponding to the precondition definition are inserted before the function call and the nodes characterizing the postcondition – after. We illustrate this by our example. Figure 5.6 on the left shows the simplified CFG of the function under test `test()`, the call to the function `func_ext()` is drawn red. On the right side the CFG with inserted nodes responsible for parameter initialization and precondition (nodes before the call to `func_ext()`) and nodes defining the postcondition (nodes after the call to `func_ext()`) is shown. The inserted nodes are drawn gray. This preprocessing ensures, that the generator will consider only values in specified range when calculating the values for stub variables and that in case when such variable assignment cannot be found, it reports the corresponding branches as unreachable.

Now the path constraint constructed by the generator to reach the line with an error is as follows:

```
func_ext_int_parametername_a@v0 > 0 &&
func_ext_int_parametername_a@v0 == p1 &&
globalVar@v0 == -p2@0 &&
globalVar@v0 > -20 &&
globalVar@v0 <  20 &&
func_ext@RETURN@0 <= 19 &&
globalVar@func_ext@0 > 17 &&
func_ext@RETURN@0 > p2 &&
func_ext_int_parametername_a@v1 > 0 &&
func_ext_int_parametername_a@v1 == p2 &&
globalVar@func_ext@0 > -20 &&
globalVar@func_ext@0 <  20 &&
func_ext@RETURN@1 <= 19 &&
globalVar@func_ext@1 > 17 &&
func_ext@RETURN@1 == p1 &&
globalVar@func_ext@1 == p2
```

As in the example from the previous section, this path constraint is simplified for better understanding, `v0` and `v1` denote here the versions of the variables. The complete version of the path constraint as well as the complete test driver, stub code and other generator outputs for this example are listed in Appendix 16. This path constraint is more complicated than the path constraint from the example without

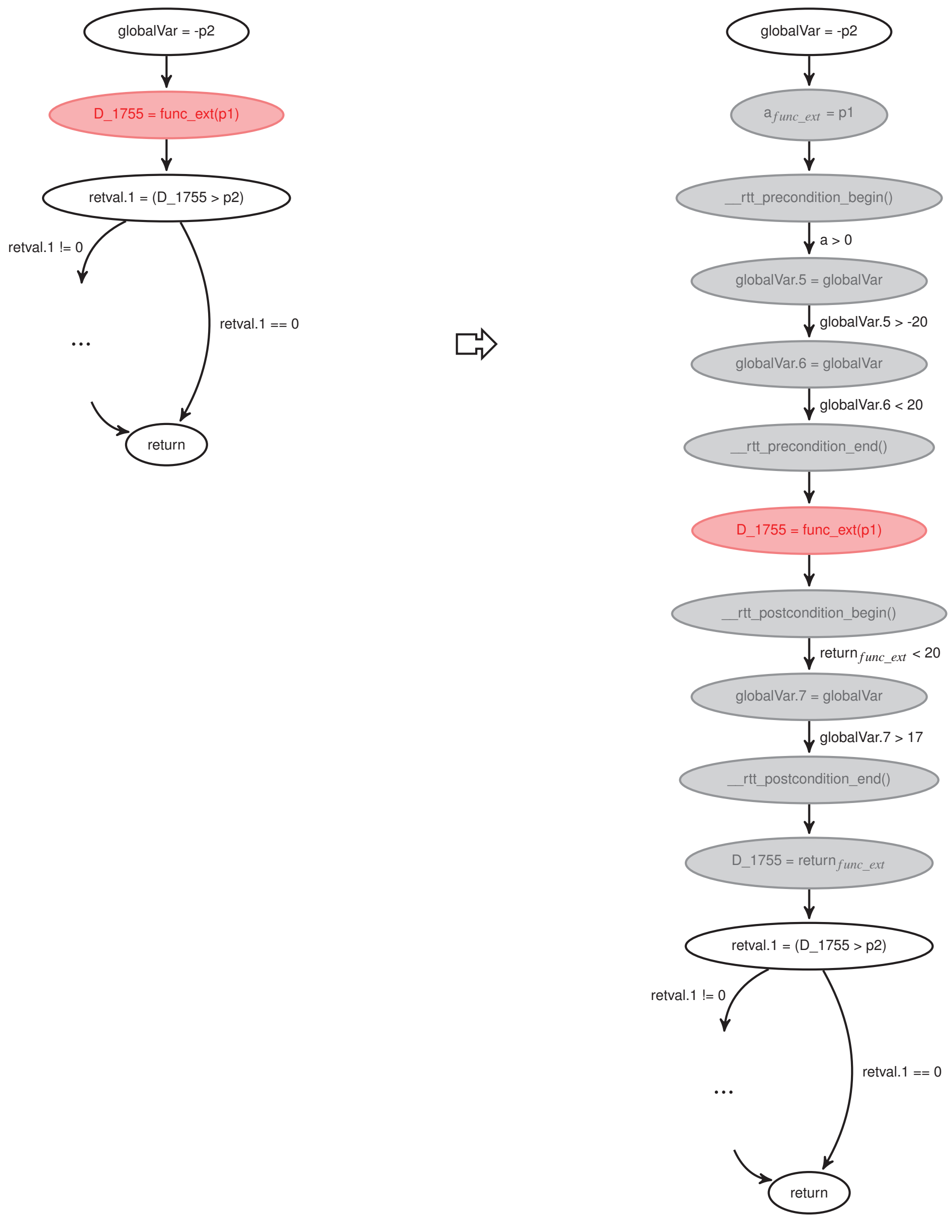


Figure 5.6: Processing specified stubs.

stub specification. To emphasize this, we highlighted the inequalities which participate in both path constraints. The part of the path constraint in lines 1 - 5 corresponds to the precondition of the first call to `func_ext()`, inequalities in lines 6-7 correlate with postcondition of the first call to `func_ext()` and the inequality in line 8 results from the guard condition in the function under test in line 8. Similarly, the lines 9 - 12 of the path constraint correspond to the precondition of the second call to `func_ext()`, lines 13-14 correlate with its postcondition and inequalities in lines 15-16 are caused by the guard conditions in the function under test in line 9.

The solver determines the path constraint as feasible and computes the following solution:

```
func_ext@RETURN@0 = 19
func_ext@RETURN@1 = 16
globalVar@func_ext@0 = 19
globalVar@func_ext@1 = 18
p1 = 16
p2 = 18
```

Note, that the analysis of inequalities in lines 6, 8, 14 and 16 from the path constraint:

```
func_ext@RETURN@0 <= 19 &&
func_ext@RETURN@0 > p2 &&
globalVar@func_ext@1 > 17 &&
globalVar@func_ext@1 == p2
```

shows, that there is only one possible combination of assignment of the parameter `p2` and the return value of the function `func_ext()` by the first call: `p2 = 18` and `func_ext@RETURN@0 = 19`, which makes it very unlikely that random testing will be capable of uncovering the error in this code.

5.13 Symbolic Execution of an Expression

After we have discussed the algorithms handling the symbolic execution of different specific forms of expressions, we define a procedure *executeExpression*() (Algorithm 33) which takes an expression *exp*, which must be executed symbolically, and a current memory specification *mem* as inputs. The expression *exp* is analyzed and is delegated further to the algorithms, handling the specific form of an expression according to its structure. So, if the passed expression does not have any left-hand side, this is a procedure call and must be handled by the algorithm *updateByProcedureCall*() correspondingly. If the expression on the left is a dereferenced pointer, the symbolic execution is handled by the procedure *updateByAssignmentToDerefPtr()* (Algorithm 13). Further, the right-hand side of the expression is analyzed and passed to the procedure *updateByBitFieldAssignment()* (Algorithm 22) if it represents a bit field, or to the procedure *updateByFctAssignment*() (Algorithm 29) if it is a function call. All other cases are handled by the algorithm *updateByAssignment*() (Algorithm 9).

5.14 Auxiliary Functions

In this section we give an overview over all auxiliary functions introduced for symbolic execution algorithms in this chapter.

$\beta : Selectors \rightarrow BaseAddress$	Maps a selector to the corresponding base address.
$\tau : Selectors \rightarrow Symbols$	Maps a selector to the type of the corresponding variable.
$\omega : Selectors \rightarrow Expression$	Maps a selector to the corresponding symbolic offset expression.
`bitsizeof`: $Selectors \rightarrow \mathbb{N}$	Maps a selector to the length of the selected memory in bits.
`basetype`: $Selectors \rightarrow Symbols$	Maps an array or a pointer selector to the base type. For example, for the variable p of type `int**`, `basetype`(p) = `int`.
$\sigma : BaseAddress \times M \rightarrow M - Item^*$	Maps a base address to the stack, heap or global data according to the current memory configuration.
$\upsilon : Expression \rightarrow \mathbb{N}$	Returns a version of the given expression.
$\alpha : Expression \rightarrow Expression$	Returns the operand with the address operation. For example, for expression e = `&a + b`, $\alpha(e)$ =`&a`.
$\delta : Expression \rightarrow Expression$	Returns the operand with the offset part. For example, for expression e = `&a + b`, $\delta(e)$ =`b`. If the offset part is not existent, $\delta(e) = 0$.
$\nu : Selectors \rightarrow Selectors$	Maps a selector to the corresponding base name. For example $\nu(x.f1.f2) = x$.
$\chi : Expression \times Expression \rightarrow Selectors$	Maps the defined memory area within a structure to the corresponding selector.
$\iota : Expression \times Expression \rightarrow Symbols$	Maps the defined memory area within a structure to the corresponding type.
$\varphi : Expression \rightarrow SymbolTable$	Maps an expression to the corresponding symbol table entry – variable or function according to the expression.
$\phi : Expression \rightarrow SymbolTable$	Maps a parameter from a function expression to the corresponding signature entry containing the symbol table information.
$\eta : Expression \rightarrow \mathbb{N}$	Returns the stub counter for the given function, i.e. how many times this function was already called on the path under investigation.

```
input:  exp – expression that must be executed symbolically
inout:  mem – current memory specification
procedure executeExpression(exp, mem){
  left = left side of exp;
  right = right side of exp;
  n = current computational step;
  if(!left){
    updateByProcedureCall(right, n, mem);
  } else if(left is a dereferenced pointer){
    updateByAssignmentToDerefPtr(left, right, n, mem);
  } else if(right is a bit field) {
    updateByBitFieldAssignment(left, right, n, mem);
  } else if(right is a function call) {
    updateByFctAssignment(left, right, n, mem);
  } else {
    updateByAssignment(left, right, n, mem);
  }
}
```

Algorithm 33: Symbolic Execution of an Expression.

6 Experimental Results and Evaluation

This chapter is an extended version of the experimental results and the evaluation published in [72].

The experimental evaluation of CTGEN and the comparison with competing tools was performed both with synthetic examples evaluating the respective tools' specific capabilities and with embedded systems code from an industrial automotive application. The latter presented specific challenges: (1) the code was automatically generated from Simulink models. This made automated testing mandatory since small model changes considerably affected the structure of the generated code, so that the re-use of existing unit tests was impossible if the models had been changed. (2) Some units were exceptionally long because insufficient hardware resources required to reduce the amount of function calls.

Table 6.1 shows the results achieved by CTGEN in the automotive test project on some selected functions. The most challenging function was f_1 with over 2000 lines of code (714 executable lines), using structures, bit vectors, pointer parameters and complex branch conditions. Nevertheless, CTGEN was able to generate 95,1% line and 89,0% branch coverage with 59 automatically generated test cases. Furthermore, by using preconditions as guides for CTGEN to cover parts of code further down in the CFG, it was possible to increase the coverage even more. Function f_2 with 50 executable lines of code (about 300 lines of code) represents a typical function in the project. For such functions CTGEN achieved 100% C1 coverage. Function f_3 includes pointer comparison, pointer dereferencing and a `for`-loop with an input parameter as a limit. However, due to the small branching factor CTGEN achieves 100% coverage with only 3 test cases and a generation time of under one second. Summarizing, CTGEN proved to be efficient for industrial test campaigns in the embedded domain and considerably reduced the overall project efforts. The more detailed report to this industrial study is given in Appendix 7.4.

In comparison (see Tables 6.2, 6.3 and 6.4), experiments with KLEE [22] and PathCrawler [18] demonstrated that CTGEN delivers competitive results and outperformed the others for the most complex function $f_1()$. The experiments with PathCrawler were made with the online version [2], so it was not possible to exactly measure the time spent by this tool. This tool, however, could not handle the complexity of $f_1()$, whereas KLEE did not achieve as much coverage as CTGEN, we assume that this is due to the path-coverage oriented search strategy, which has not been optimized for achieving C1 coverage.

Functions $f_4()$ and $f_5()$ are also taken from the automotive testing project. Function $f_5()$ has struct-inputs with bit fields. KLEE achieved 100% path coverage. PathCrawler also targets path coverage but due to limitations of the online version (number of generated test cases, available amount of memory) delivers only 201 test cases. However, we assume that without these limitations it will also achieve 100% path coverage although in a larger amount of time than KLEE. For the example function *Tritype()* KLEE delivers poor results because it does not support floating types. There is, however, an extension KLEE-FP [30] targeting this problem. PathCrawler excels CTGEN and KLEE but can handle only the `double` type, not `float`, while CTGEN can calculate bit-precise solutions for both. $alloc\_ptr()$ and $comp\_ptr()$ demonstrate handling of symbolic pointers, which is not supported by PathCrawler; KLEE and CTGEN deliver comparable results. Functions $test\_sym1()$ and $test\_sym2()$ demonstrate handling

	Executable Lines	Branches	Time	Nr of Test Cases	Lines Coverage	Branch Coverage
$f_1()$	714	492	31m27.098s	59	95,1%	89,0%
$f_2()$	50	30	0m1.444s	8	100%	100%
$f_3()$	11	4	0m0.228s	3	100%	100%

Table 6.1: Experimental results on some functions of HELLA software.

	CTGEN	KLEE		PathCrawler
$f_1()$ (714 lines, 492 branches)				
Time	31m27.098s	16m50s	586m16.590s	-
Nr of Test Cases	59	1120	24311	-
Lines Coverage	95,1%	77,9%	78,54%	-
Branch Coverage	89,0%	58,2%	59,36%	-
$f_4()$ (19 lines, 4 branches)				
Time	0.062s	0.040s		< 1s
Nr of Test Cases	3	3		9
Lines Coverage	100%	100%		100%
Branch Coverage	100%	100%		100%
$f_5()$ (28 lines, 35 branches)				
Time	0.337s	3.234s	41,176s	10s
Nr of Test Cases	3	463	2187	201
Lines Coverage	100%	100%	100%	85,71%
Branch Coverage	100%	100%	100%	85,71%

Table 6.2: Experimental results compared with other tools on some functions of HELLA software.

of unions, which is not supported by PathCrawler. CTGEN and KLEE achieved 100% branch and line coverage in comparable time.

Aliasing problems were investigated by example of the function *input_array*() (Table 6.4). CTGEN generated two test cases, which achieved 100% line and branch coverage. KLEE could determine the out of bound pointer in the guard condition of the `if` statement. However, as we have already pointed out, CTGEN does not aim to detect such problems, because these are often more successfully investigated by means of formal verification, static analysis or abstract interpretation. PathCrawler has aborted the test generation after getting a segmentation fault in the tested function. Nevertheless, after the indices `x` and `y` were bounded by an `if` statement, PathCrawler was also able to generate test cases that achieved 100% line and branch coverage.

	CTGEN	KLEE	PathCrawler
Tritype()			
Time	8.404	0.095	< 1s
Nr of Test Cases	8	1	11
Lines Coverage	100%	41,66%	100%
Branch Coverage	100%	20%	100%

```
int Tritype(double i, double j, double k){
  int trityp = 0;
  if (i < 0.0 || j < 0.0 || k < 0.0)
    return 3;
  if (i + j <= k || j + k <= i || k + i <= j)
    return 3;
  if (i == j) trityp = trityp + 1;
  if (i == k) trityp = trityp + 1;
  if (j == k) trityp = trityp + 1;
  if (trityp >= 2)
      trityp = 2;
  return trityp;
}
```

	CTGEN	KLEE	PathCrawler
alloc_ptr()			
Time	0.071s	0.064s	< 1s
Nr of Test Cases	4	4	2
Lines Coverage	100%	100%	42,86%
Branch Coverage	100%	100%	50%

```
char *alloc_ptr(char *allocbufp, char *allocp,
                unsigned int n)
{
    if(allocbufp == 0 || allocp == 0)
        return 0;

    if(allocbufp + ALLOCSIZE - allocp >= n){
        allocp += n;
        return allocp - n;
    }
    return 0;
}
```

	CTGEN	KLEE	PathCrawler
comp_ptr()			
Time	0.032s	0.055s	< 1s
Nr of Test Cases	4	4	2
Lines Coverage	100%	100%	75%
Branch Coverage	100%	100%	50%

```
int comp_ptr(char *p1, char *p2)
{
    if(p1 != NULL && p2 != NULL && p1 == p2){
        return 1;
    }
    return 0;
}
```

Table 6.3: Experimental results compared with other tools – floating point and pointer comparison.

	CTGEN	KLEE	PathCrawler
*test*1()			
Time	0.029s	0.033s	∼8s
Nr of Test Cases	3	3	1
Lines Coverage	100%	100%	80%
Branch Coverage	100%	100%	25%
*test*2()			
Time	0.027s	0.035s	∼9s
Nr of Test Cases	2	2	1
Lines Coverage	100%	100%	83%
Branch Coverage	100%	100%	50%

```
typedef union {
  unsigned short  c2u16;
  unsigned char c2u8[2];
} union_u16;

union_u16 globalV;
int test_sym1(unsigned short x)
{
  globalV.c2u16 = x;
  if(globalV.c2u8[0] == 0xff && globalV.c2u8[1] == 85){
      return 1;
  }
  return 0;
}

int test_sym2(unsigned char x, unsigned char y)
{
  globalV.c2u8[0] = x;
  globalV.c2u8[1] = y;

  if(globalV.c2u16 == 0x5555){
    return 1;
  }
  return 0;
}
```

	CTGEN	KLEE		PathCrawler
input_array()				
Time	0.044s	27.849s	14m16.702	-
Nr of Test Cases	2	108	676	-
Lines Coverage	100%	100%	100%	-
Branch Coverage	100%	100%	100%	-

```
#define N 2
typedef int my_array[N];
int input_array(my_array a, unsigned int x, unsigned int y)
{
  int retval = 0;
  if(a[x] > a[y]){
    retval = 1;
  } else {
    retval = 0;
  }
  return retval;
}
```

Table 6.4: Experimental results compared with other tools – unions and input arrays.

7 Conclusion

7.1 Summary

In this thesis a new method for automated verification of C functions has been presented. This method consists of two parts: the specification of the module under test and the actual verification.

To allow a formalized module specification, an annotation language as an extension of a pre- and postcondition syntax was developed and discussed. This annotation language allows the definition of logical conditions relating the program's prestate to its poststate after the module's execution. More complex correctness conditions, such as the number of subfunction calls performed by the UUT, may also be specified by means of pre- and postconditions if auxiliary variables are introduced, such as counters for the number of subfunction calls performed. Via the specification of pre- and postconditions test case generation for both structural and functional testing is reduced to a reachability problem within the module's control flow graph, as known from bounded model checking.

The solution of the reachability problem, presented in this thesis is based on symbolic execution. The strength of symbolic execution is in its precision and ability to reason about multiple program inputs simultaneously [65]. However, symbolic execution also has limitations, which were analyzed, and new algorithms were developed, which allow to overtake most of the identified limitations. The discussed algorithms were illustrated by examples demonstrating the proceedings.

The test case selection process was discussed and expansion and selection strategies minimizing the size of the structure underlying this process (STCT) and the number of test cases needed for the coverage achievement were presented.

The elaborated algorithms and strategies were implemented in a test generator, CTGEN, whose architecture was presented. CTGEN was used in industrial scale test campaigns for embedded systems code in the automotive domain. The overview over the most challenging as well as over the most common modules under test from these campaigns was presented in the industrial study. Furthermore, the performance of CTGEN was compared to other test generation tools. The evaluation was performed both with synthetic examples evaluating the respective tool's specific capabilities and with embedded systems code from an industrial automotive application. The results of the evaluation are encouraging.

7.2 Assessment of Results

The objective of this dissertation is the development of a framework for automated verification of C modules. The verification of a module under test is based on the provided specification. Therefore, an annotation language was developed (see Chapter 3) which enables the user to define the expected behavior of a module under test by means of pre- and postconditions, to refine the specification with the help of auxiliary variables, to introduce functional coverage by the definition of test cases with corresponding requirements and to reason about global variables, initial values of variables and return values of the module under test.

Via specification of pre- and postconditions the verification problem is reduced to a reachability problem within the control flow graph of the module under test. To prove a verification condition symbolic execution [65, 28] is used: it is sufficient to execute all program paths symbolically and show that the statements indicating the violation of the verification condition are unreachable. In case when such a statement can be covered, a counter example is produced.

The strengths and limitations of symbolic execution were analyzed and new algorithms resolving some of the detected limitations were introduced. To evaluate the results achieved in symbolic execution, we recall the limitations of symbolic execution which were identified in Section 5.2 and enumerate which of them were solved:

1. **Float/double data type variables**. This limitation is successfully overcome by the constraint solver SONOLAR [82], which supports the test generation process.

2. **Non-linear arithmetic operations**. Similar to the previous case, this limitation is surmounted by the underlying solver SONOLAR.

3. **Bitwise operations**. Similar to the previous two limitations, the elimination of this limitation is due to the underlying solver.

4. **External function calls**. This limitation was successfully solved by the introduction of automatically generated mock objects. The algorithm developed for automated stub generation is discussed in Section 5.12.2. The behavior of the generated stubs can be modeled by means of an annotation language that is presented in Section 5.12.3.

5. **Pointers**. The approach to abstract pointers to pairs of integers was developed. This abstraction enables the solution of constraints over pointers by the underlying constraint solver capable of integer arithmetic. The algorithm implementing this approach is discussed and illustrated in Section 5.7.

6. **Symbolic offsets**. The algorithm employing the capability of the underlying solver of array theories was developed to handle array inputs. This algorithm as well as an illustrating example are discussed in Section 5.11. Only arrays of atomic types are handled, since SONOLAR does not support structure or union types. However, the solver treats arrays as bit vectors [21] which implies, that SONOLAR is basically able to handle arrays of complex data types, but internal data structures of the tool chain do not support this presently.

Note that the limitations, identified by the authors of [89] as most dominant, which prevented tested tools from generating high coverage (pointers and external function calls), are resolved by the algorithms developed in this thesis. Nevertheless, the following limitations remain: function pointers, recursive data structures and multithreading.

To alleviate the state explosion problem, expansion and selection strategies were developed and are presented in Chapter 4. These strategies make the complete expansion of the structure underlying the test generation process (STCT) not always necessary. Additionally, the introduced search strategies allow to produce maximal code coverage with a minimal number of test cases. This leads to a better performance compared with other tools when handling functions with a big branching factor (see Table 6.2, function f_1). However, the developed test generator was not able to achieve the maximal possible coverage of this function, so that the developed strategies have to be improved further. One of the possible improvements (backtracking) of this method is discussed in Section 7.4.

The developed strategies and algorithms are implemented in the test generator CTGEN, presented in Chapter 2. CTGEN was evaluated and compared with the competing tools KLEE [22] and PathCrawler [18], both with synthetic examples estimating the tool's specific capabilities and with embedded systems code from an industrial automotive application (Chapter 6). CTGEN delivered comparable results or even outperformed the others for the most complex function. Furthermore, CTGEN was used in industrial scale test campaigns for embedded systems code in the automotive domain and demonstrated very good results. The overview over the most challenging as well as of the most common modules under test from these campaigns is presented in the industrial study in Appendix 7.4.

CTGEN supports statement and branch coverage. While the branch coverage is a popular test technique [19], the execution of branches does not imply that all combinations of control transfers are tested [107]. The experimental evaluations show that branch coverage is not a good indicator for the effectiveness of the test suite [102]. Standards for safety-critical systems such as [4, 40] require – in addition to the statement and branch coverage criteria – compliance with other structural coverage criteria such as MCDC coverage (for software of criticality level A). Thus, the expansion of the test generator to support further coverage criteria is reasonable. We discuss the possibilities for an integration of MCDC and path coverage as well as of the boundary value analysis in Section 7.4.

7.3 Discussion of Alternatives

State Explosion Problem The STCT used in this thesis for symbolic test case generation, however, has an obvious problem: state explosion. The *state merging* techniques [68, 53] reduce the number of states and number of paths to be explored. Thus, they in fact work against the state explosion problem but also increase the size of the symbolic path conditions (which impairs the performance) and handicap the application of search strategies. Furthermore, to alleviate the state explosion problem of the STCT, the search and expansion strategies are developed and discussed in this thesis (Section 4.3). These strategies make the complete expansion of the STCT not always necessary. Additionally, the introduced search strategies allow to produce the maximal code coverage with a minimal number of test cases. This leads to better performance compared to other tools when handling functions with a big branching factor (see Table 6.2, function f_1).

Similar to search strategies proposed in this thesis, other search heuristics were presented by researchers to address the state explosion problem. Among these techniques we want to highlight the Random Path Selection and the Coverage-Optimized Search (KLEE [22]), the Best First Search (EXE [23]), the Generational Search (SAGE [49]) and the Hybrid Concolic Testing [71].

Another approach for reducing the number of states, which is orthogonal to search strategies, is *compositional symbolic execution* [6, 45]. It proposes to test functions in isolation, creating the function summaries which can be re-used by testing the higher-level functions. This approach would help to avoid the repeated analysis of these functions at every call as it is the case with the function inlining approach and reduce the number of paths to be explored. The automated generation of stubs proposed in this thesis for handling of external function calls (Section 5.12.2) can also be applied to defined function calls to omit inlining. This approach is not as exact as compositional symbolic execution, but it is less expensive and also allows to handle external function calls. Furthermore, the stubs' behavior can be influenced by the user by the definition of pre- and postconditions as it is discussed in Section 5.12.3. However, the compositional symbolic execution is very promising, and it can be an advantage to integrate this approach in the developed test generator for the handling of defined function calls.

Static vs dynamic symbolic execution Dynamic symbolic execution or concolic testing [47, 23, 103, 93] is a technique, which performs a concrete execution on random inputs and records the path constraints along the executed path. The collected path constraints are used to estimate new values which guide the following executions through alternative paths. It is argued, that dynamic symbolic execution is more powerful because of its ability to use concrete values for the estimation of the path constraints [46]. However, as was shown in the case study of concolic testing tools [89], most of the limitations of the dynamic symbolic execution are caused by limitations of the static symbolic execution. Hence, we have focused on these limitations. Furthermore, static symbolic analysis can be extended so that whenever it encounters an expression which cannot be handled symbolically due to limitations, the concretization of the symbolic values can be performed, though [64, 85, 88].

Memory representation The theory of arrays [21] is broadly used for modeling the memory of a program in software verification, bounded model checking, symbolic execution etc [42]. Usually in this case, the whole memory is represented as a big one-dimensional array of bit-vectors [21, 97]. It allows to model the program memory very precisely and use an SMT solver for the resolution of typical aliasing problems. Our symbolic execution algorithm is more complex but it is also very precise and the solver has to make less decisions. Furthermore, our approach allows to avoid keeping a huge array representing the whole memory of a program.

7.4 Future Work

Backtracking One of the problems observed during the application of the test generator in the industrial scale test campaigns was the enormous number of paths which had to be analyzed. To alleviate this problem expansion and search strategies were developed. These strategies made it possible for CTGEN to outperform other tools when generating test data for the most challenging function f_1 with a huge branching factor (see Chapter 6), but still, CTGEN was not able to generate the test data for maximal achievable coverage of this function. One of the possibilities to further improve the developed search strategies would, therefore, be to learn from SAT solvers, that successfully apply backtracking by search for variable assignment, evaluating the given Boolean formula to true. Conflict Driven Clause Learning (CDCL) algorithm [73, 62, 106, 96, 25] was proposed in the late nineties and made a big contribution to the growing popularity of SAT solvers. CDCL is employed in solvers like MiniSAT [39, 98], Zchaff [106] and Z3 [75]. The main idea of CDCL algorithm is as follows:

- Pick an unassigned variable from the given Boolean formula and assign it to 0 or 1.
- Perform Boolean constraint propagation.

- In case when there is any conflict, identify the *conflict clause* and non-chronologically backtrack to the decision level that caused this conflict and try to find the solution with an inversed decision.
- Otherwise go back to step 1 and proceed until no unassigned variables exist.

The test generator can proceed in a similar way: in case, when the path constraint was detected as infeasible, identify the minimal subset of clauses, which is still infeasible and non-chronologically backtrack to the edge where the clause from the identified subset is a guard condition. Try another edge outgoing from the same node as the found one.

However, the identification of the minimal unsatisfiable core of the path constraint by the solver is relatively expensive [34, 77] and, since the objective of the generator is to cover all branches and not exactly the one, whose path condition is infeasible, the application of this approach is meaningful only in cases when the CFG is mostly covered and the generator has troubles to cover *exactly* this branch. For more speedup of the test data generation process non-chronological backtracking can be supported by abstract interpretation, which is significantly faster than a SMT solver [83]. This technique was already successfully applied in a framework for automated model-based test case and test data generation [81, 82].

Boundary Value Analysis Practice shows that errors often appear at the boundaries of the equivalence classes [76]. Hence, it is advisable to generate test cases that explore such boundary conditions and to expand the test data generator to provide support of the boundary value analysis. The input equivalence classes can be identified on the basis of defined preconditions and test cases. The output equivalence classes – with the help of defined postconditions and test cases. After the equivalence classes are determined, the boundary values can be found by introducing additional constraints requiring that the values of variables lay on the boundaries of the equivalence classes or near to them. To support the boundary value analysis of structure equivalence classes, the approach proposed in [79] can be used: the source code of the module under test is automatically instrumented in such a way that new branches are introduced, whose coverage leads to boundary value coverage. For this purpose each branch condition is analyzed and new conditional statements containing constraints that need to be satisfied to achieve boundary value coverage are inserted. For example, if a module under test contains an `if` statement like `if(x >= 3)`, where `x` is an integer, the source code would be instrumented as follows [79]:

```
if(x == 3){}
else if(x > 3){}
else if(x == 3 - 1){}
else if(x < 3 - 1){}

// actual program code
if(x >= 3){
  ...
```

This technique allows to use the existing test data generator, which is optimized to achieve the maximal possible branch coverage but also dramatically increases the branching factor of the module under test and the number of possible paths correspondingly. Alternatively, the internal test data generation mechanism can be modified, so that for each branch under consideration a new path constraint is generated where the current branch condition is modified in such a way that satisfaction of this alternative path constraint leads to the coverage of a boundary value. In this way, for each analyzed branch condition maximal three additional path constraints will be examined. Whereas by the instrumentation approach each instrumented branch multiplies the number of path constraints to be analyzed by maximal factor 5.

Path Coverage is the strongest criterion in white box testing but is generally not feasible because of loops or huge branching factor of the module under test [107]. However, it is reasonable to attempt to provide maximal possible path coverage, since in this way the level of confidence in reliability of the module under test increases. In the course of the present work experiments with the detection of all paths through the function were undertaken and we found out that the extension for naive path coverage support can be easily done. However it would be more of interest to extend the expansion and selection strategies discussed in this thesis so that the generator first would have to aim to achieve a 100% branch coverage and only after that would attempt the path coverage. This approach would ensure that the generator will get better results in cases when the complete path coverage is not possible due to size and branching factor of module under test, since the coverage of different paths not always leads to the coverage of new branches.

MCDC Coverage The standards for safety-critical systems such as [4, 40] require additionally to the statement and branch coverage criteria (supported by the developed test generator) compliance with other structural coverage criteria such as MCDC coverage (for software of criticality level A). To provide MCDC coverage support, the instrumentation approach proposed in [79] can be used. Similar to the technique for accessing boundary value coverage, the source code of the module under test is automatically instrumented with new conditional statements which need to be satisfied in order to achieve MCDC coverage. These transformations are based on the Correlated Active Clause Coverage (CACC), also known as the masking MCDC criterion [5]: *"For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. TR has two requirements for each c_i: c_i evaluates to $true$ and c_i evaluates to $false$. The values chosen for the minor clauses c_j must cause p to be $true$ for one value of the major clause c_i and $false$ for the other, that is, it is required that $p(c_i = true) \neq p(c_i = false)$".* Where P is a set of predicates, C_p is a set of clauses in p for each $p \in P$ and TR is a set of test requirements that must be satisfied. For example, for the `if` statement

```
if(x && (y || z)) { ... }
```

the following instrumentation code will be introduced [79]:

```
if( (true && (y || z)) != (false && (y || z))){
  if(x) { }
  else if(!x) { }
}
if( (x && (true || z)) != (x && (false || z))){
  if(y) { }
  else if(!y) { }
}
if( (x && (y || true)) != (x && (y || false))){
  if(z) { }
  else if(!z) { }
}
```

Again, this technique allows the usage of the existing test data generator but increases the branching factor of the module under test. However, the introduction of alternative path conditions – as in the case of boundary value analysis – is in this case not possible in the developed test generator. This is due to the fact that the source code of the module under test is in this stage of the generation process already in a three-address statement representation so that a single guard condition does not hold enough information. This, however, would be necessary for the constraint generation, whose satisfaction would lead to MCDC coverage.

Remaining Limitations The following limitations of symbolic execution remain unresolved: handling of function pointers, recursive data structures and multithreading. To make the analysis more precise and to allow the developed test generator to handle a larger range of modules under test, further algorithms have to be developed. Recursive input data structures can be handled e.g. with help of *lazy initialization* algorithm already used by Symbolic PathFinder [87, 64] and Bogor/Kiasan [33, 32]. The basic principle of this algorithm is as follows: at the beginning all recursive input data structures have uninitialized fields. These fields are initialized lazily, not until they are accessed by the symbolic execution. A field of type T is initialized nondeterministically to null, to a reference to a new object of type T with uninitialized fields or a reference to an existing object of type T built during earlier field initialization. [64]

Bibliography

[1] *GIMPLE - GNU Compiler Collection (GCC)*. Available at http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html. Last visited March 2013.

[2] *PathCrawler*. Available at http://pathcrawler-online.com/. Last visited September 2012.

[3] (2010): *International Software Testing Qualification Board, ISTQB Standard Glossary of Terms used in Software Testing V.2.1*. Available at http://www.german-testing-board.info/downloads/pdf/CT_Glossar_EN_DE_V21.pdf.

[4] RTCA. Special Committee 167 & Eurocae. Working Group 12 (1992): *Software Considerations in Airborne Systems and Equipment Certification*. *Radio Technical Commission for Aeronautics (RTCA), European Organization for Civil Aviation Electronics (EUROCAE), DO178-B*.

[5] Paul Ammann, Jeff Offutt & Hong Huang (2003): *Coverage criteria for logical expressions*. In: *In 14th International Symposium on Software Reliability Engineering (ISSRE'03*, IEEE Computer Society Press, pp. 99–107.

[6] Saswat Anand, Patrice Godefroid & Nikolai Tillmann (2008): *Demand-driven compositional symbolic execution*. In: *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, pp. 367–381.

[7] Hans Schaefer Andreas Spillner, Tilo Linz (2007): *Software Testing Foundations (2nd edition)*. Rocky Nook Inc.

[8] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu & Salvatore Sabina (2010): *Using Bounded Model Checking for Coverage Analysis of Safety-Critical Software in an Industrial Setting*. *Journal of Automated Reasoning* 45, pp. 397–414, doi:10.1007/s10817-010-9172-3.

[9] Krzysztof R. Apt & Ernst-Rüdiger Olderog (1991): *Verification of Sequential and Concurrent Programs*. Springer, doi:10.1007/978-1-84882-745-5.

[10] Motor Industry Software Reliability Association (2004): *MISRA-C:2004: Guidelines for the Use of the C Language in Critical Systems*. Mira Books Limited.

[11] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant & Dawn Song (2011): *Statically-directed dynamic automated test generation*. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, ACM, New York, NY, USA, pp. 12–22, doi:10.1145/2001420.2001423.

[12] Bahareh Badban, Martin Fränzle, Jan Peleska & Tino Teige (2006): *Test automation for hybrid systems*. In: *Proceedings of the 3rd international workshop on Software quality assurance*, SOQUA '06, ACM, New York, NY, USA, pp. 14–21, doi:10.1145/1188895.1188902.

[13] Michael F. Banahan, Declan Brady & Mark Doran (1991): *The C book - featuring the ANSI C standard (2. ed.)*. Addison-Wesley.

[14] Armin Biere & Carla P. Gomes, editors (2006): *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*. *Lecture Notes in Computer Science* 4121, Springer.

[15] Armin Biere, Marijn Heule, Hans van Maaren & Toby Walsh, editors (2009): *Handbook of Satisfiability*. *Frontiers in Artificial Intelligence and Applications* 185, IOS Press.

[16] Robert V. Binder (1999): *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series, Addison Wesley.

[17] Peter Boonstoppel, Cristian Cadar & Dawson Engler (2008): *RWset: attacking path explosion in constraint-based test generation*. In: *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, pp. 351–366.

[18] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger & N. Williams (2009): *Automating structural testing of C programs: Experience with PathCrawler*. In: *Automation of Software Test, 2009. AST '09. ICSE Workshop on*, pp. 70 –78, doi:10.1109/IWAST.2009.5069043.

[19] P. Bourque & R. Dupuis (2004): *Guide to the Software Engineering Body of Knowledge 2004 Version*. *Guide to the Software Engineering Body of Knowledge, 2004. SWEBOK*.

[20] Robert S. Boyer, Bernard Elspas & Karl N. Levitt (1975): *SELECT — a formal system for testing and debugging programs by symbolic execution*. In: *Proceedings of the international conference on Reliable software*, ACM, New York, NY, USA, pp. 234–245, doi:10.1145/800027.808445.

[21] Robert Brummayer & Armin Biere (2008): *Lemmas on demand for the extensional theory of arrays*. In: *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, ACM, New York, NY, USA, pp. 6–11, doi:10.1145/1512464.1512467.

[22] Cristian Cadar, Daniel Dunbar & Dawson Engler (2008): *KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs*. In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, USENIX Association, Berkeley, CA, USA, pp. 209–224. Available at `http://dl.acm.org/citation.cfm?id=1855741.1855756`.

[23] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill & Dawson R. Engler (2008): *EXE: Automatically Generating Inputs of Death*. *ACM Trans. Inf. Syst. Secur.* 12(2), pp. 10:1–10:38, doi:10.1145/1455518.1455522.

[24] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann & Willem Visser (2011): *Symbolic execution for software testing in practice: preliminary assessment*. In: *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, ACM, New York, NY, USA, pp. 1066–1071, doi:10.1145/1985793.1985995.

[25] Koen Claessen, Niklas Een, Mary Sheeran, Niklas Sörensson, Alexey Voronov & Knut Akesson (2009): *SAT-Solving in Practice, with a Tutorial Example from Supervisory Control*. *Discrete Event Dynamic Systems* 19(4), pp. 495–524, doi:10.1007/s10626-009-0081-8.

[26] Edmund Clarke, Daniel Kroening & Flavio Lerda (2004): *A Tool for Checking ANSI-C Programs*. In Kurt Jensen & Andreas Podelski, editors: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, *Lecture Notes in Computer Science* 2988, Springer, pp. 168–176, doi:10.1007/978-3-540-24730-2_15.

[27] L.A. Clarke (1976): *A program testing system*. In: *Proceedings of the 1976 annual conference*, ACM '76, ACM, New York, NY, USA, pp. 488–491, doi:10.1145/800191.805647.

[28] L.A. Clarke (1976): *A System to Generate Test Data and Symbolically Execute Programs*. *Software Engineering, IEEE Transactions on* SE-2(3), pp. 215 – 222, doi:10.1109/TSE.1976.233817.

[29] Lori A. Clarke & Debra J. Richardson (1985): *Applications of symbolic evaluation*. *J. Syst. Softw.* 5(1), pp. 15–35, doi:10.1016/0164-1212(85)90004-4.

[30] Peter Collingbourne, Cristian Cadar & Paul H.J. Kelly (2011): *Symbolic crosschecking of floating-point and SIMD code*. In: *Proceedings of the sixth conference on Computer systems*, EuroSys '11, ACM, New York, NY, USA, pp. 315–328, doi:10.1145/1966445.1966475.

[31] International Electrotechnical Commission (2006): *IEC 61508 - Functional safety of electric/-electronic/programmable electronic safety-related systems*. *IEC*.

[32] Xianghua Deng, Jooyong Lee & Robby (2006): *Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems*. In: *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pp. 157–166, doi:10.1109/ASE.2006.26.

[33] Xianghua Deng, Jooyong Lee & Robby (2012): *Efficient and formal generalized symbolic execution*. *Automated Software Engineering* 19(3), pp. 233–301, doi:10.1007/s10515-011-0089-9.

[34] Nachum Dershowitz, Ziyad Hanna & Alexander Nadel (2006): *A Scalable Algorithm for Minimal Unsatisfiable Core Extraction*. In: *SAT*, pp. 36–41. Available at `http://dx.doi.org/10.1007/11814948_5`.

[35] L.P. Deutsch (1973): *An Interactive Program Verifier*. Ph.d. th., University of California, Berkeley.

[36] Edsger W. Dijkstra (1975): *Guarded commands, nondeterminacy and formal derivation of programs*. *Commun. ACM* 18(8), pp. 453–457, doi:10.1145/360933.360975.

[37] Edsger W. Dijkstra (1976): *A Discipline of Programming*. Prentice-Hall.

[38] Deepak D'Souza, Telikepalli Kavitha & Jaikumar Radhakrishnan, editors (2012): *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*. *LIPIcs* 18, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. Available at `http://drops.dagstuhl.de/portals/extern/index.php?semnr=12014`.

[39] N. Een & N. Sörensson (2005): *MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition*.

[40] European Committee for Electrotechnical Standardization (2001): *Cenelec EN 50128, Railway Applications – Communications, Signalling and Processing Systems - Software for Railway Control and Protection Systems*. *CENELEC, Brussels*.

[41] Bassem Elkarablieh, Patrice Godefroid & Michael Y. Levin (2009): *Precise pointer reasoning for dynamic test generation*. In: *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, ACM, New York, NY, USA, pp. 129–140, doi:10.1145/1572272.1572288.

[42] Stephan Falke, Florian Merz & Carsten Sinz (2012): *A Theory of Arrays with Set and Copy Operations (Extended Abstract)*. In Pascal Fontaine & Amit Goel, editors: *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories (SMT '12)*, Manchester, UK, pp. 97–106.

[43] R. W. Floyd (1967): *Assigning Meaning to Programs*. In: *Proceedings of the Symposium on Applied Maths*, 19, AMS, pp. 19–32.

[44] Verified Systems International GmbH (2012): *RT-Tester 6.x: User Manual*.

[45] Patrice Godefroid (2007): *Compositional dynamic test generation*. In Hofmann & Felleisen [58], pp. 47–54. Available at `http://doi.acm.org/10.1145/1190216.1190226`.

[46] Patrice Godefroid (2012): *Test Generation Using Symbolic Execution*. In D'Souza et al. [38], pp. 24–33. Available at `http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2012.24`.

[47] Patrice Godefroid, Nils Klarlund & Koushik Sen (2005): *DART: directed automated random testing*. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, ACM, New York, NY, USA, pp. 213–223, doi:10.1145/1065010.1065036.

[48] Patrice Godefroid, Shuvendu K. Lahiri & Cindy Rubio-González (2011): *Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation*. In Yahav [105], pp. 112–128. Available at `http://dx.doi.org/10.1007/978-3-642-23702-7_12`.

[49] Patrice Godefroid, Michael Y. Levin & David A. Molnar (2008): *Automated Whitebox Fuzz Testing*. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, The Internet Society.

[50] Robert Gold (2010): *Control flow graphs and code coverage*. International Journal of Applied Mathematics and Computer Science Volume 20, pp. 730 – 749. Available at `http://versita.metapress.com/content/85L623M5V6373H18`.

[51] Brian Hackett (2010): *sixgill*. Available at `http://sixgill.org/`. Last visited August 2012.

[52] A. Hall (1990): *Seven myths of formal methods*. Software, IEEE 7(5), pp. 11 –19, doi:10.1109/52.57887.

[53] Trevor Hansen, Peter Schachte & Harald Søndergaard (2009): *Runtime Verification*. chapter State Joining and Splitting for the Symbolic Execution of Binaries, Springer-Verlag, Berlin, Heidelberg, pp. 76–92, doi:10.1007/978-3-642-04694-0_6.

[54] Sidney L. Hantler & James C. King (1976): *An Introduction to Proving the Correctness of Programs*. ACM Comput. Surv. 8(3), pp. 331–353, doi:10.1145/356674.356677.

[55] Anne Mette Jonassen Hass (2008): *Guide to advanced software testing*. Artech House, Boston [u.a.]. Xxxi, 427 S. ; 24 cm : zahlr. graph. Darst.

[56] C. A. R. Hoare (1969): *An axiomatic basis for computer programming*. Commun. ACM 12(10), pp. 576–580, doi:10.1145/363235.363259.

[57] C. A. R. Hoare (2002): *Software pioneers*. chapter Assertions: a personal perspective, Springer-Verlag New York, Inc., New York, NY, USA, pp. 356–366. Available at `http://dl.acm.org/citation.cfm?id=944331.944354`.

[58] Martin Hofmann & Matthias Felleisen, editors (2007): *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. ACM. Available at `http://dl.acm.org/citation.cfm?id=1190216`.

[59] Bernard Homés (2012): *Fundamentals of software testing*. ISTE [u.a.], London.

[60] W.E. Howden (1977): *Symbolic Testing and the DISSECT Symbolic Evaluation System*. Software Engineering, IEEE Transactions on SE-3(4), pp. 266–278, doi:10.1109/TSE.1977.231144.

[61] Capers Jones & Olivier Bonsignour (2011): *The Economics of Software Quality*. Addison-Wesley Professional.

[62] Roberto Bayardo Jr & Robert C. Schrag (1997): *Using CSP Look-Back Techniques to Solve Real-World SAT Instances*. AAAI Press, pp. 203–208.

[63] Brian W. Kernighan (1988): *The C Programming Language*, 2nd edition. Prentice Hall Professional Technical Reference.

[64] Sarfraz Khurshid, Corina S. Păsăreanu & Willem Visser (2003): *Generalized symbolic execution for model checking and testing*. In: Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'03, Springer-Verlag, Berlin, Heidelberg, pp. 553–568. Available at `http://dl.acm.org/citation.cfm?id=1765871.1765924`.

[65] James C. King (1976): *Symbolic execution and program testing*. *Commun. ACM* 19(7), pp. 385–394, doi:10.1145/360248.360252.

[66] Nikolai Kosmatov (2008): *All-Paths Test Generation for Programs with Internal Aliases*. In: *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, ISSRE '08, IEEE Computer Society, Washington, DC, USA, pp. 147–156, doi:10.1109/ISSRE.2008.25. Available at `http://dx.doi.org/10.1109/ISSRE.2008.25`.

[67] S. Krishnamoorthy, M.S. Hsiao & L. Lingappan (2010): *Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs*. In: *Test Symposium (ATS), 2010 19th IEEE Asian*, pp. 59–64, doi:10.1109/ATS.2010.19.

[68] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur & George Candea (2012): *Efficient state merging in symbolic execution*. *SIGPLAN Not.* 47(6), pp. 193–204, doi:10.1145/2345156.2254088.

[69] Guodong Li, Indradeep Ghosh & Sreeranga Rajan (2011): *KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification*, *Lecture Notes in Computer Science* 6806, Springer Berlin / Heidelberg, pp. 609–615, doi:10.1007/978-3-642-22110-1_49.

[70] Helge Löding & Jan Peleska (2008): *Symbolic and Abstract Interpretation for C/C++ Programs*. *Electronic Notes in Theoretical Computer Science* 217(0), pp. 113 – 131, doi:10.1016/j.entcs.2008.06.045. Available at `http://www.sciencedirect.com/science/article/pii/S1571066108003885`.

[71] Rupak Majumdar & Koushik Sen (2007): *Hybrid Concolic Testing*. In: *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, IEEE Computer Society, Washington, DC, USA, pp. 416–426, doi:10.1109/ICSE.2007.41.

[72] Tatiana Mangels & Jan Peleska (2012): *CTGEN - a Unit Test Generator for C*. In: *Proceedings Seventh Conference on Systems Software Verification*, *EPTCS* 102, pp. 88–102. Available at `http://dx.doi.org/10.4204/EPTCS.102.9`.

[73] Joaäo P. Marques-silva & Karem A. Sakallah (1999): *GRASP: A Search Algorithm for Propositional Satisfiability*. *IEEE Transactions on Computers* 48, pp. 506–521.

[74] Aditya P. Mathur (2009): *Foundations of software testing: fundamental algorithms and techniques*. Pearson, Delhi [u.a.]. XX, 689 S. : graph. Darst.

[75] Leonardo Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C.R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 4963, Springer Berlin Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[76] Glenford J. Myers (2004): *The art of software testing (2. ed.)*. Wiley. Available at `http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471469122.html`.

[77] A. Nadel (2010): *Boosting minimal unsatisfiable core extraction*. In: *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pp. 221–229.

[78] Michael Newman: *Software Errors Cost U.S. Economy $59.5 Billion Annually. NIST Assesses Technical Needs of Industry to Improve Software-Testing*. Available at `http://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm`. Last visited Februar 2013.

[79] Rahul Pandita, Tao Xie, Nikolai Tillmann & Jonathan de Halleux (2010): *Guided Test Generation for Coverage Criteria*. In: *Proc. 26th IEEE International Conference on Software Maintenance (ICSM 2010)*. Available at `http://www.csc.ncsu.edu/faculty/xie/publications/icsm10-coverage.pdf`.

[80] Jan Peleska (2010): *Integrated and Automated Abstract Interpretation, Verification and Testing of C/C++ Modules*. In Dennis Dams, Ulrich Hannemann & Martin Steffen, editors: *Concurrency, Compositionality, and Correctness*, *Lecture Notes in Computer Science* 5930, Springer Berlin / Heidelberg, pp. 277–299, doi:10.1007/978-3-642-11512-7_18.

[81] Jan Peleska, Artur Honisch, Florian Lapschies, Helge Löding, Hermann Schmid, Peer Smuda, Elena Vorobev & Cornelia Zahlten (2011): *A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain*. In Burkhart Wolff & Fatiha Zaïdi, editors: *Testing Software and Systems*, *Lecture Notes in Computer Science* 7019, Springer Berlin Heidelberg, pp. 146–161, doi:10.1007/978-3-642-24580-0_11.

[82] Jan Peleska, Elena Vorobev & Florian Lapschies (2011): *Automated test case generation with SMT-solving and abstract interpretation*. In: *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, Springer-Verlag, Berlin, Heidelberg, pp. 298–312, doi:10.1007/978-3-642-20398-5_22. Available at `http://dl.acm.org/citation.cfm?id=1986308.1986333`.

[83] Jan Peleska, Elena Vorobev, Florian Lapschies & Cornelia Zahlten (2011): *Automated Model-Based Testing with RT-Tester. Technical Report*. Available at `http://www.informatik.uni-bremen.de/agbs/testingbenchmarks/turn_indicator/tool/rtt-mbt.pdf`.

[84] Mauro Pezzé & Michal Young (2008): *Software testing and analysis: process, principles and techniques*. Wiley, Hoboken, NJ. XXII, 488 S.

[85] Corina S. Păsăreanu, Neha Rungta & Willem Visser (2011): *Symbolic execution with mixed concrete-symbolic solving*. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, ACM, New York, NY, USA, pp. 34–44, doi:10.1145/2001420.2001425.

[86] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person & Mark Pape (2008): *Combining unit-level symbolic execution and system-level*

concrete execution for testing nasa software. In: *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, ACM, New York, NY, USA, pp. 15–26, doi:10.1145/1390630.1390635.

[87] Corina S. Păsăreanu & Willem Visser (2009): *A survey of new trends in symbolic execution for software testing and analysis*. *Int. J. Softw. Tools Technol. Transf.* 11(4), pp. 339–353, doi:10.1007/s10009-009-0118-1.

[88] CorinaS. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz & Neha Rungta (2013): *Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis*. *Automated Software Engineering* 20(3), pp. 391–425, doi:10.1007/s10515-013-0122-2.

[89] Xiao Qu & Brian Robinson (2011): *A Case Study of Concolic Testing Tools and their Limitations*. In: *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, IEEE Computer Society, Washington, DC, USA, pp. 117–126, doi:10.1109/ESEM.2011.20.

[90] C.V. Ramamoorthy, Siu-Bun F Ho & W.T. Chen (1976): *On the Automated Generation of Program Test Data*. *Software Engineering, IEEE Transactions on* SE-2(4), pp. 293–300, doi:10.1109/TSE.1976.233835.

[91] E.J. Schwartz, T. Avgerinos & D. Brumley (2010): *All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)*. In: *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 317–331, doi:10.1109/SP.2010.26.

[92] Koushik Sen & Gul Agha (2007): *A Race-Detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs*. In Eyal Bin, Avi Ziv & Shmuel Ur, editors: *Hardware and Software, Verification and Testing*, *Lecture Notes in Computer Science* 4383, Springer Berlin Heidelberg, pp. 166–182, doi:10.1007/978-3-540-70889-6_13.

[93] Koushik Sen, Darko Marinov & Gul Agha (2005): *CUTE: a concolic unit testing engine for C*. In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, ACM, New York, NY, USA, pp. 263–272, doi:10.1145/1081706.1081750.

[94] Koushik Sen, Darko Marinov & Gul Agha (2005): *CUTE: a concolic unit testing engine for C. Technical Report*. UIUCDCS-R-2005-2597, UIUC.

[95] Koushik Sen, Darko Marinov & Gul Agha (2006): *Concolic Testing of Sequential and Concurrent Programs*.

[96] João P. Marques Silva, Inês Lynce & Sharad Malik (2009): *Conflict-Driven Clause Learning SAT Solvers*. In Biere et al. [15], pp. 131–153. Available at `http://dx.doi.org/10.3233/978-1-58603-929-5-131`.

[97] Carsten Sinz, Stephan Falke & Florian Merz (2010): *A Precise Memory Model for Low-Level Bounded Model Checking*. In: *Proceedings of the 5th International Workshop on Systems Software Verification (SSV '10)*, Vancouver, Canada.

[98] Niklas Sörensson & Armin Biere (2009): *Minimizing Learned Clauses*. In Oliver Kullmann, editor: *Theory and Applications of Satisfiability Testing - SAT 2009*, *Lecture Notes in Computer Science* 5584, Springer Berlin Heidelberg, pp. 237–243, doi:10.1007/978-3-642-02777-2_23.

[99] G. Tassey (2002): *The economic impacts of inadequate infrastructure for software testing*. Technical Report, National Institute of Standards and Technology.

[100] Nikolai Tillmann & Jonathan de Halleux (2008): *Pex–White Box Test Generation for .NET*. In Bernhard Beckert & Reiner Hähnle, editors: *Tests and Proofs*, *Lecture Notes in Computer Science* 4966, Springer Berlin / Heidelberg, pp. 134–153, doi:10.1007/978-3-540-79124-9_10.

[101] Dries Vanoverberghe, Nikolai Tillmann & Frank Piessens (2009): *Test Input Generation for Programs with Pointers*. In Stefan Kowalewski & Anna Philippou, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 5505, Springer Berlin / Heidelberg, pp. 277–291, doi:10.1007/978-3-642-00768-2_25.

[102] Yi Wei, Bertrand Meyer & Manuel Oriol (2012): *Is Branch Coverage a Good Measure of Testing Effectiveness?* In Bertrand Meyer & Martin Nordio, editors: *Empirical Software Engineering and Verification*, *Lecture Notes in Computer Science* 7007, Springer Berlin Heidelberg, pp. 194–212, doi:10.1007/978-3-642-25231-0_5.

[103] Nicky Williams, Bruno Marre, Patricia Mouy & Muriel Roger (2005): *PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis*. In Mario Dal Cin, Mohamed Kaâniche & András Pataricza, editors: *Dependable Computing - EDCC 5*, *Lecture Notes in Computer Science* 3463, Springer Berlin / Heidelberg, pp. 281–292. Available at `http://dx.doi.org/10.1007/11408901_21`.

[104] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui & John Fitzgerald (2009): *Formal methods: Practice and experience*. *ACM Comput. Surv.* 41(4), pp. 19:1–19:36, doi:10.1145/1592434.1592436.

[105] Eran Yahav, editor (2011): *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*. *Lecture Notes in Computer Science* 6887, Springer. Available at `http://dx.doi.org/10.1007/978-3-642-23702-7`.

[106] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz & Sharad Malik (2001): *Efficient conflict driven learning in a boolean satisfiability solver*. In: *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, IEEE Press, Piscataway, NJ, USA, pp. 279–285. Available at `http://dl.acm.org/citation.cfm?id=603095.603153`.

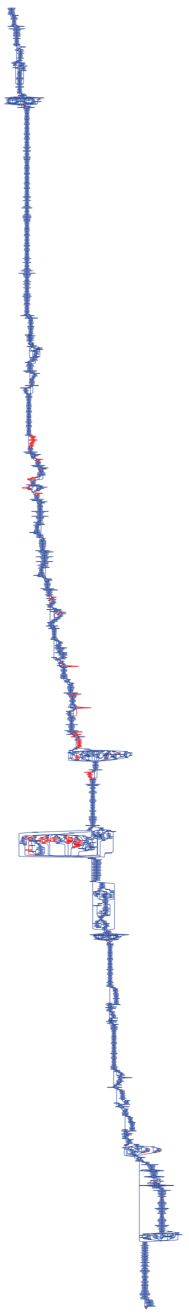
[107] Hong Zhu, Patrick A. V. Hall & John H. R. May (1997): *Software unit test coverage and adequacy*. *ACM Comput. Surv.* 29(4), pp. 366–427, doi:10.1145/267580.267590. Available at `http://doi.acm.org/10.1145/267580.267590`.

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

München, den 14.10.2013 Tatiana Mangels

Industrial Case Study



CTGEN was used in industrial scale test campaigns for embedded systems code in the automotive domain and demonstrated very good results. During these test campaigns CTGEN has analyzed functions which contained bit field and structure pointer inputs, unions accesses and pointer comparisons. Some of them were automatically generated from a Simulink model, which made automated testing mandatory since small model changes affected the whole structure of the generated code, so that reuse of existing unit tests was impossible. Some of the analyzed functions were exceptionally long because insufficient hardware resources required to reduce the amount of function calls.

The results achieved by CTGEN in these test campaigns are presented in Chapter 6, where an overview of analyzed functions, reached coverage and number of generated test cases is given. As we point out in this overview, the most challenging function was f_1. It has over 2000 lines of code (714 executable lines) and 492 branches. It makes use of structure pointer parameters, bit vectors and complex branch conditions. In this chapter we give a more detailed insight into the structure of this function. We present some of the most typical parts of f_1 as anonymized code as well as in the form of a CFG. However, the function f_1 with its huge branch factor and length represents an exceptional case. Therefore, in conclusion of this chapter, we also demonstrate a typical function that was analyzed during the test campaigns which utilized CTGEN.

We have already mentioned that the function f_1 has a remarkable size. Still, to give an impression of its dimensions, we demonstrate the CFG of this function scaled to the size of this page on the right.

We now consider the first segment of f_1. It represents a typical calculation of a local auxiliary variable, which in turn participates in further calculations of other auxiliary variables or in conditions of `if` statements. Here we examine the computations made to estimate the value of the auxiliary variable `local_d`, for a better overview all occurrences of `local_d` are highlighted in red:

```
1  i = (global1.SF > parameter->x2);
2  local2 = ((parameter->x2 - parameter->x1) << 7);
3  ar = (local2 == 0);
4  ...
5  if (ar) {
6    local1 = 768U;
7  } else {
8    local1 = (uint16_T)func1(((int32_T)((parameter->y2 - parameter->y1) << 7)) << 8,
9                             (int32_T)local2);
10 }
11 if (i) {
12   local_d = (parameter->y2 << 7);
13 } else {
14   if (global1.SF < parameter->x1) {
15     local_d = (parameter->y1 << 7);
16   } else {
17     local_d = ((int16_T)((((int32_T)((global1.SF - parameter->x1) << 7)) *
18               ((int32_T)local1)) >> 8)) + (parameter->y1 << 7);
19   }
20 }
21
22 local_d = (((local_d >> 7) * parameter->off) << 1);
23 ...
24 local_DOTA = (uint8_T)(((local_d >> 2) + (global1.DOTA << 6)) >> 6);
25 ...
26 if(local_A <= local_DOTA){
27   ...
```

`global1` is a global variable of a structure type and `parameter` is an input parameter of a pointer structure type. The value of `local_d` is dependent on the values of other auxiliary variables `local1` and `local2`, which are highlighted in blue. While the value of the variable `local2` depends only on inputs, so that it is simple to set it in order to satisfy a path constraint, the value of `local1` is more complicated to estimate, since its calculation depends not only on inputs or auxiliary variables, but, in case when the variable `ar` is evaluated to `false`, includes also an invocation of a function call `func1()`. Consequently, the estimation of the value of `localD` is even more complicated, since it can depend on the value of `local1`. Henceforth the deeper in the code of the function we go, the more difficult it is to estimate the values of the auxiliary variables. Most of the auxiliary variables not only take part in computations of other auxiliary variables, but they also participate in guard conditions. For example, the variable `local_DOTA`: the condition of the `if` statement in line 26 of our example is evaluated to `true` if the value of `local_DOTA` is greater than or equal to the value of another auxiliary variable `local_A` (which computation we have omitted here to keep the example simple, but it is not less complicated than the computation of `local_DOTA`). To make the calculations even more complex, the expressions for the estimation of the values of auxiliary variables contain not only arithmetical or Boolean operations but also bit shifting and casting. The manual elaboration of test cases for such conditions is very laborious and time consuming.

The next code example represents a typical `switch` statement implemented in f_1:

```
if (global2.is_active == 0) {
  global2.is_active = 1U; global2.state = value1;
  global1.state_int = ((uint8_T)1U);
} else {
  switch (global2.state) {
  case value2:
    if (local1 == TRUE) {
      global2.state = value5; global1.state_int = ((uint8_T)9U);
    } else if (local2 == TRUE) {
      global2.state = value3; global1.state_int = ((uint8_T)8U);
    } else {
      if (local3 == TRUE) {
        global2.state = value6; global1.state_int = ((uint8_T)3U);
      }
    }
    break;
  case value3:
    if ((local4 || (!local5)) == 1) {
      global2.state = value6; global1.state_int = ((uint8_T)3U);
    } else {
      if (local6 == TRUE) {
        global2.state = value5; global1.state_int = ((uint8_T)9U);
      }
    }
    break;
  case value4:
    if (local7 == TRUE) {
      global2.state = value8; global1.state_int = ((uint8_T)6U);
    } else {
      if (local7a == TRUE) {
        global2.state = value6; global1.state_int = ((uint8_T)3U);
      }
    }
    break;
  case value5:
    if (global2.f.Q == TRUE) {
      global2.state = value6; global1.state_int = ((uint8_T)3U);
    } else {
      if (((local8 || (local9 > 57600U)) && (local15 == ((uint8_T)9U))) == 1) {
        global2.state = value7; global1.state_int = ((uint8_T)2U);
      }
    }
    break;
  case value6:
    if ((local10 && local11) == 1) {
      global2.state = value4; global1.state_int = ((uint8_T)4U);
    } else {
      if ((local8q || local12) == 1) {
        global2.state = value7; global1.state_int = ((uint8_T)2U);
      }
    }
    break;
  case value7:
    if (local13 == TRUE) {
      global2.state = value6; global1.state_int = ((uint8_T)3U);
    }
    break;
  case value1:
    if (input->SIL == TRUE) {
      global2.state = value7; global1.state_int = ((uint8_T)2U);
    }
    break;
  case value8:
    if ((local14 || local4f) == 1) {
      global2.state = value2; global1.state_int = ((uint8_T)7U);
    }
    break;
  default:
    global2.state = value1; global1.state_int = ((uint8_T)1U);
    break;
  }
}
```

All together f_1 contains 11 `switch` statements. Each `switch` statement corresponds to a state machine from the Simulink model and each case of a `switch` statement implements transitions from a particular state to another, dependent on the respective guard conditions.

The guard conditions occurring in the `switch` statement from the example depend on the global variable `global2` and on several local variables. The variable `global2` is of a structure type, which size is 2464 bits. It contains members of atomic types and of array or structure types as well. Even though `global2` is a global variable, it can not be considered as a pure input since its members are overwritten within the function f_1. The values of the local variables `localX` (`X` denotes the name add ons like `2` or `4f`) are computed in the manner described in the previous example. Furthermore, one has to keep in mind, that this switch statement is approximately from the middle of the function f_1 and the most of the variables participating in guard conditions of this example also take part in other guard conditions which were already evaluated during the processing of the function. These preceding guard conditions have restricted the range of allowed values of the variables, and this makes it more complicated to find a solution which satisfies the guard conditions. Nevertheless, CTGEN is still able to achieve 88% coverage of this `switch` statement. Figure 1 shows the control flow graph of the example. All covered nodes and edges of this CFG are drawn blue, all uncovered ones are drawn red.

The last example represents a typical function that was analyzed during the test campaigns CTGEN took part in. The function `example3()` has seven input parameters of atomic types and one input/output parameter of a structure pointer type. Furthermore the global variables `global3` and `global4` are used. They are like the global variables from the previous examples of structure types. The function `example3()` contains two calls to sub-functions – `func1()` and `func2()` and a `switch` statement. The conditions occurring in the function `example3()` depend on the input parameter `parameter2` and the global variables. However, similar to the conditions in the `switch` statement from the previous example, the member `ddcd` of the global variable `global4` (highlighted red in the listing below) is overwritten within the function by a value of a local variable which computation depends on inputs and contains bit shifting, casting and arithmetical operations. Nevertheless, in contrast to the previous example, the function `example3()` is a separate module, so that the values of its variables are not restricted by the previous guard conditions and CTGEN is able to achieve 100% branch coverage for this function with only 8 test cases. Figure 2 shows the control flow graph of the function `example3()`, all nodes and edges are drawn blue, since all branches were covered.

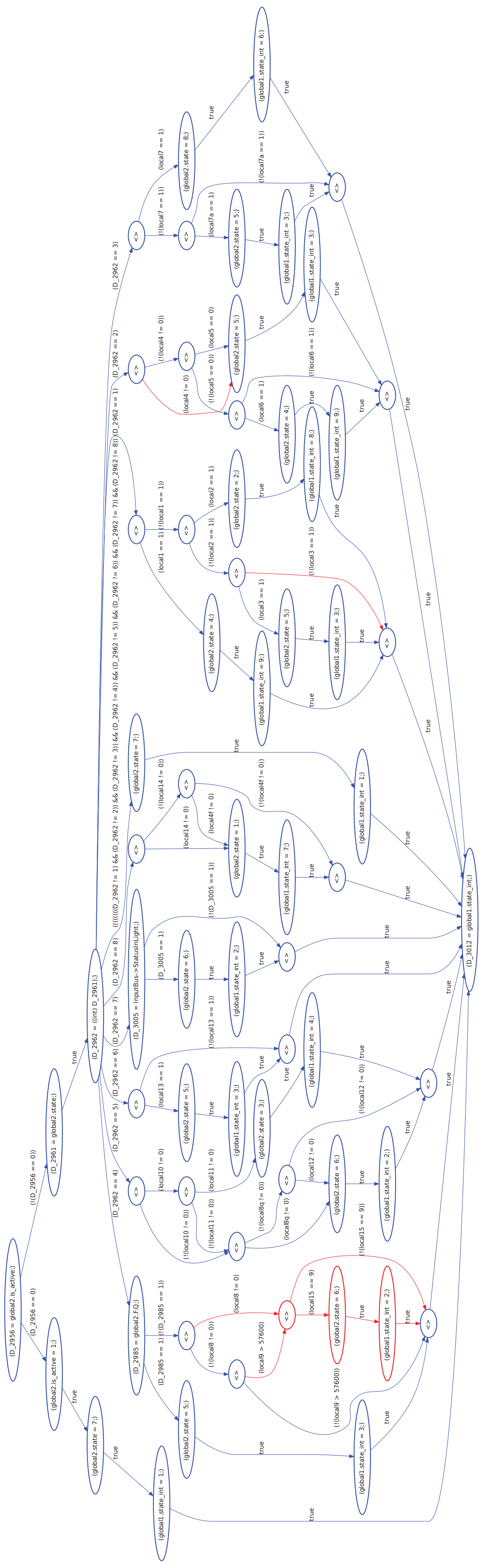


Figure 1: Control flow graph of a typical `switch` statement of function f_1.

```
void example3(unsigned char parameter1, unsigned char parameter2,
              unsigned char parameter3, short int parameter4,
              unsigned char parameter5, unsigned char parameter6,
              unsigned char parameter7, struct1 *parameter8)
{
  unsigned char local1;
  short int local2;
  short int local3;
  short int local4;
  short int local5;
  short int local6;
  unsigned int tmp;
  unsigned int tmp_0;

  local1 = parameter8->UD_D;
  func1(local1, parameter2, &parameter8->s_r_iO);
  func2(parameter1, parameter6, parameter7, parameter4,
        parameter8->s_r_iO.reset_offset_bt, parameter5,
        &parameter8->c_o);

  if (parameter2) {
    tmp = ((((unsigned int)((unsigned char)(global3.GWCO << 3)))
          * ((unsigned int)parameter4)) >> 6);
    if (((int32_T)tmp) > 65535L) {
      tmp = 65535UL;
    }

    tmp_0 = ((((unsigned int)((unsigned char)(global3.GCO << 3)))
            * ((unsigned int)parameter8->c_o.S)) >> 6);
    if (((int32_T)tmp_0) > 65535L) {
      tmp_0 = 65535UL;
    }

    tmp += tmp_0;
    if (((int32_T)tmp) > 65535L) {
      tmp = 65535UL;
    }

    tmp += (unsigned int)(((short int)parameter5) * ((short int)
      parameter8->c_o.S1));
    if (((int32_T)tmp) > 65535L) {
      tmp = 65535UL;
    }

    global4.ddsd = (short int)tmp;
    local2 = ((short int)parameter5) + ((short int)parameter8->c_o.S2);
    if (local2 > 255U) {
      local2 = 255U;
    }

    global4.ddcd = (unsigned char)local2;
  }

  local2 = (short int)((((unsigned int)global3.Y1) << 13) / 25UL);
  local3 = (short int)((((unsigned int)global3.Y2) << 13) / 25UL);
  local4 = (short int)((((unsigned int)global3.Y3) << 13) / 25UL);
  local5 = (short int)((((unsigned int)global3.Y4) << 13) / 25UL);

  if (parameter2) {
    if (!(global4.ddcd <= global3.X1)) {
      if (global4.ddcd <= global3.X2) {
        local2 -= (short int)((((unsigned int)((short int)((((unsigned int)
          (local2 - local3)) << 3) / ((unsigned int)((unsigned char)
          (global3.X2 - global3.X1)))))) * ((unsigned int)((unsigned char)
          (global4.ddcd - global3.X1)))) >> 3);
      } else {
        if (global4.ddcd <= global3.X3) {
          local2 = local3 - ((short int)((((unsigned int)((short int)
            ((((unsigned int)(local3 - local4)) << 3) / ((unsigned int)
            ((unsigned char)(global3.X3 - global3.X2)))))) * ((unsigned int)
            ((unsigned char)(global4.ddcd - global3.X2)))) >> 3));
        } else {
          if (global4.ddcd <= global3.X4) {
            local2 = local4 - ((short int)((((unsigned int)((short int)
              ((((unsigned int)(local4 - local5)) << 3) / ((unsigned int)
              ((unsigned char)(global3.X4 - global3.X3)))))) *
              ((unsigned int)((unsigned char)(global4.ddcd - global3.X3)))) >> 3));
          } else {
            if (global4.ddcd <= 200) {
              local2 = local5 - ((short int)((((unsigned int)
                ((short int)((((unsigned int)(local5 - 164U)) << 3) /
                ((unsigned int)((unsigned char)(200 - global3.X4))))))
                * ((unsigned int)((unsigned char)(global4.ddcd - global3.X4)))) >> 3))
                    ;
            } else {
              if (global4.ddcd < 255) {
                local6 = (short int)(164U - ((((short int)((unsigned char)
                  (global4.ddcd - 200))) * 5U) >> 3));
              } else {
                local6 = 128;
              }
              local2 = (short int)local6;
            }
          }
        }
      }
    }

    parameter8->UD4_D = (short int)((((unsigned int)
      global4.ddsd) * ((unsigned int)local2)) >> 12);
  }

  if (parameter2 > parameter8->UD3_D) {
    global4.bidv = parameter8->UD4_D;
  } else {
    if (parameter3) {
      global4.bidv = 0U;
    } else {
      global4.bidv = parameter8->UD1_D;
    }
  }

  parameter8->UD_D = parameter1;
  parameter8->UD3_D = parameter2;
  parameter8->UD1_D = global4.bidv;
}
```

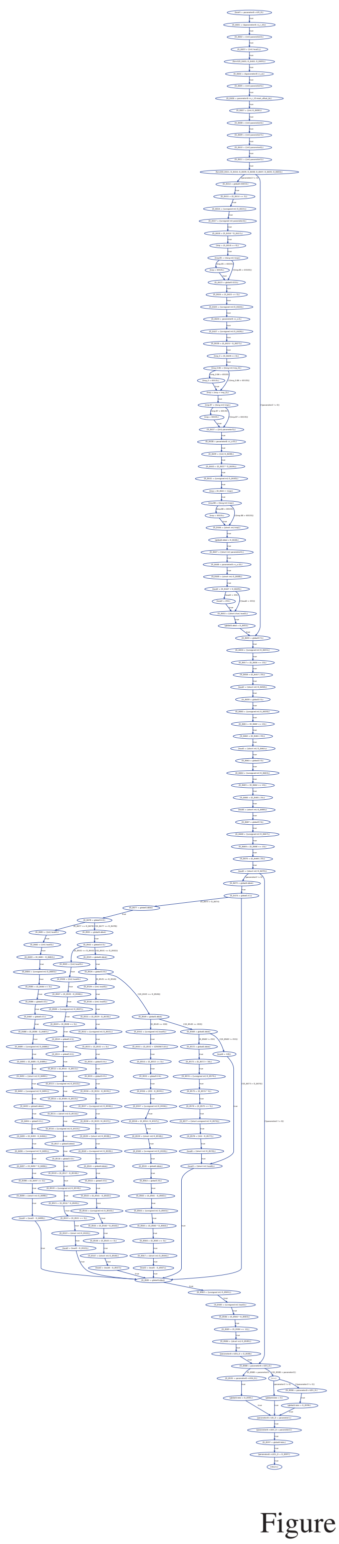


Figure 2: Control flow graph of the function `example3()`.

CTGEN Usage

Usage format for the unit test generator is as follows:

```
ctgen --sourceFile <SRC_FILE>
      [--pathForGeneratedTest <TARGET_DIR>]
      [--interpretFctCalls]
      [--stctMaxExp <NUM_OF_EXPANSIONS>]
      [--stctMaxLeaves <NUM_OF_LEAVES>]
      [--interpretFunction <FUNCTION_NAME>]
      [--numOfTestCases <NUM_OF_TEST_CASES>]
      [--proofMode]
      [--checkGlobalsForModification]
      [--onlyCFG]
      [--memorySaveMode]
      [--useGlobalsInitValues]
      [--arrayParameterSize <SIZE>]
      [--DB <PATH_TO_TC_DB>]
```

Parameter	**Mandatory**	**Description**
sourceFile	X	File to be analyzed.
pathForGeneratedTest		Path, where the generated test will be stored. If not set the generated test will be stored in the current directory.
interpretFctCalls		If set, function calls within analyzed modules, whose definition is available, will be interpreted. Otherwise they will be handled as stubs.
stctMaxExp		Set number of maximal allowed expansions for the STCT. After the maximal number of expansions is reached, the generation process will be stopped independent from the coverage status of the module under analysis.
stctMaxLeaves		Set number of maximal allowed leaves for the STCT. After the maximal number of leaves is reached, the generation process will be stopped independent from the coverage status of the module under analysis.
interpretFunction		Generate a test only for the module with the given name. If this parameter is not set all modules defined in the given source file will be interpreted.

numOfTestCases		Maximum number of test cases which will be generated. The generation process will be stopped independent from the coverage status of the module under analysis. If not set, as many as needed for the complete coverage or as many as possible (see parameters `stctMaxExp` and `stctMaxLeaves`) number of test cases will be generated.
proofMode		If set the test generator will try to prove that postconditions are not violated and will generate robustness tests (tests, which violate the precondition). Otherwise the test generator will only try to achieve 100% branch coverage of the module under test.
checkGlobalsForModification		If set the test generator will check if only allowed global variables are modified.
onlyCFG		If set only the CFG will be build and printed out. The test generation process will not be started.
memorySaveMode		If set, the stored memory models and feasibility constraints will be removed from the STCT after each completed test case.
useGlobalsInitValues		If set the initial values of global variables will be used, otherwise global variables are handled as unknown and are free for initial assignment by the generator.
arrayParameterSize		Set the size of an auxiliary array used for pointer handling. Default size is 100.
DB		Set the path to the test case database. If not set the database is considered to be empty.

Example:

```
ctgen --sourceFile cfg_ex.c --interpretFctCalls --stctMaxExp 1500
--pathForGeneratedTest $TESTPROJECT/unit_test_autogen
--interpretFunction checkAvailable
```

Examples of CTGEN Usage.

In this chapter we present simple examples of CTGEN usage to illustrate techniques discussed in the dissertation. These examples are kept small to simplify their reading and understanding.

Each example is structured as follows:

- First the source code of the module under test is presented.
- Next the source code of the test driver is presented. The test driver aims to deliver as complete branch coverage of the module under test as possible. The source code of the test driver is separated into so called test steps. Each test step corresponds to an analyzed complete path through the module under test and contains assignments of the inputs and return values of the stub functions (if any stub function was generated) of the module under test according to the values, calculated by the generator in order to satisfy the corresponding path constraint. Furthermore a test step contains a call to the module under test. In case when a specification of the module under test was given by means of the annotation language (see Chapter 3), the test step also includes assertions which correspond to this specification and indicate whether the module under test satisfies its specification or not.
- In case when a stub function was generated, its source code is presented. The stub function contains assignments of the global variables and output parameters modified by this stub function in order to satisfy the guard conditions of the module under test.
- Next the solution file for the generation process is presented. This file holds information about which paths through the module under test with which settings of the inputs were supposed to be taken according to the generator. For each listed path the calculated path constraint is reported. Furthermore the solution file contains information if the generator was able to calculate inputs to cover the module under test completely or not. The percentage of the covered transitions is listed as well as the list of uncovered transitions.
- Finally the graphical output for the generated test is demonstrated. The generator produces the graphical representation of the CFG corresponding to the GIMPLE representation of the module under test. In this representation the nodes and edges, which the generator was able to cover are drawn blue and the nodes and edges for which the generator could not find any feasible path are drawn red.

The generated test drivers and stub functions are written in RT-Tester syntax [44].

1 Overview Example

This example corresponds to the example discussed in Chapters 2 and 4 and demonstrates the overview over the CTGEN method of operation. This is the only example that lists the output produced by the

GCC plugin additionally to the generator output.

1.1 Analyzed Code

The program listed below contains the implementation of the module under test `checkAvailable()`. This function sets the global variable `rainActive` to one if and only if the global variables `rainSensor` and `rainFunction` have non-zero values. Furthermore, it sets the global variable `solarActive` to one if and only if the global variables `solarSensor` and `solarFunction` have non-zero values.

```
int rainSensor, rainFunction, rainActive;
int solarSensor, solarFunction, solarActive;

void checkAvailable(){
  if(rainSensor && rainFunction){
    rainActive = 1;
  } else {
    rainActive = 0;
  }
  if(solarSensor && solarFunction){
    solarActive = 1;
  } else {
    solarActive = 0;
  }
}
```

Inputs for this module under test are the global variables `rainSensor`, `rainFunction`, `rainActive`, `solarSensor`, `solarFunction` and `solarActive`.

1.2 GCC Plugin output

CFG Information

This section contains the code of the module under test `checkAvailable()` after processing it by the GCC plugin. The expressions of the module under test are broken down into tuples of no more than three operands [1]. Auxiliary variables are introduced to hold the temporary values needed for the transformation of complex expressions. Additionally CFG information is inserted: The source code is divided into blocks of statements executed without conditional jumps between them, conditional jumps between these blocks are defined.

```
  # BEGIN_GLOBALS
  static int solarActive;
  static int solarFunction;
  static int solarSensor;
  static int rainActive;
  static int rainFunction;
  static int rainSensor;
  # END_GLOBALS


void checkAvailable() ()
{
  # BEGIN_SCOPE_BLOCK
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  # SCOPE_BLOCK_VARS
  # END_SCOPE_BLOCK
  # BEGIN_LOCAL_HELP_DECLS
  int solarFunction.3;
  int solarSensor.2;
  int rainFunction.1;
  int rainSensor.0;
  # END_LOCAL_HELP_DECLS

  # BLOCK 2
  # PRED: ENTRY (fallthru)
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 5] rainSensor.0 = rainSensor;
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 5] if (rainSensor.0 != 0)
    goto <bb 3>;
  else
    goto <bb 5>;
  # SUCC: 3 (true) 5 (false)


  # BLOCK 3
  # PRED: 2 (true)
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 5] rainFunction.1 = rainFunction;
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 5] if (rainFunction.1 != 0)
    goto <bb 4>;
  else
    goto <bb 5>;
  # SUCC: 4 (true) 5 (false)


  # BLOCK 4
  # PRED: 3 (true)
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 6] rainActive = 1;
  goto <bb 6>;
  # SUCC: 6 (fallthru)


  # BLOCK 5
  # PRED: 2 (false) 3 (false)
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 8] rainActive = 0;
  # SUCC: 6 (fallthru)


  # BLOCK 6
  # PRED: 4 (fallthru) 5 (fallthru)
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 10] solarSensor.2 = solarSensor;
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 10] if (solarSensor.2 != 0)
    goto <bb 7>;
  else
    goto <bb 9>;
  # SUCC: 7 (true) 9 (false)


  # BLOCK 7
  # PRED: 6 (true)
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
```

```
  ["cfg_ex.c": 10] solarFunction.3 = solarFunction;
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 10] if (solarFunction.3 != 0)
    goto <bb 8>;
  else
    goto <bb 9>;
  # SUCC: 8 (true) 9 (false)


  # BLOCK 8
  # PRED: 7 (true)
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 11] solarActive = 1;
  goto <bb 10>;
  # SUCC: 10 (fallthru)


  # BLOCK 9
  # PRED: 6 (false) 7 (false)
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 13] solarActive = 0;
  # SUCC: 10 (fallthru)


  # BLOCK 10
  # PRED: 8 (fallthru) 9 (fallthru)
  # SCOPE_BLOCK_DEPTH 1 <0x40be76b4>
  ["cfg_ex.c": 15] return;
  # SUCC: EXIT


}
```

Symbol Table Information

This section contains the symbol table information of the module under test `checkAvailable()` produced by the GCC plugin. It contains the list of all used types with the specification of the name, size and other characteristics of the type. Furthermore, all declared global variables are listed with a reference to their type and location where they were declared. Finally, the list of all declared functions with the specification of the return type, function parameters and local variables is given.

```
TDGDATA{
  TYPES{
    TYPE T1{
      NAME: "int";
      SIZE: 32;
      TYPECLASS: PRIMITIVE;
      SIGNED: YES;
    }
    TYPE T0{
      NAME: "void";
      SIZE: 0;
      TYPECLASS: PRIMITIVE;
      SIGNED: YES;
    }
  }
  GLOBALS{
    VAR "rainSensor": TYPE T1 DEPTH D0<(nil)> FILE "cfg_ex.c" LINE 1;
    VAR "rainFunction": TYPE T1 DEPTH D0<(nil)> FILE "cfg_ex.c" LINE 1;
    VAR "rainActive": TYPE T1 DEPTH D0<(nil)> FILE "cfg_ex.c" LINE 1;
    VAR "solarSensor": TYPE T1 DEPTH D0<(nil)> FILE "cfg_ex.c" LINE 2;
    VAR "solarFunction": TYPE T1 DEPTH D0<(nil)> FILE "cfg_ex.c" LINE 2;
    VAR "solarActive": TYPE T1 DEPTH D0<(nil)> FILE "cfg_ex.c" LINE 2;
  }
  FUNCTIONS{
    FUNCTION "checkAvailable" FILE "cfg_ex.c" LINE 4{
      RETURNS: T0;
      LOCALS{
        VAR "rainSensor.0": TYPE T1 DEPTH D1<(nil)> FILE "cfg_ex.c" LINE 5;
        VAR "rainFunction.1": TYPE T1 DEPTH D1<(nil)> FILE "cfg_ex.c" LINE 5;
        VAR "solarSensor.2": TYPE T1 DEPTH D1<(nil)> FILE "cfg_ex.c" LINE 10;
        VAR "solarFunction.3": TYPE T1 DEPTH D1<(nil)> FILE "cfg_ex.c" LINE 10;
      }
    }
  }
}
```

1.3 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve the 100% branch coverage for the module under test `checkAvailable()`. The generator is able to achieve complete branch coverage with three cases, thus the test driver contains three test steps. Since no specification of the module under test was given, the test driver contains no assertions. `checkAvailable()` contains no defined or undefined function calls, thus no stub functions were generated.

First, the module under test and all used global variables are declared. Then in each test step assignment of these global variables is made according to the calculated values and after the setting is done, the module under test is invoked.

```
/* ================================
 * Include section
 * ================================*/


/* ================================
 * Global or static C declarations and definitions
 * ================================*/

extern void checkAvailable();
@uut void checkAvailable();

extern int rainFunction;
extern int solarFunction;
extern int rainSensor;
extern int solarSensor;

/* ================================
 * Abstract machine declaration.
 * ================================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:
```

```
@PROCESS:


    @rttBeginTestStep;  // ------------------------------------------------
    {
       rainFunction = −2147483648;
       rainFunction = −2147483648;

       solarFunction = −2147483648;
       solarFunction = −2147483648;

       rainSensor = −2147483648;
       rainSensor = −2147483648;

       solarSensor = −2147483648;
       solarSensor = −2147483648;



       @rttCall(checkAvailable());
    }
    @rttEndTestStep;  // ------------------------------------------------

    @rttBeginTestStep;  // ------------------------------------------------
    {
       rainSensor = 0;
       rainSensor = 0;

       solarSensor = 0;
       solarSensor = 0;



       @rttCall(checkAvailable());
    }
    @rttEndTestStep;  // ------------------------------------------------

    @rttBeginTestStep;  // ------------------------------------------------
    {
       rainFunction = 0;
       rainFunction = 0;

       solarFunction = 0;
       solarFunction = 0;

       rainSensor = −2147483648;
       rainSensor = −2147483648;

       solarSensor = −2147483648;
       solarSensor = −2147483648;



       @rttCall(checkAvailable());
    }
    @rttEndTestStep;  // ------------------------------------------------

}
```

1.4 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of three traces (corresponding to the three test steps in the test driver). Each trace is first specified by a list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported. In this example it is 100%.

```
SOLUTION FOR FUNCTION checkAvailable

TRACE 4
TRACE COMPLETED

TRACE
{ (rainSensor.0 = rainSensor;),
  (rainFunction.1 = rainFunction;),
  (rainActive = 1;),
  (solarSensor.2 = solarSensor;),
  (solarFunction.3 = solarFunction;),
  (solarActive = 1;),
  (checkAvailable_return;) }
  traceState:   CONT_OF_ANOTHER_TRACE
  currentStepNr: 6
  feasible: 1

CONSTRAINT:
((((((rainSensor.0@12 != 0) &&
 (rainSensor.0@12 == ((int) rainSensor@11))) &&
 (rainSensor@11 == rainSensor@0)) &&
 (rainFunction.1@13 != 0) &&
 (rainFunction.1@13 == ((int) rainFunction@12)) &&
 (rainFunction@12 == rainFunction@0)) &&
 (solarSensor.2@15 != 0) &&
 (solarSensor.2@15 == ((int) solarSensor@14)) &&
 (solarSensor@14 == solarSensor@0)) &&
 (solarFunction.3@16 != 0) &&
 (solarFunction.3@16 == ((int) solarFunction@15)) &&
 (solarFunction@15 == solarFunction@0))
SOLUTION:
    rainSensor@0 = (ConcreteLattice<signed int>, −2147483648)
    rainFunction@0 = (ConcreteLattice<signed int>, −2147483648)
    solarSensor@0 = (ConcreteLattice<signed int>, −2147483648)
    solarFunction@0 = (ConcreteLattice<signed int>, −2147483648)
    rainSensor@11 = (ConcreteLattice<signed int>, −2147483648)
    rainSensor.0@12 = (ConcreteLattice<signed int>, −2147483648)
    rainFunction@12 = (ConcreteLattice<signed int>, −2147483648)
    rainFunction.1@13 = (ConcreteLattice<signed int>, −2147483648)
    solarSensor@14 = (ConcreteLattice<signed int>, −2147483648)
    solarSensor.2@15 = (ConcreteLattice<signed int>, −2147483648)
    solarFunction@15 = (ConcreteLattice<signed int>, −2147483648)
    solarFunction.3@16 = (ConcreteLattice<signed int>, −2147483648)


TRACE 7
TRACE COMPLETED

TRACE
{ (rainSensor.0 = rainSensor;),
  (rainActive = 0;),
   (solarSensor.2 = solarSensor;),
   (solarActive = 0;),
   (checkAvailable_return;) }
   traceState:      CONT_OF_ANOTHER_TRACE
   currentStepNr: 4
   feasible: 1

CONSTRAINT:
((((rainSensor.0@12 == 0) &&
 (rainSensor.0@12 == ((int) rainSensor@11))) &&
 (rainSensor@11 == rainSensor@0)) &&
 (solarSensor.2@14 == 0) &&
 (solarSensor.2@14 == ((int) solarSensor@13)) &&
 (solarSensor@13 == solarSensor@0))
SOLUTION:
    rainSensor@0 = (ConcreteLattice<signed int>, 0)
    solarSensor@0 = (ConcreteLattice<signed int>, 0)
    rainSensor@11 = (ConcreteLattice<signed int>, 0)
    rainSensor.0@12 = (ConcreteLattice<signed int>, 0)
    solarSensor@13 = (ConcreteLattice<signed int>, 0)
    solarSensor.2@14 = (ConcreteLattice<signed int>, 0)


TRACE 9
TRACE COMPLETED

TRACE
{ (rainSensor.0 = rainSensor;),
   (rainFunction.1 = rainFunction;),
   (rainActive = 0;),
   (solarSensor.2 = solarSensor;),
   (solarFunction.3 = solarFunction;),
   (solarActive = 0;) }
   traceState:      CONT_AFTER_SOLVING
   currentStepNr: 5
   feasible: 1

CONSTRAINT:
((((((rainSensor.0@12 != 0) &&
 (rainSensor.0@12 == ((int) rainSensor@11))) &&
 (rainSensor@11 == rainSensor@0)) &&
 (rainFunction.1@13 == 0) &&
 (rainFunction.1@13 == ((int) rainFunction@12)) &&
 (rainFunction@12 == rainFunction@0)) &&
 (solarSensor.2@15 != 0) &&
 (solarSensor.2@15 == ((int) solarSensor@14)) &&
 (solarSensor@14 == solarSensor@0)) &&
 (solarFunction.3@16 == 0) &&
 (solarFunction.3@16 == ((int) solarFunction@15)) &&
 (solarFunction@15 == solarFunction@0))
SOLUTION:
    rainSensor@0 = (ConcreteLattice<signed int>, −2147483648)
    rainFunction@0 = (ConcreteLattice<signed int>, 0)
    solarSensor@0 = (ConcreteLattice<signed int>, −2147483648)
    solarFunction@0 = (ConcreteLattice<signed int>, 0)
    rainSensor@11 = (ConcreteLattice<signed int>, −2147483648)
    rainSensor.0@12 = (ConcreteLattice<signed int>, −2147483648)
    rainFunction@12 = (ConcreteLattice<signed int>, 0)
    rainFunction.1@13 = (ConcreteLattice<signed int>, 0)
    solarSensor@14 = (ConcreteLattice<signed int>, −2147483648)
    solarSensor.2@15 = (ConcreteLattice<signed int>, −2147483648)
    solarFunction@15 = (ConcreteLattice<signed int>, 0)
```

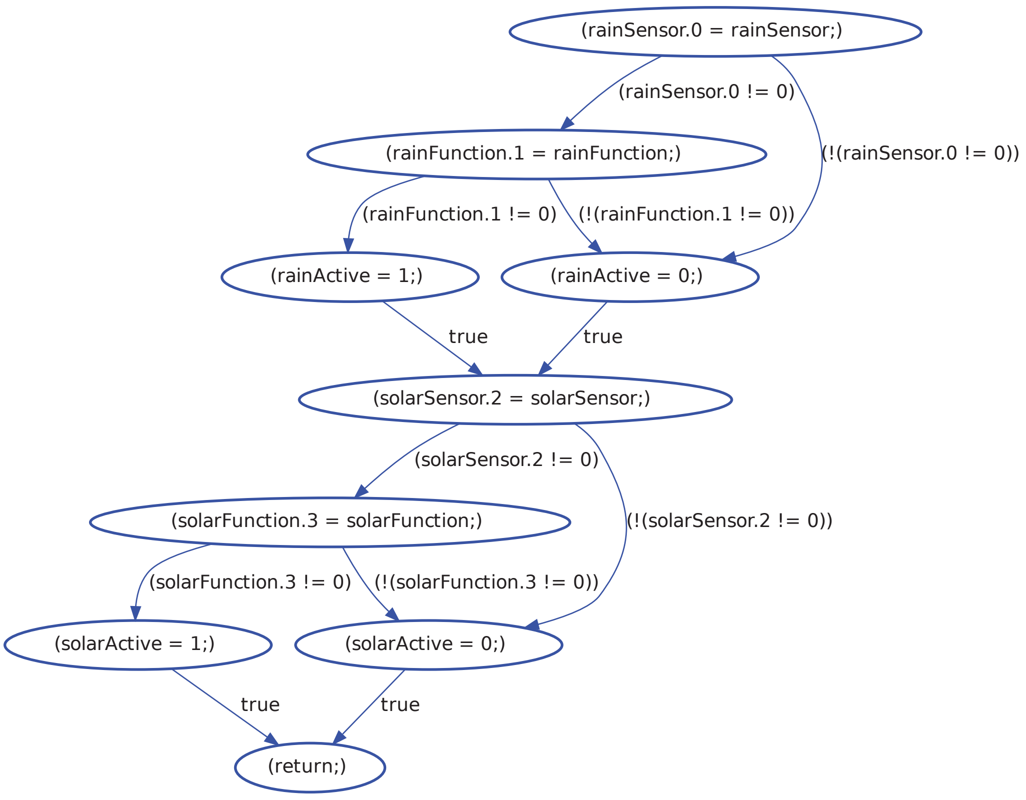


Figure 3: Graphical representation for the overview example.

```
    solarFunction.3@16 = (ConcreteLattice<signed int>, 0)


Operation checkAvailable() is covered.

Covered:
Total transitions:        100%
Transitions with guards:  100%
```

1.5 Graphical Output

Figure 3 demonstrates the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue, which indicates that all of them were successfully covered.

2 Annotation Example

This example corresponds to the example discussed in Chapter 3 and demonstrates the usage of the CTGEN annotation language.

2.1 Analyzed Code

The source code listed below contains the implementation of the module under test `alloc()`. The module under test `alloc()` returns a pointer `allocp` to `n` successive characters if there is still enough room in the buffer `allocbuf` and zero if this is not the case. First, by using `__rtt_modifies` we state that `alloc()` can only modify `allocp`, and a modification of `allocbuf` is consequently prohibited. The annotation `__rtt_precondition` specifies that the expected behaviour of `alloc()` is guaranteed only if the parameter `n` is greater as or equal to zero and `allocp` is not a `NULL`-pointer. Furthermore `__rtt_postcondition` states that after the execution of the function under test `allocp` must still be within the bounds of the array `allocbuf`. Finally, test cases are defined for situations where (a) memory can still be allocated and (b) not enough memory is available.

```
#include "ctgen_annotation.h"
#define ALLOCSIZE 1000
char allocbuf[ALLOCSIZE];
char *allocp = allocbuf;

char *alloc(int n){
  __rtt_modifies(allocp);
  __rtt_precondition(n >= 0 && allocp != 0);
  __rtt_postcondition(allocp != 0 && allocp <= allocbuf + ALLOCSIZE);
  __rtt_testcase(allocbuf + ALLOCSIZE - __rtt_initial(allocp) < n,
                 __rtt_return == 0,
                 "CTGEN_001");
  __rtt_testcase(allocbuf + ALLOCSIZE - __rtt_initial(allocp) >= n,
                 __rtt_return == __rtt_initial(allocp),
                 "CTGEN_002");

  char *retval = 0;
  if(allocbuf + ALLOCSIZE - allocp >= n){
    allocp += n;
    retval = allocp - n;
  }

  return retval;
}
```

2.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve the maximal possible branch coverage for the module under test `alloc()`. The generator was run in proof mode and tried to find counter examples for the test cases and the postcondition defined by means of the annotation language. For each of the specified test cases as well as for the postcondition a test case specification is generated. The specification of test cases contains the test case identifier, the definition of the condition and the expected result, the indication of the corresponding requirements and a short description. For each test step the test generator recognizes test cases which can be applied at the particular step and inserts an assertion corresponding to the condition before the call to the UUT and an assertion corresponding to the expected result after the call to the UUT.

```
#define ALLOCSIZE 1000
/* ===============================
 * Include section
 * ===============================*/


/* ===============================
 * Structures
 * ===============================*/


/* ===============================
 * Global or static C declarations and definitions
 * ===============================*/

extern char* alloc(int n);
@uut char* alloc(int n);

extern char allocbuf[1000];
extern char* allocp;

/* ===============================
 * Abstract machine declaration.
 * ===============================*/

@abstract machine unit_test(){

  @INIT:
  {
  }
  @FINIT:

  @PROCESS:

  char* __rtt_return;
  char* __rtt_initial_allocp_;
  int n;

  unsigned int allocp_PointsTo_offset_;

  /** @rttPrint
  * This test case evaluates, whether the function alloc
  * behaves correctly.
  * @tag TC_UNIT_TEST_AUTOGEN_ALLOC_0001
  * @condition (n >= 0 && allocp != 0)
  * @event      The unit under test 'alloc' is called.
  * @expected   (allocp != 0 && allocp <= allocbuf + ALLOCSIZE)
  * @req
  */

  /** @rttPrint
  * This test case evaluates, whether the function alloc
  * behaves correctly.
  * @tag TC_UNIT_TEST_AUTOGEN_ALLOC_0002
  * @condition allocbuf+ALLOCSIZE−__rtt_initial_allocp_<n
  * @event      The unit under test 'alloc' is called.
  * @expected   __rtt_return==0
  * @req        CTGEN_001
  */
```

```
/** @rttPrint
* This test case evaluates, whether the function alloc
* behaves correctly.
* @tag TC_UNIT_TEST_AUTOGEN_ALLOC_0003
* @condition allocbuf+ALLOCSIZE−__rtt_initial_allocp_>=n
* @event      The unit under test 'alloc' is called.
* @expected   __rtt_return==__rtt_initial_allocp_
* @req        CTGEN_002
*/

@rttBeginTestStep; // ———————————————————————————
{
  n = −2147483648;
  __rtt_initial_allocp_ = allocp;
  @rttCall(__rtt_return = alloc(n));
}
@rttEndTestStep; // ————————————————————————————

@rttBeginTestStep; // ———————————————————————————
{
  allocp = NULL;
  allocp = NULL;
  n = 0;
  __rtt_initial_allocp_ = allocp;
  @rttCall(__rtt_return = alloc(n));
}
@rttEndTestStep; // ————————————————————————————

@rttBeginTestStep; // ———————————————————————————
{
  n = 1073742735;
  allocp = allocbuf;
  allocp_PointsTo_offset_ = 72;
  allocp += allocp_PointsTo_offset_;
  __rtt_initial_allocp_ = allocp;
  @rttAssert((n >= 0 && allocp != 0), "TC_UNIT_TEST_AUTOGEN_ALLOC_0001");
  @rttAssert(allocbuf+ALLOCSIZE−__rtt_initial_allocp_<n, "
      TC_UNIT_TEST_AUTOGEN_ALLOC_0002");
  @rttCall(__rtt_return = alloc(n));
  @rttAssert((allocp != 0 && allocp <= allocbuf + ALLOCSIZE), "
      TC_UNIT_TEST_AUTOGEN_ALLOC_0001");
  @rttAssert(__rtt_return==0, "TC_UNIT_TEST_AUTOGEN_ALLOC_0002");
}
@rttEndTestStep; // ————————————————————————————

@rttBeginTestStep; // ———————————————————————————
{
  n = 0;

  allocp = allocbuf;
  allocp_PointsTo_offset_ = 99;
  allocp += allocp_PointsTo_offset_;
  __rtt_initial_allocp_ = allocp;
  @rttAssert((n >= 0 && allocp != 0), "TC_UNIT_TEST_AUTOGEN_ALLOC_0001");
  @rttAssert(allocbuf+ALLOCSIZE−__rtt_initial_allocp_>=n, "
      TC_UNIT_TEST_AUTOGEN_ALLOC_0003");
  @rttCall(__rtt_return = alloc(n));
  @rttAssert((allocp != 0 && allocp <= allocbuf + ALLOCSIZE), "
      TC_UNIT_TEST_AUTOGEN_ALLOC_0001");
  @rttAssert(__rtt_return==__rtt_initial_allocp_, "TC_UNIT_TEST_AUTOGEN_ALLOC_0003");
}
```

```
    @rttEndTestStep; // ------------------------------------------------
}
```

2.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of four traces (corresponding to the four test steps in the test driver). Each trace is first specified by the list of it's statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

Since the test generator for this example was running in proof mode, it reports the coverage for the control flow graph containing pre- and postconditions as well as their violations. And although the announced coverage is only 77%, the generator was able to achieve 100% branch coverage of the UUT. The analysis of uncovered transitions listed at the bottom of the solution file states that all uncovered transitions correspond to the violations of the test cases or postcondition respectively.

```
SOLUTION FOR FUNCTION alloc

TRACE 10
TRACE COMPLETED

TRACE
{ (__rtt_modifies__((&"allocp"));),
  (__rtt_precondition_begin__();),
  (return;) }
  traceState:     CONT_AFTER_SOLVING
  currentStepNr: 2
  feasible: 1

CONSTRAINT:
((n@61 < 0) &&
 (n@61 == ((int) n@0)))
SOLUTION:
   n@0 = (ConcreteLattice<signed int>, -2147483648)
   n@61 = (ConcreteLattice<signed int>, -2147483648)


TRACE 11
TRACE COMPLETED

TRACE
{ (__rtt_modifies__((&"allocp"));),
  (__rtt_precondition_begin__();),
  (allocp_0 = allocp;),
  (return;) }
  traceState:     CONT_AFTER_SOLVING
  currentStepNr: 3
  feasible: 1

CONSTRAINT:
(((n@61 >= 0) &&
 (n@61 == ((int) n@0))) &&
 (allocp_0@baseAddr@62 == 0) &&
 (allocp_0@baseAddr@62 == allocp@offset@61) &&
 (allocp@baseAddr@61 == allocp@baseAddr@0) &&
 (allocp@offset@61 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@61 == allocp@offset@0) &&
 (allocp@offset@61 < 100) &&
 (allocp_0@baseAddr@62 == allocp@baseAddr@61))
SOLUTION:
    n@0 = (ConcreteLattice<signed int>, 0)
    allocp@baseAddr@0 = (ConcreteLattice<unsigned int>, 0)
    allocp@offset@0 = (ConcreteLattice<unsigned int>, 0)
    n@61 = (ConcreteLattice<signed int>, 0)
    allocp@baseAddr@61 = (ConcreteLattice<unsigned int>, 0)
    allocp@offset@61 = (ConcreteLattice<unsigned int>, 0)
    allocp_0@baseAddr@62 = (ConcreteLattice<unsigned int>, 0)


TRACE 36
TRACE COMPLETED

TRACE
{ (__rtt_modifies__((&"allocp"));),
   (__rtt_precondition_begin__();),
   (allocp_0 = allocp;),
   (__rtt_precondition_end__();),
   (retval_0x40bf1270 = 0;),
   (D_1759 = ((&allocbuf[0]) + 1000);),
   (D_1760 = ((int) D_1759);),
   (allocp_1 = allocp;),
   (allocp_2 = ((int) allocp_1);),
   (D_1763 = (D_1760 - allocp_2);),
   (__rtt_postcondition_begin__();),
   (allocp_8 = allocp;),
   (allocp_9 = allocp;),
   (D_1778 = ((&allocbuf[0]) + 1000);),
   (__rtt_postcondition_end__();),
   (__rtt_testcase__((&"(n␣>=␣0␣&&␣allocp␣!=␣0)"), (&"(allocp␣!=␣0␣&&␣allocp␣<=␣allocbuf␣+␣
        ALLOCSIZE)"), (&""));),
   (D_1781 = ((&allocbuf[0]) + 1000);),
   (D_1782 = ((int) D_1781);),
   (allocp_11 = allocp;),
   (D_1784 = __rtt_initial(allocp_11@82);),
   (D_1785 = ((int) D_1784);),
   (D_1786 = (D_1782 - D_1785);),
   (retval_10 = (D_1786 < n);),
   (<EMPTYSTATEMENT>;),
   (__rtt_testcase__((&"allocbuf+ALLOCSIZE-__rtt_initial(allocp)<n"), (&"__rtt_return==0"),
         (&"CTGEN_001"));),
   (D_1794 = ((&allocbuf[0]) + 1000);),
   (D_1795 = ((int) D_1794);),
   (allocp_13 = allocp;),
   (D_1797 = __rtt_initial(allocp_13@89);),
   (D_1798 = ((int) D_1797);),
   (D_1799 = (D_1795 - D_1798);),
   (retval_12 = (D_1799 >= n);),
   (D_1809 = retval_0x40bf1270;),
   (return = D_1809;) }
   traceState:    CONT_OF_ANOTHER_TRACE
   currentStepNr: 33
   feasible: 1

CONSTRAINT:
(((((((((alloc@61 >= 0) &&
 (alloc@61 == ((int) alloc@0))) &&
 (allocp_0@baseAddr@62 != 0) &&
 (allocp_0@baseAddr@62 == allocp@offset@61) &&
 (allocp@baseAddr@61 == allocp@baseAddr@0) &&
 (allocp@offset@61 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@61 == allocp@offset@0) &&
 (allocp@offset@61 < 100) &&
 (allocp_0@baseAddr@62 == allocp@baseAddr@61)) &&
 (D_1763@68 < alloc@68) &&
 (D_1763@68 == ((int) (D_1760@67 - allocp_2@67))) &&
 (alloc@68 == ((int) alloc@0)) &&
 (D_1760@67 == ((int) ((int) D_1759@offset@64))) &&
 (allocp_2@67 == ((int) ((int) allocp_1@offset@66))) &&
 (D_1759@baseAddr@64 == allocbuf@baseAddr@63) &&
 (allocbuf@offset@63 == 0) &&
 (allocbuf@offset@63 < 1000) &&
 (D_1759@offset@64 == (allocbuf@offset@63 + 1000)) &&
 (allocp_1@offset@66 == allocp@offset@65) &&
 (allocbuf@offset@63 == allocbuf@offset@0) &&
 (allocp@baseAddr@65 == allocp@baseAddr@0) &&
 (allocp@offset@65 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@65 == allocp@offset@0) &&
 (allocp@offset@65 < 100) &&
 (allocbuf@offset@63 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (D_1759@baseAddr@64 == allocp_1@baseAddr@66) &&
 (allocp_1@baseAddr@66 == allocp@baseAddr@65) &&
 (allocp@baseAddr@65 == allocbuf@baseAddr@63) &&
 (allocbuf@baseAddr@63 == allocbuf@baseAddr@0)) &&
 (allocp_8@baseAddr@69 != 0) &&
 (allocp_8@baseAddr@69 == allocp@offset@68) &&
 (allocp@baseAddr@68 == allocp@baseAddr@0) &&
 (allocp@offset@68 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@68 == allocp@offset@0) &&
 (allocp@offset@68 < 100) &&
 (allocp_8@baseAddr@69 == allocp@baseAddr@68)) &&
 (allocp_9@offset@71 <= D_1778@offset@71) &&
 (allocp_9@offset@71 == allocp@offset@69) &&
 (D_1778@baseAddr@71 == allocbuf@baseAddr@70) &&
 (allocbuf@offset@70 == 0) &&
 (allocbuf@offset@70 < 1000) &&
 (D_1778@offset@71 == (allocbuf@offset@70 + 1000)) &&
 (allocp@baseAddr@69 == allocp@baseAddr@0) &&
 (allocp@offset@69 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@69 == allocp@offset@0) &&
 (allocbuf@offset@70 == allocbuf@offset@0) &&
 (allocp@offset@69 < 100) &&
 (allocbuf@offset@70 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (allocp_9@baseAddr@71 == D_1778@baseAddr@71) &&
 (D_1778@baseAddr@71 == allocp@baseAddr@69) &&
 (allocp@baseAddr@69 == allocbuf@baseAddr@70) &&
 (allocbuf@baseAddr@70 == allocbuf@baseAddr@0)) &&
 (retval_10@78 != 0) &&
 (retval_10@78 == ((bool) (D_1786@77 < alloc@77))) &&
 (D_1786@77 == ((int) (D_1782@76 - D_1785@76))) &&
 (alloc@77 == ((int) alloc@0)) &&
 (D_1782@76 == ((int) ((int) D_1781@offset@72))) &&
 (D_1785@76 == ((int) ((int) D_1784@offset@75))) &&
 (D_1781@baseAddr@72 == allocbuf@baseAddr@71) &&
 (allocbuf@offset@71 == 0) &&
 (allocbuf@offset@71 < 1000) &&
 (D_1781@offset@72 == (allocbuf@offset@71 + 1000)) &&
 (D_1784@offset@75 == allocp@offset@0) &&
 (allocbuf@offset@71 == allocbuf@offset@0) &&
 (allocp@offset@0 < 100) &&
 (allocbuf@offset@71 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (D_1781@baseAddr@72 == D_1784@baseAddr@75) &&
 (D_1784@baseAddr@75 == allocp@baseAddr@0) &&
 (allocp@baseAddr@0 == allocbuf@baseAddr@71) &&
 (allocbuf@baseAddr@71 == allocbuf@baseAddr@0)) &&
 (__rtt_return_0x40bf12d8@baseAddr@78 == 0)) &&
 (retval_12@85 == 0) &&
 (retval_12@85 == ((bool) (D_1799@84 >= alloc@84))) &&
 (D_1799@84 == ((int) (D_1795@83 - D_1798@83))) &&
 (alloc@84 == ((int) alloc@0)) &&
 (D_1795@83 == ((int) ((int) D_1794@offset@79))) &&
 (D_1798@83 == ((int) ((int) D_1797@offset@82))) &&
 (D_1794@baseAddr@79 == allocbuf@baseAddr@78) &&
 (allocbuf@offset@78 == 0) &&
 (allocbuf@offset@78 < 1000) &&
 (D_1794@offset@79 == (allocbuf@offset@78 + 1000)) &&
 (D_1797@offset@82 == allocp@offset@0) &&
 (allocbuf@offset@78 == allocbuf@offset@0) &&
 (allocp@offset@0 < 100) &&
 (allocbuf@offset@78 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (D_1794@baseAddr@79 == D_1797@baseAddr@82) &&
 (D_1797@baseAddr@82 == allocp@baseAddr@0) &&
 (allocp@baseAddr@0 == allocbuf@baseAddr@78) &&
 (allocbuf@baseAddr@78 == allocbuf@baseAddr@0)) &&
 (alloc@61 >= 0) &&
 (alloc@61 == ((int) alloc@0)) &&
 (allocp_0@baseAddr@62 != 0) &&
 (allocp_0@baseAddr@62 == allocp@offset@61) &&
 (allocp@baseAddr@61 == allocp@baseAddr@0) &&
 (allocp@offset@61 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@61 == allocp@offset@0) &&
 (allocp@offset@61 < 100) &&
 (allocp_0@baseAddr@62 == allocp@baseAddr@61) &&
 (D_1763@68 < alloc@68) &&
 (D_1763@68 == ((int) (D_1760@67 - allocp_2@67))) &&
 (alloc@68 == ((int) alloc@0)) &&
 (D_1760@67 == ((int) ((int) D_1759@offset@64))) &&
 (allocp_2@67 == ((int) ((int) allocp_1@offset@66))) &&
 (D_1759@baseAddr@64 == allocbuf@baseAddr@63) &&
 (allocbuf@offset@63 == 0) &&
 (allocbuf@offset@63 < 1000) &&
 (D_1759@offset@64 == (allocbuf@offset@63 + 1000)) &&
 (allocp_1@offset@66 == allocp@offset@65) &&
 (allocbuf@offset@63 == allocbuf@offset@0) &&
 (allocp@baseAddr@65 == allocp@baseAddr@0) &&
 (allocp@offset@65 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@65 == allocp@offset@0) &&
 (allocp@offset@65 < 100) &&
 (allocbuf@offset@63 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (D_1759@baseAddr@64 == allocp_1@baseAddr@66) &&
 (allocp_1@baseAddr@66 == allocp@baseAddr@65) &&
 (allocp@baseAddr@65 == allocbuf@baseAddr@63) &&
 (allocbuf@baseAddr@63 == allocbuf@baseAddr@0) &&
 (allocp_8@baseAddr@69 != 0) &&
 (allocp_8@baseAddr@69 == allocp@offset@68) &&
 (allocp@baseAddr@68 == allocp@baseAddr@0) &&
 (allocp@offset@68 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@68 == allocp@offset@0) &&
 (allocp@offset@68 < 100) &&
 (allocp_8@baseAddr@69 == allocp@baseAddr@68) &&
 (allocp_9@offset@71 <= D_1778@offset@71) &&
 (allocp_9@offset@71 == allocp@offset@69) &&
 (D_1778@baseAddr@71 == allocbuf@baseAddr@70) &&
 (allocbuf@offset@70 == 0) &&
 (allocbuf@offset@70 < 1000) &&
 (D_1778@offset@71 == (allocbuf@offset@70 + 1000)) &&
 (allocp@baseAddr@69 == allocp@baseAddr@0) &&
 (allocp@offset@69 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@69 == allocp@offset@0) &&
 (allocbuf@offset@70 == allocbuf@offset@0) &&
 (allocp@offset@69 < 100) &&
 (allocbuf@offset@70 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (allocp_9@baseAddr@71 == D_1778@baseAddr@71) &&
 (D_1778@baseAddr@71 == allocp@baseAddr@69) &&
 (allocp@baseAddr@69 == allocbuf@baseAddr@70) &&
 (allocbuf@baseAddr@70 == allocbuf@baseAddr@0) &&
 (retval_10@78 != 0) &&
 (retval_10@78 == ((bool) (D_1786@77 < alloc@77))) &&
 (D_1786@77 == ((int) (D_1782@76 - D_1785@76))) &&
 (alloc@77 == ((int) alloc@0)) &&
 (D_1782@76 == ((int) ((int) D_1781@offset@72))) &&
 (D_1785@76 == ((int) ((int) D_1784@offset@75))) &&
 (D_1781@baseAddr@72 == allocbuf@baseAddr@71) &&
 (allocbuf@offset@71 == 0) &&
 (allocbuf@offset@71 < 1000) &&
 (D_1781@offset@72 == (allocbuf@offset@71 + 1000)) &&
 (D_1784@offset@75 == allocp@offset@0) &&
 (allocbuf@offset@71 == allocbuf@offset@0) &&
 (allocp@offset@0 < 100) &&
 (allocbuf@offset@71 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (D_1781@baseAddr@72 == D_1784@baseAddr@75) &&
 (D_1784@baseAddr@75 == allocp@baseAddr@0) &&
 (allocp@baseAddr@0 == allocbuf@baseAddr@71) &&
 (allocbuf@baseAddr@71 == allocbuf@baseAddr@0) &&
 (__rtt_return_0x40bf12d8@baseAddr@78 == 0) &&
 (retval_12@85 == 0) &&
 (retval_12@85 == ((bool) (D_1799@84 >= alloc@84))) &&
 (D_1799@84 == ((int) (D_1795@83 - D_1798@83))) &&
 (alloc@84 == ((int) alloc@0)) &&
 (D_1795@83 == ((int) ((int) D_1794@offset@79))) &&
 (D_1798@83 == ((int) ((int) D_1797@offset@82))) &&
 (D_1794@baseAddr@79 == allocbuf@baseAddr@78) &&
 (allocbuf@offset@78 == 0) &&
 (allocbuf@offset@78 < 1000) &&
 (D_1794@offset@79 == (allocbuf@offset@78 + 1000)) &&
 (D_1797@offset@82 == allocp@offset@0) &&
 (allocbuf@offset@78 == allocbuf@offset@0) &&
 (allocp@offset@0 < 100) &&
 (allocbuf@offset@78 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
  (D_1794@baseAddr@79 == D_1797@baseAddr@82) &&
  (D_1797@baseAddr@82 == allocp@baseAddr@0) &&
  (allocp@baseAddr@0 == allocbuf@baseAddr@78) &&
  (allocbuf@baseAddr@78 == allocbuf@baseAddr@0) &&
  (__rtt_return_0x40bf12d8@baseAddr@78 == return@offset@87) &&
  (return@offset@87 == D_1809@offset@86) &&
  (D_1809@offset@86 == retval_0x40bf1270@offset@85) &&
  (retval_0x40bf1270@offset@85 == 0) &&
  (return@baseAddr@87 == D_1809@baseAddr@86) &&
  (D_1809@baseAddr@86 == retval_0x40bf1270@baseAddr@85))
SOLUTION:
    alloc@0 = (ConcreteLattice<signed int>, 1073742735)
    allocp@baseAddr@0 = (ConcreteLattice<unsigned int>, 72)
    allocp@offset@0 = (ConcreteLattice<unsigned int>, 72)
    allocbuf@baseAddr@0 = (ConcreteLattice<unsigned int>, 72)
    allocbuf@offset@0 = (ConcreteLattice<unsigned int>, 0)
    alloc@61 = (ConcreteLattice<signed int>, 1073742735)
    allocp@baseAddr@61 = (ConcreteLattice<unsigned int>, 72)
    allocp@offset@61 = (ConcreteLattice<unsigned int>, 72)
    allocp_0@baseAddr@62 = (ConcreteLattice<unsigned int>, 72)
    allocbuf@baseAddr@63 = (ConcreteLattice<unsigned int>, 72)
    allocbuf@offset@63 = (ConcreteLattice<unsigned int>, 0)
    D_1759@baseAddr@64 = (ConcreteLattice<unsigned int>, 72)
    D_1759@offset@64 = (ConcreteLattice<unsigned int>, 1000)
    allocp@baseAddr@65 = (ConcreteLattice<unsigned int>, 72)
    allocp@offset@65 = (ConcreteLattice<unsigned int>, 72)
    allocp_1@baseAddr@66 = (ConcreteLattice<unsigned int>, 72)
    allocp_1@offset@66 = (ConcreteLattice<unsigned int>, 72)
    D_1760@67 = (ConcreteLattice<signed int>, 1000)
    allocp_2@67 = (ConcreteLattice<signed int>, 72)
    alloc@68 = (ConcreteLattice<signed int>, 1073742735)
    allocp@baseAddr@68 = (ConcreteLattice<unsigned int>, 72)
    allocp@offset@68 = (ConcreteLattice<unsigned int>, 72)
    D_1763@68 = (ConcreteLattice<signed int>, 928)
    allocp@baseAddr@69 = (ConcreteLattice<unsigned int>, 72)
    allocp@offset@69 = (ConcreteLattice<unsigned int>, 72)
    allocp_8@baseAddr@69 = (ConcreteLattice<unsigned int>, 72)
    allocbuf@baseAddr@70 = (ConcreteLattice<unsigned int>, 72)
    allocbuf@offset@70 = (ConcreteLattice<unsigned int>, 0)
    allocbuf@baseAddr@71 = (ConcreteLattice<unsigned int>, 72)
    allocbuf@offset@71 = (ConcreteLattice<unsigned int>, 0)
    D_1778@baseAddr@71 = (ConcreteLattice<unsigned int>, 72)
    D_1778@offset@71 = (ConcreteLattice<unsigned int>, 1000)
    allocp_9@baseAddr@71 = (ConcreteLattice<unsigned int>, 72)
    allocp_9@offset@71 = (ConcreteLattice<unsigned int>, 72)
    D_1781@baseAddr@72 = (ConcreteLattice<unsigned int>, 72)
    D_1781@offset@72 = (ConcreteLattice<unsigned int>, 1000)
    D_1784@baseAddr@75 = (ConcreteLattice<unsigned int>, 72)
    D_1784@offset@75 = (ConcreteLattice<unsigned int>, 72)
    D_1782@76 = (ConcreteLattice<signed int>, 1000)
    D_1785@76 = (ConcreteLattice<signed int>, 72)
    alloc@77 = (ConcreteLattice<signed int>, 1073742735)
    D_1786@77 = (ConcreteLattice<signed int>, 928)
    allocbuf@baseAddr@78 = (ConcreteLattice<unsigned int>, 72)
    allocbuf@offset@78 = (ConcreteLattice<unsigned int>, 0)
    retval_10@78 = (ConcreteLattice<bool>, 1)
    __rtt_return_0x40bf12d8@baseAddr@78 = (ConcreteLattice<unsigned int>, 0)
    D_1794@baseAddr@79 = (ConcreteLattice<unsigned int>, 72)
    D_1794@offset@79 = (ConcreteLattice<unsigned int>, 1000)
    D_1797@baseAddr@82 = (ConcreteLattice<unsigned int>, 72)
    D_1797@offset@82 = (ConcreteLattice<unsigned int>, 72)
    D_1795@83 = (ConcreteLattice<signed int>, 1000)
    D_1798@83 = (ConcreteLattice<signed int>, 72)
    alloc@84 = (ConcreteLattice<signed int>, 1073742735)
    D_1799@84 = (ConcreteLattice<signed int>, 928)
    retval_12@85 = (ConcreteLattice<bool>, 0)
    retval_0x40bf1270@baseAddr@85 = (ConcreteLattice<unsigned int>, 0)
    retval_0x40bf1270@offset@85 = (ConcreteLattice<unsigned int>, 0)
    D_1809@baseAddr@86 = (ConcreteLattice<unsigned int>, 0)
    D_1809@offset@86 = (ConcreteLattice<unsigned int>, 0)
    return@baseAddr@87 = (ConcreteLattice<unsigned int>, 0)
    return@offset@87 = (ConcreteLattice<unsigned int>, 0)

TRACE 38
TRACE COMPLETED

TRACE
{ (__rtt_modifies__((&"allocp"));),
  (__rtt_precondition_begin__();),
  (allocp_0 = allocp;),
  (__rtt_precondition_end__();),
  (retval_0x40bf1270 = 0;),
  (D_1759 = ((&allocbuf[0]) + 1000);),
  (D_1760 = ((int) D_1759);),
  (allocp_1 = allocp;),
  (allocp_2 = ((int) allocp_1);),
  (D_1763 = (D_1760 - allocp_2);),
  (allocp_3 = allocp;),
  (n_4 = ((unsigned int) n);),
  (allocp_5 = (allocp_3 + n_4);),
  (allocp = allocp_5;),
  (allocp_6 = allocp;),
  (n_7 = ((unsigned int) n);),
  (D_1771 = (-n_7);),
  (retval_0x40bf1270 = (allocp_6 + D_1771);),
  (__rtt_postcondition_begin__();),
  (allocp_8 = allocp;),
  (allocp_9 = allocp;),
  (D_1778 = ((&allocbuf[0]) + 1000);),
  (__rtt_postcondition_end__();),
  (__rtt_testcase__((&"(n␣>=␣0␣&&␣allocp␣!=␣0)"), (&"(allocp␣!=␣0␣&&␣allocp␣<=␣allocbuf␣+␣
      ALLOCSIZE)"), (&""));),
  (D_1781 = ((&allocbuf[0]) + 1000);),
  (D_1782 = ((int) D_1781);),
  (allocp_11 = allocp;),
  (D_1784 = __rtt_initial(allocp_11@82);),
  (D_1785 = ((int) D_1784);),
  (D_1786 = (D_1782 - D_1785);),
  (retval_10 = (D_1786 < n);),
  (D_1794 = ((&allocbuf[0]) + 1000);),
  (D_1795 = ((int) D_1794);),
  (allocp_13 = allocp;),
  (D_1797 = __rtt_initial(allocp_13@89);),
  (D_1798 = ((int) D_1797);),
  (D_1799 = (D_1795 - D_1798);),
  (retval_12 = (D_1799 >= n);),
  (allocp_15 = allocp;),
  (D_1804 = __rtt_initial(allocp_15@94);),
  (retval_14 = (D_1804 != __rtt_return_0x40bf12d8);),
  (__rtt_testcase__((&"allocbuf+ALLOCSIZE-__rtt_initial(allocp)>=n"), (&"__rtt_return==
      __rtt_initial(allocp)"), (&"CTGEN_002"));),
  (D_1809 = retval_0x40bf1270;),
  (return = D_1809;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 43
  feasible: 1

CONSTRAINT:
(((((((((alloc@61 >= 0) &&
 (alloc@61 == ((int) alloc@0))) &&
 (allocp_0@baseAddr@62 != 0) &&
 (allocp_0@baseAddr@62 == allocp@offset@61) &&
 (allocp@baseAddr@61 == allocp@baseAddr@0) &&
 (allocp@offset@61 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@61 == allocp@offset@0) &&
 (allocp@offset@61 < 100) &&
 (allocp_0@baseAddr@62 == allocp@baseAddr@61)) &&
 (D_1763@68 >= alloc@68) &&
 (D_1763@68 == ((int) (D_1760@67 - allocp_2@67))) &&
 (alloc@68 == ((int) alloc@0)) &&
 (D_1760@67 == ((int) ((int) D_1759@offset@64))) &&
 (allocp_2@67 == ((int) ((int) allocp_1@offset@66))) &&
 (D_1759@baseAddr@64 == allocbuf@baseAddr@63) &&
 (allocbuf@offset@63 == 0) &&
 (allocbuf@offset@63 < 1000) &&
 (D_1759@offset@64 == (allocbuf@offset@63 + 1000)) &&
 (allocp_1@offset@66 == allocp@offset@65) &&
 (allocbuf@offset@63 == allocbuf@offset@0) &&
 (allocp@baseAddr@65 == allocp@baseAddr@0) &&
 (allocp@offset@65 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@65 == allocp@offset@0) &&
 (allocp@offset@65 < 100) &&
 (allocbuf@offset@63 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (D_1759@baseAddr@64 == allocp_1@baseAddr@66) &&
 (allocp_1@baseAddr@66 == allocp@baseAddr@65) &&
 (allocp@baseAddr@65 == allocbuf@baseAddr@63) &&
 (allocbuf@baseAddr@63 == allocbuf@baseAddr@0)) &&
 (allocp_8@baseAddr@77 != 0) &&
 (allocp_8@baseAddr@77 == allocp@offset@76) &&
 (allocp@offset@76 == allocp_5@offset@71) &&
 (allocp_5@offset@71 == (allocp_3@offset@70 + n_4@70)) &&
 (allocp_3@offset@70 == allocp@offset@68) &&
 (n_4@70 == ((unsigned int) ((unsigned int) alloc@69))) &&
 (allocp@baseAddr@68 == allocp@baseAddr@0) &&
 (allocp@offset@68 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@68 == allocp@offset@0) &&
 (alloc@69 == ((int) alloc@0)) &&
 (allocp@offset@76 < 100) &&
 (allocp@offset@68 < 100) &&
 (allocp_8@baseAddr@77 == allocp@baseAddr@76) &&
 (allocp@baseAddr@76 == allocp_5@baseAddr@71) &&
 (allocp_5@baseAddr@71 == allocp_3@baseAddr@70) &&
 (allocp_3@baseAddr@70 == allocp@baseAddr@68)) &&
 (allocp_9@offset@79 <= D_1778@offset@79) &&
 (allocp_9@offset@79 == allocp@offset@77) &&
 (D_1778@baseAddr@79 == allocbuf@baseAddr@78) &&
 (allocbuf@offset@78 == 0) &&
 (allocbuf@offset@78 < 1000) &&
 (D_1778@offset@79 == (allocbuf@offset@78 + 1000)) &&
 (allocp@offset@77 == allocp_5@offset@71) &&
 (allocbuf@offset@78 == allocbuf@offset@0) &&
 (allocp@offset@77 < 100) &&
 (allocbuf@offset@78 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (allocp_9@baseAddr@79 == D_1778@baseAddr@79) &&
 (D_1778@baseAddr@79 == allocp@baseAddr@77) &&
 (allocp@baseAddr@77 == allocp_5@baseAddr@71) &&
 (allocp_5@baseAddr@71 == allocbuf@baseAddr@78) &&
 (allocbuf@baseAddr@78 == allocbuf@baseAddr@0)) &&
 (retval_10@86 == 0) &&
 (retval_10@86 == ((bool) (D_1786@85 < alloc@85))) &&
 (D_1786@85 == ((int) (D_1782@84 - D_1785@84))) &&
 (alloc@85 == ((int) alloc@0)) &&
 (D_1782@84 == ((int) ((int) D_1781@offset@80))) &&
 (D_1785@84 == ((int) ((int) D_1784@offset@83))) &&
 (D_1781@baseAddr@80 == allocbuf@baseAddr@79) &&
 (allocbuf@offset@79 == 0) &&
 (allocbuf@offset@79 < 1000) &&
 (D_1781@offset@80 == (allocbuf@offset@79 + 1000)) &&
 (D_1784@offset@83 == allocp@offset@0) &&
 (allocbuf@offset@79 == allocbuf@offset@0) &&
 (allocp@offset@0 < 100) &&
 (allocbuf@offset@79 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (D_1781@baseAddr@80 == D_1784@baseAddr@83) &&
 (D_1784@baseAddr@83 == allocp@baseAddr@0) &&
 (allocp@baseAddr@0 == allocbuf@baseAddr@79) &&
 (allocbuf@baseAddr@79 == allocbuf@baseAddr@0)) &&
 (retval_12@93 != 0) &&
 (retval_12@93 == ((bool) (D_1799@92 >= alloc@92))) &&
 (D_1799@92 == ((int) (D_1795@91 - D_1798@91))) &&
 (alloc@92 == ((int) alloc@0)) &&
 (D_1795@91 == ((int) ((int) D_1794@offset@87))) &&
 (D_1798@91 == ((int) ((int) D_1797@offset@90))) &&
 (D_1794@baseAddr@87 == allocbuf@baseAddr@86) &&
 (allocbuf@offset@86 == 0) &&
 (allocbuf@offset@86 < 1000) &&
 (D_1794@offset@87 == (allocbuf@offset@86 + 1000)) &&
 (D_1797@offset@90 == allocp@offset@0) &&
 (allocbuf@offset@86 == allocbuf@offset@0) &&
 (allocp@offset@0 < 100) &&
 (allocbuf@offset@86 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (D_1794@baseAddr@87 == D_1797@baseAddr@90) &&
 (D_1797@baseAddr@90 == allocp@baseAddr@0) &&
 (allocp@baseAddr@0 == allocbuf@baseAddr@86) &&
 (allocbuf@baseAddr@86 == allocbuf@baseAddr@0)) &&
 (retval_14@96 == 0) &&
 (retval_14@96 == ((bool) (D_1804@offset@95 != __rtt_return_0x40bf12d8@offset@95))) &&
 (D_1804@offset@95 == allocp@offset@0) &&
 (allocp@offset@0 < 100) &&
 (D_1804@baseAddr@95 == __rtt_return_0x40bf12d8@baseAddr@95) &&
 (__rtt_return_0x40bf12d8@baseAddr@95 == allocp@baseAddr@0)) &&
 (alloc@61 >= 0) &&
 (alloc@61 == ((int) alloc@0)) &&
 (allocp_0@baseAddr@62 != 0) &&
 (allocp_0@baseAddr@62 == allocp@offset@61) &&
 (allocp@baseAddr@61 == allocp@baseAddr@0) &&
 (allocp@offset@61 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@61 == allocp@offset@0) &&
 (allocp@offset@61 < 100) &&
 (allocp_0@baseAddr@62 == allocp@baseAddr@61) &&
 (D_1763@68 >= alloc@68) &&
 (D_1763@68 == ((int) (D_1760@67 - allocp_2@67))) &&
 (alloc@68 == ((int) alloc@0)) &&
 (D_1760@67 == ((int) ((int) D_1759@offset@64))) &&
 (allocp_2@67 == ((int) ((int) allocp_1@offset@66))) &&
 (D_1759@baseAddr@64 == allocbuf@baseAddr@63) &&
 (allocbuf@offset@63 == 0) &&
 (allocbuf@offset@63 < 1000) &&
 (D_1759@offset@64 == (allocbuf@offset@63 + 1000)) &&
 (allocp_1@offset@66 == allocp@offset@65) &&
 (allocbuf@offset@63 == allocbuf@offset@0) &&
 (allocp@baseAddr@65 == allocp@baseAddr@0) &&
 (allocp@offset@65 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@65 == allocp@offset@0) &&
 (allocp@offset@65 < 100) &&
 (allocbuf@offset@63 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (D_1759@baseAddr@64 == allocp_1@baseAddr@66) &&
 (allocp_1@baseAddr@66 == allocp@baseAddr@65) &&
 (allocp@baseAddr@65 == allocbuf@baseAddr@63) &&
 (allocbuf@baseAddr@63 == allocbuf@baseAddr@0) &&
 (allocp_8@baseAddr@77 != 0) &&
 (allocp_8@baseAddr@77 == allocp@offset@76) &&
 (allocp@offset@76 == allocp_5@offset@71) &&
 (allocp_5@offset@71 == (allocp_3@offset@70 + n_4@70)) &&
 (allocp_3@offset@70 == allocp@offset@68) &&
 (n_4@70 == ((unsigned int) ((unsigned int) alloc@69))) &&
 (allocp@baseAddr@68 == allocp@baseAddr@0) &&
 (allocp@offset@68 < 100) &&
 (allocp@offset@0 < 100) &&
 (allocp@offset@68 == allocp@offset@0) &&
 (alloc@69 == ((int) alloc@0)) &&
 (allocp@offset@76 < 100) &&
 (allocp@offset@68 < 100) &&
 (allocp_8@baseAddr@77 == allocp@baseAddr@76) &&
 (allocp@baseAddr@76 == allocp_5@baseAddr@71) &&
 (allocp_5@baseAddr@71 == allocp_3@baseAddr@70) &&
 (allocp_3@baseAddr@70 == allocp@baseAddr@68) &&
 (allocp_9@offset@79 <= D_1778@offset@79) &&
 (allocp_9@offset@79 == allocp@offset@77) &&
 (D_1778@baseAddr@79 == allocbuf@baseAddr@78) &&
 (allocbuf@offset@78 == 0) &&
 (allocbuf@offset@78 < 1000) &&
 (D_1778@offset@79 == (allocbuf@offset@78 + 1000)) &&
 (allocp@offset@77 == allocp_5@offset@71) &&
 (allocbuf@offset@78 == allocbuf@offset@0) &&
 (allocp@offset@77 < 100) &&
 (allocbuf@offset@78 < 1000) &&
 (allocbuf@offset@0 < 1000) &&
 (allocp_9@baseAddr@79 == D_1778@baseAddr@79) &&
 (D_1778@baseAddr@79 == allocp@baseAddr@77) &&
 (allocp@baseAddr@77 == allocp_5@baseAddr@71) &&
 (allocp_5@baseAddr@71 == allocbuf@baseAddr@78) &&
 (allocbuf@baseAddr@78 == allocbuf@baseAddr@0) &&
 (retval_10@86 == 0) &&
 (retval_10@86 == ((bool) (D_1786@85 < alloc@85))) &&
 (D_1786@85 == ((int) (D_1782@84 - D_1785@84))) &&
 (alloc@85 == ((int) alloc@0)) &&
 (D_1782@84 == ((int) ((int) D_1781@offset@80))) &&
 (D_1785@84 == ((int) ((int) D_1784@offset@83))) &&
 (D_1781@baseAddr@80 == allocbuf@baseAddr@79) &&
 (allocbuf@offset@79 == 0) &&
  (allocbuf@offset@79 < 1000) &&
  (D_1781@offset@80 == (allocbuf@offset@79 + 1000)) &&
  (D_1784@offset@83 == allocp@offset@0) &&
  (allocbuf@offset@79 == allocbuf@offset@0) &&
  (allocp@offset@0 < 100) &&
  (allocbuf@offset@79 < 1000) &&
  (allocbuf@offset@0 < 1000) &&
  (D_1781@baseAddr@80 == D_1784@baseAddr@83) &&
  (D_1784@baseAddr@83 == allocp@baseAddr@0) &&
  (allocp@baseAddr@0 == allocbuf@baseAddr@79) &&
  (allocbuf@baseAddr@79 == allocbuf@baseAddr@0) &&
  (retval_12@93 != 0) &&
  (retval_12@93 == ((bool) (D_1799@92 >= alloc@92))) &&
  (D_1799@92 == ((int) (D_1795@91 - D_1798@91))) &&
  (alloc@92 == ((int) alloc@0)) &&
  (D_1795@91 == ((int) ((int) D_1794@offset@87))) &&
  (D_1798@91 == ((int) ((int) D_1797@offset@90))) &&
  (D_1794@baseAddr@87 == allocbuf@baseAddr@86) &&
  (allocbuf@offset@86 == 0) &&
  (allocbuf@offset@86 < 1000) &&
  (D_1794@offset@87 == (allocbuf@offset@86 + 1000)) &&
  (D_1797@offset@90 == allocp@offset@0) &&
  (allocbuf@offset@86 == allocbuf@offset@0) &&
  (allocp@offset@0 < 100) &&
  (allocbuf@offset@86 < 1000) &&
  (allocbuf@offset@0 < 1000) &&
  (D_1794@baseAddr@87 == D_1797@baseAddr@90) &&
  (D_1797@baseAddr@90 == allocp@baseAddr@0) &&
  (allocp@baseAddr@0 == allocbuf@baseAddr@86) &&
  (allocbuf@baseAddr@86 == allocbuf@baseAddr@0) &&
  (retval_14@96 == 0) &&
  (retval_14@96 == ((bool) (D_1804@offset@95 != __rtt_return_0x40bf12d8@offset@95))) &&
  (D_1804@offset@95 == allocp@offset@0) &&
  (allocp@offset@0 < 100) &&
  (D_1804@baseAddr@95 == __rtt_return_0x40bf12d8@baseAddr@95) &&
  (__rtt_return_0x40bf12d8@baseAddr@95 == allocp@baseAddr@0) &&
  (__rtt_return_0x40bf12d8@offset@95 == return@offset@98) &&
  (return@offset@98 == D_1809@offset@97) &&
  (D_1809@offset@97 == retval_0x40bf1270@offset@96) &&
  (retval_0x40bf1270@offset@96 == (allocp_6@offset@75 + D_1771@75)) &&
  (allocp_6@offset@75 == allocp@offset@72) &&
  (D_1771@75 == ((unsigned int) (-n_7@74))) &&
  (allocp@offset@72 == allocp_5@offset@71) &&
  (n_7@74 == ((unsigned int) ((unsigned int) alloc@73))) &&
  (alloc@73 == ((int) alloc@0)) &&
  (allocp@offset@72 < 100) &&
  (return@baseAddr@98 == D_1809@baseAddr@97) &&
  (D_1809@baseAddr@97 == retval_0x40bf1270@baseAddr@96) &&
  (retval_0x40bf1270@baseAddr@96 == allocp_6@baseAddr@75) &&
  (allocp_6@baseAddr@75 == allocp@baseAddr@72) &&
  (allocp@baseAddr@72 == allocp_5@baseAddr@71))
SOLUTION:
    alloc@0 = (ConcreteLattice<signed int>, 0)
    allocp@baseAddr@0 = (ConcreteLattice<unsigned int>, 99)
    allocp@offset@0 = (ConcreteLattice<unsigned int>, 99)
    allocbuf@baseAddr@0 = (ConcreteLattice<unsigned int>, 99)
    allocbuf@offset@0 = (ConcreteLattice<unsigned int>, 0)
    alloc@61 = (ConcreteLattice<signed int>, 0)
    allocp@baseAddr@61 = (ConcreteLattice<unsigned int>, 99)
    allocp@offset@61 = (ConcreteLattice<unsigned int>, 99)
    allocp_0@baseAddr@62 = (ConcreteLattice<unsigned int>, 99)
    allocbuf@baseAddr@63 = (ConcreteLattice<unsigned int>, 99)
```

```
allocbuf@offset@63 = (ConcreteLattice<unsigned int>, 0)
D_1759@baseAddr@64 = (ConcreteLattice<unsigned int>, 99)
D_1759@offset@64 = (ConcreteLattice<unsigned int>, 1000)
allocp@baseAddr@65 = (ConcreteLattice<unsigned int>, 99)
allocp@offset@65 = (ConcreteLattice<unsigned int>, 99)
allocp_1@baseAddr@66 = (ConcreteLattice<unsigned int>, 99)
allocp_1@offset@66 = (ConcreteLattice<unsigned int>, 99)
D_1760@67 = (ConcreteLattice<signed int>, 1000)
allocp_2@67 = (ConcreteLattice<signed int>, 99)
alloc@68 = (ConcreteLattice<signed int>, 0)
allocp@baseAddr@68 = (ConcreteLattice<unsigned int>, 99)
allocp@offset@68 = (ConcreteLattice<unsigned int>, 99)
D_1763@68 = (ConcreteLattice<signed int>, 901)
alloc@69 = (ConcreteLattice<signed int>, 0)
allocp_3@baseAddr@70 = (ConcreteLattice<unsigned int>, 99)
allocp_3@offset@70 = (ConcreteLattice<unsigned int>, 99)
n_4@70 = (ConcreteLattice<unsigned int>, 0)
allocp_5@baseAddr@71 = (ConcreteLattice<unsigned int>, 99)
allocp_5@offset@71 = (ConcreteLattice<unsigned int>, 99)
allocp@baseAddr@72 = (ConcreteLattice<unsigned int>, 99)
allocp@offset@72 = (ConcreteLattice<unsigned int>, 99)
alloc@73 = (ConcreteLattice<signed int>, 0)
n_7@74 = (ConcreteLattice<unsigned int>, 0)
D_1771@75 = (ConcreteLattice<unsigned int>, 0)
allocp_6@baseAddr@75 = (ConcreteLattice<unsigned int>, 99)
allocp_6@offset@75 = (ConcreteLattice<unsigned int>, 99)
allocp@baseAddr@76 = (ConcreteLattice<unsigned int>, 99)
allocp@offset@76 = (ConcreteLattice<unsigned int>, 99)
allocp@baseAddr@77 = (ConcreteLattice<unsigned int>, 99)
allocp@offset@77 = (ConcreteLattice<unsigned int>, 99)
allocp_8@baseAddr@77 = (ConcreteLattice<unsigned int>, 99)
allocbuf@baseAddr@78 = (ConcreteLattice<unsigned int>, 99)
allocbuf@offset@78 = (ConcreteLattice<unsigned int>, 0)
allocbuf@baseAddr@79 = (ConcreteLattice<unsigned int>, 99)
allocbuf@offset@79 = (ConcreteLattice<unsigned int>, 0)
D_1778@baseAddr@79 = (ConcreteLattice<unsigned int>, 99)
D_1778@offset@79 = (ConcreteLattice<unsigned int>, 1000)
allocp_9@baseAddr@79 = (ConcreteLattice<unsigned int>, 99)
allocp_9@offset@79 = (ConcreteLattice<unsigned int>, 99)
D_1781@baseAddr@80 = (ConcreteLattice<unsigned int>, 99)
D_1781@offset@80 = (ConcreteLattice<unsigned int>, 1000)
D_1784@baseAddr@83 = (ConcreteLattice<unsigned int>, 99)
D_1784@offset@83 = (ConcreteLattice<unsigned int>, 99)
D_1782@84 = (ConcreteLattice<signed int>, 1000)
D_1785@84 = (ConcreteLattice<signed int>, 99)
alloc@85 = (ConcreteLattice<signed int>, 0)
D_1786@85 = (ConcreteLattice<signed int>, 901)
allocbuf@baseAddr@86 = (ConcreteLattice<unsigned int>, 99)
allocbuf@offset@86 = (ConcreteLattice<unsigned int>, 0)
retval_10@86 = (ConcreteLattice<bool>, 0)
D_1794@baseAddr@87 = (ConcreteLattice<unsigned int>, 99)
D_1794@offset@87 = (ConcreteLattice<unsigned int>, 1000)
D_1797@baseAddr@90 = (ConcreteLattice<unsigned int>, 99)
D_1797@offset@90 = (ConcreteLattice<unsigned int>, 99)
D_1795@91 = (ConcreteLattice<signed int>, 1000)
D_1798@91 = (ConcreteLattice<signed int>, 99)
alloc@92 = (ConcreteLattice<signed int>, 0)
D_1799@92 = (ConcreteLattice<signed int>, 901)
retval_12@93 = (ConcreteLattice<bool>, 1)
__rtt_return_0x40bf12d8@baseAddr@95 = (ConcreteLattice<unsigned int>, 99)
D_1804@baseAddr@95 = (ConcreteLattice<unsigned int>, 99)
D_1804@offset@95 = (ConcreteLattice<unsigned int>, 99)
```

```
    __rtt_return_0x40bf12d8@offset@95 = (ConcreteLattice<unsigned int>, 99)
    retval_0x40bf1270@baseAddr@96 = (ConcreteLattice<unsigned int>, 99)
    retval_0x40bf1270@offset@96 = (ConcreteLattice<unsigned int>, 99)
    retval_14@96 = (ConcreteLattice<bool>, 0)
    D_1809@baseAddr@97 = (ConcreteLattice<unsigned int>, 99)
    D_1809@offset@97 = (ConcreteLattice<unsigned int>, 99)
    return@baseAddr@98 = (ConcreteLattice<unsigned int>, 99)
    return@offset@98 = (ConcreteLattice<unsigned int>, 99)

MAX SIZE IS STILL NOT REACHED
NO EXPANSION IS POSSIBLE ANY MORE AND NO NEW TRACE
Operation alloc() is not covered.
Uncovered transitions:
[id=40] —— [ (!(allocp_8 != 0)) ] ——> __rtt_testcase_error__((&"(n␣>=␣0␣&&␣allocp␣!=␣0)
    "), (&"(allocp␣!=␣0␣&&␣allocp␣<=␣allocbuf␣+␣ALLOCSIZE)"), (&""));
[id=42] —— [ (!(allocp_9 <= D_1778)) ] ——> __rtt_testcase_error__((&"(n␣>=␣0␣&&␣allocp␣
    !=␣0)"), (&"(allocp␣!=␣0␣&&␣allocp␣<=␣allocbuf␣+␣ALLOCSIZE)"), (&""));
[id=44] —— [ true ] ——> D_1781 = ((&allocbuf[0]) + 1000);
[id=47] —— [ (__rtt_return_0x40bf12d8 != 0) ] ——> __rtt_testcase_error__((&"allocbuf+
    ALLOCSIZE−__rtt_initial(allocp)<n"), (&"__rtt_return==0"), (&"CTGEN_001"));
[id=49] —— [ true ] ——> D_1794 = ((&allocbuf[0]) + 1000);
[id=53] —— [ (retval_14 != 0) ] ——> __rtt_testcase_error__((&"allocbuf+ALLOCSIZE−
    __rtt_initial(allocp)>=n"), (&"__rtt_return==__rtt_initial(allocp)"), (&"CTGEN_002"))
    ;
[id=55] —— [ true ] ——> D_1809 = retval_0x40bf1270;

Covered:
Total transitions:       87%
Transitions with guards: 77%
```

2.4 GCOV Output

To control the coverage declared by the generator, we measured the actual coverage produced by running of the generated test against the UUT with GCC. The result listed below shows, that as expected, the generated test driver produced 100% branch coverage.

```
        4:    6:char *alloc(int n){
        4:    7:  __rtt_modifies(allocp);
call    0 returned 100%
        -:    8:  __rtt_precondition(n >= 0 && allocp != 0);
        -:    9:  __rtt_postcondition(allocp != 0 && allocp <= allocbuf + ALLOCSIZE);
        -:   10:  __rtt_testcase(allocbuf + ALLOCSIZE − __rtt_initial(allocp) < n,
        -:   11:                 __rtt_return == 0,
        4:   12:                 "CTGEN_001");
call    0 returned 100%
        -:   13:  __rtt_testcase(allocbuf + ALLOCSIZE − __rtt_initial(allocp) >= n,
        -:   14:                 __rtt_return == __rtt_initial(allocp),
        4:   15:                 "CTGEN_002");
call    0 returned 100%
        -:   16:
        4:   17:  char *retval = 0;
        4:   18:  if(allocbuf + ALLOCSIZE − allocp >= n){
branch  0 taken 75% (fallthrough)
branch  1 taken 25%
        3:   19:    allocp += n;
        3:   20:    retval = allocp − n;
        -:   21:  }
        -:   22:
        4:   23:  return retval;
        -:    24:}
```

2.5 Graphical Output

Figure 4 shows the graphical representation of the CFG corresponding to the module under test. As expected, the nodes and edges corresponding to the test case or the postcondition violation are drawn red, which indicates that the generator was not able to find any input data to cover them. All remaining nodes and edges of this CFG are drawn blue, which indicates that all of them were successfully covered.

3 Dereferenced Pointer Resolution

This example corresponds to the example discussed in Subsection 5.6.2 and demonstrates how CTGEN handles dereferenced pointers.

3.1 Analyzed Code

The source code listed below contains the implementation of the module under test `ptr_test()`. The function receives two pointer parameters as input and reaches the line with an "error" in case when the guard condition `(*p1 == 1)` is evaluated to true.

```
int ptr_test(int *p1, int *p2)
{
  *p1 = 0;
  *p2 = 1;
  if(*p1 == 1){
    // error
    return 1;
  } else {
    return 0;
  }
}
```

3.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve 100% branch coverage for the module under test `ptr_test()`. The generator needs two test cases to achieve complete branch coverage, therefore the test driver contains two test steps. Since no specification of the module under test was given, the test driver contains no assertions. The function `ptr_test()` contains no defined or undefined function calls, thus no stub functions are generated.

First, the module under test is declared. Then auxiliary variables needed to manipulate pointer parameters in order to fulfill the guard conditions as well as variables for passing function parameters are declared. In each test step the assignment of pointer parameters as well as auxiliary variables is made according to the calculated values and after the setting is done, the module under test is invoked.

```
/* ==============================
 * Include section
 * ==============================*/
```

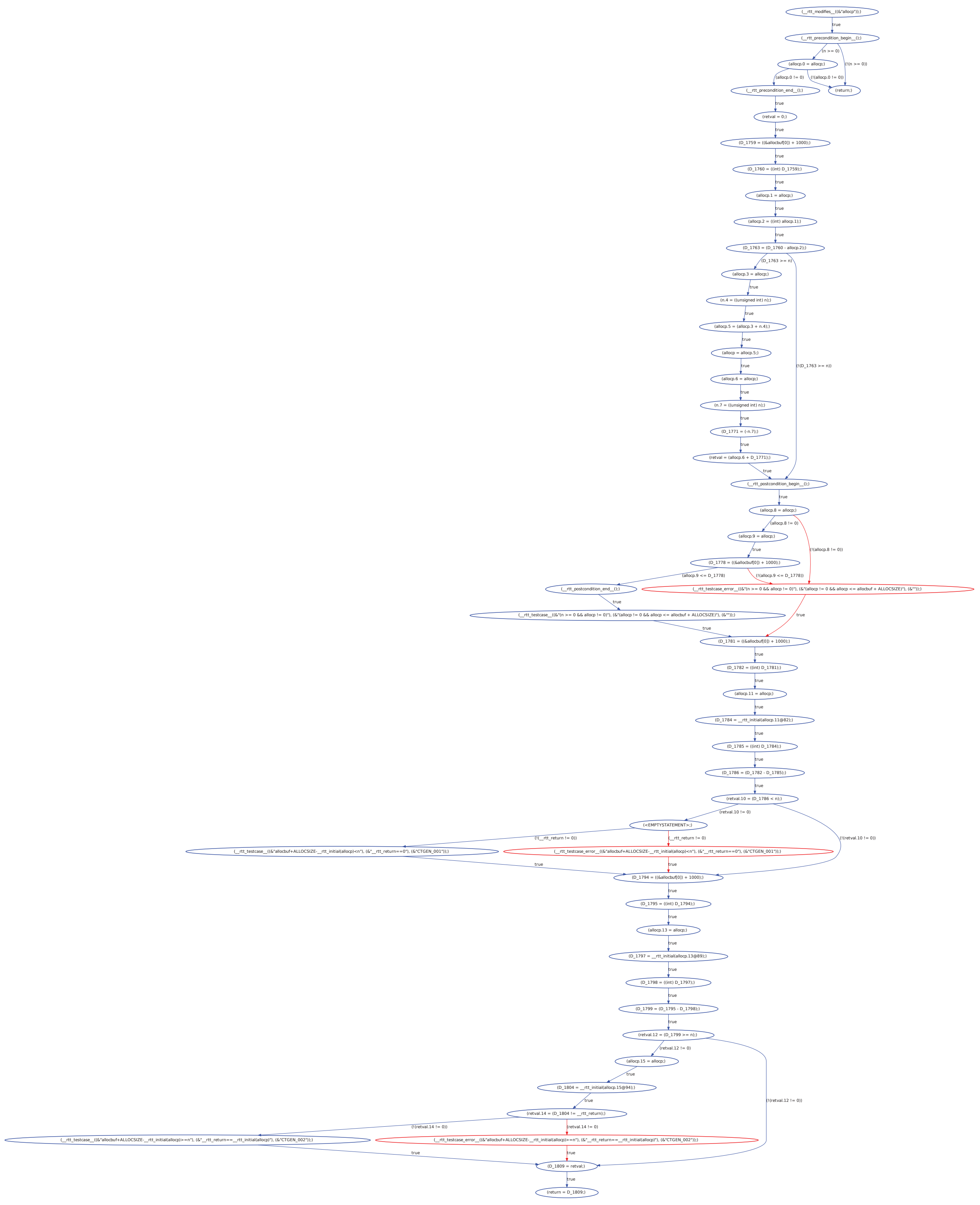

(__rtt_modifies__((&"allocp"));)
true
(__rtt_precondition_begin__();)
(n >= 0)
(!(n >= 0))
(allocp.0 = allocp;)
(allocp.0 != 0)
(!(allocp.0 != 0))
(__rtt_precondition_end__();)
(return;)
(retval = 0;)
(D_1759 = ((&allocbuf[0]) + 1000);)
(D_1760 = ((int) D_1759);)
(allocp.1 = allocp;)
(allocp.2 = ((int) allocp.1);)
(D_1763 = (D_1760 - allocp.2);)
(D_1763 >= n)
(allocp.3 = allocp;)
(n.4 = ((unsigned int) n);)
(allocp.5 = (allocp.3 + n.4);)
(allocp = allocp.5;)
(!(D_1763 >= n))
(allocp.6 = allocp;)
(n.7 = ((unsigned int) n);)
(D_1771 = (-n.7);)
(retval = (allocp.6 + D_1771);)
(__rtt_postcondition_begin__();)
(allocp.8 = allocp;)
(allocp.8 != 0)
(allocp.9 = allocp;)
(!(allocp.8 != 0))
(D_1778 = ((&allocbuf[0]) + 1000);)
(allocp.9 <= D_1778)
(!(allocp.9 <= D_1778))
(__rtt_postcondition_end__();)
(__rtt_testcase_error__((&"(n >= 0 && allocp != 0)"), (&"(allocp != 0 && allocp <= allocbuf + ALLOCSIZE)"), (&""));)
(__rtt_testcase__((&"(n >= 0 && allocp != 0)"), (&"(allocp != 0 && allocp <= allocbuf + ALLOCSIZE)"), (&""));)
(D_1781 = ((&allocbuf[0]) + 1000);)
(D_1782 = ((int) D_1781);)
(allocp.11 = allocp;)
(D_1784 = __rtt_initial(allocp.11@82);)
(D_1785 = ((int) D_1784);)
(D_1786 = (D_1782 - D_1785);)
(retval.10 = (D_1786 < n);)
(retval.10 != 0)
(<EMPTYSTATEMENT>;)
(!(__rtt_return != 0))
(__rtt_return != 0)
(!(retval.10 != 0))
(__rtt_testcase__((&"allocbuf+ALLOCSIZE-__rtt_initial(allocp)<n"), (&"__rtt_return==0"), (&"CTGEN_001"));)
(__rtt_testcase_error__((&"allocbuf+ALLOCSIZE-__rtt_initial(allocp)<n"), (&"__rtt_return==0"), (&"CTGEN_001"));)
(D_1794 = ((&allocbuf[0]) + 1000);)
(D_1795 = ((int) D_1794);)
(allocp.13 = allocp;)
(D_1797 = __rtt_initial(allocp.13@89);)
(D_1798 = ((int) D_1797);)
(D_1799 = (D_1795 - D_1798);)
(retval.12 = (D_1799 >= n);)
(retval.12 != 0)
(allocp.15 = allocp;)
(D_1804 = __rtt_initial(allocp.15@94);)
(!(retval.12 != 0))
(retval.14 = (D_1804 != __rtt_return);)
(!(retval.14 != 0))
(retval.14 != 0)
(__rtt_testcase__((&"allocbuf+ALLOCSIZE-__rtt_initial(allocp)>=n"), (&"__rtt_return==__rtt_initial(allocp)"), (&"CTGEN_002"));)
(__rtt_testcase_error__((&"allocbuf+ALLOCSIZE-__rtt_initial(allocp)>=n"), (&"__rtt_return==__rtt_initial(allocp)"), (&"CTGEN_002"));)
(D_1809 = retval;)
(return = D_1809;)

Figure 4: Graphical representation for the annotation example.

```
/* ===============================
 * Structures
 * ===============================*/


/* ===============================
 * Global or static C declarations and definitions
 * ===============================*/

extern int ptr_test(int* p1, int* p2);
@uut int ptr_test(int* p1, int* p2);


/* ===============================
 * Abstract machine declaration.
 * ===============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:

    int __rtt_return;
    int p1_PointsTo;
    int* p1 = &p1_PointsTo;
    int p2_PointsTo;
    int* p2 = &p2_PointsTo;

    int p1_PointsTo_[100];
    unsigned int p1_PointsTo_offset_;
    unsigned int p2_PointsTo_offset_;

    @rttBeginTestStep; // ----------------------------------------------------------
    {

       p1 = p1_PointsTo_;
       p2 = p1_PointsTo_;

       p1_PointsTo_offset_ = 48;
       p1 += p1_PointsTo_offset_;
       p2_PointsTo_offset_ = 48;
       p2 += p2_PointsTo_offset_;

       @rttCall(__rtt_return = ptr_test(p1, p2));
    }
    @rttEndTestStep; // ------------------------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------------------
    {

       p1 = p1_PointsTo_;
       p2 = p1_PointsTo_;

       p1_PointsTo_offset_ = 50;
       p1 += p1_PointsTo_offset_;
       p2_PointsTo_offset_ = 51;
       p2 += p2_PointsTo_offset_;
```

```
        @rttCall(__rtt_return = ptr_test(p1, p2));
    }
    @rttEndTestStep; // ----------------------------------------

}
```

3.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of two traces (corresponding to the two test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported, in this example it is 100%.

```
SOLUTION FOR FUNCTION ptr_test

TRACE 1
TRACE COMPLETED

TRACE
{ ((*p1) = 0;),
  ((*p2) = 1;),
  (D_1724 = (*p1);),
  (D_1727 = 1;),
  (return = D_1727;) }
  traceState:     CONT_OF_ANOTHER_TRACE
  currentStepNr: 4
  feasible: 1

CONSTRAINT:
((((((((D_1724@16 == 1) &&
 ((D_1724@16 == 1) &&
 (p1@offset@0 == p2@offset@0) || ((D_1724@16 == 0) &&
 (p1@offset@0 != p2@offset@0)))) &&
 (p1@offset@0 < 100)) &&
 (p1@offset@0 == p1rttTgenIdxExp0@15)) &&
 (p2@offset@0 < 100)) &&
 (p2@offset@0 == p2rttTgenIdxExp0@15)) &&
 (p1@baseAddr@0 == p1@baseAddr@0)) &&
 (p1@baseAddr@0 == p2@baseAddr@0))
SOLUTION:
    p1@baseAddr@0 = (ConcreteLattice<unsigned int>, 2147483648)
    p1@offset@0 = (ConcreteLattice<unsigned int>, 48)
    p2@baseAddr@0 = (ConcreteLattice<unsigned int>, 2147483648)
    p2@offset@0 = (ConcreteLattice<unsigned int>, 48)
    p1rttTgenIdxExp0@15 = (ConcreteLattice<unsigned int>, 48)
    p2rttTgenIdxExp0@15 = (ConcreteLattice<unsigned int>, 48)
    D_1724@16 = (ConcreteLattice<signed int>, 1)


TRACE 3
TRACE COMPLETED

TRACE
{ ((*p1) = 0;),
  ((*p2) = 1;),
  (D_1724 = (*p1);),
   (D_1727 = 0;),
   (return = D_1727;) }
   traceState:      CONT_OF_ANOTHER_TRACE
   currentStepNr: 4
   feasible: 1

CONSTRAINT:
(((((((( D_1724@16 != 1) &&
 ((D_1724@16 == 1) &&
 (p1@offset@0 == p2@offset@0) || ((D_1724@16 == 0) &&
 (p1@offset@0 != p2@offset@0)))) &&
 (p1@offset@0 < 100)) &&
 (p1@offset@0 == p1rttTgenIdxExp0@15)) &&
 (p2@offset@0 < 100)) &&
 (p2@offset@0 == p2rttTgenIdxExp0@15)) &&
 (p1@baseAddr@0 == p1@baseAddr@0)) &&
 (p1@baseAddr@0 == p2@baseAddr@0))
SOLUTION:
    p1@baseAddr@0 = (ConcreteLattice<unsigned int>, 2147483648)
    p1@offset@0 = (ConcreteLattice<unsigned int>, 50)
    p2@baseAddr@0 = (ConcreteLattice<unsigned int>, 2147483648)
    p2@offset@0 = (ConcreteLattice<unsigned int>, 51)
    p1rttTgenIdxExp0@15 = (ConcreteLattice<unsigned int>, 50)
    p2rttTgenIdxExp0@15 = (ConcreteLattice<unsigned int>, 51)
    D_1724@16 = (ConcreteLattice<signed int>, 0)


Operation ptr_test() is covered.

Covered:
Total transitions:          100%
Transitions with guards: 100%
```

3.4 Graphical Output

Figure 5 shows the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue, which indicates that all of them were successfully covered.

4 Pointer Resolution

This example corresponds to the example discussed in Section 5.7 and demonstrates how CTGEN handles pointer operations.

4.1 Analyzed Code

The source code listed below contains the implementation of the module under test `test()`. The function receives two pointer parameters as input and reaches the line with an "error" in case when the guard condition `(p1 < p2)` is evaluated to true.

```
void test(char *p1, char *p2)
{
    if(p1 < p2){
        int error = 1;
    }
```

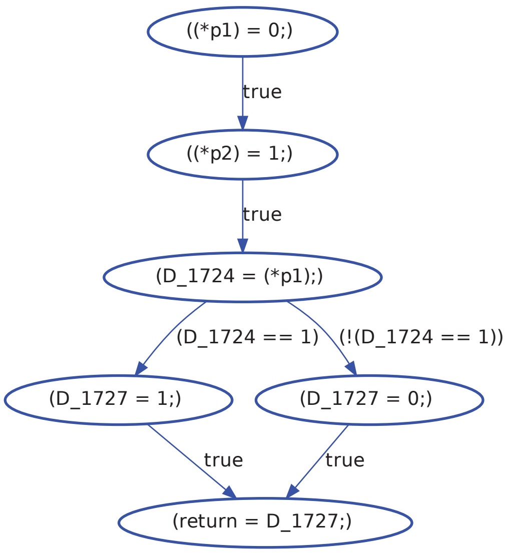


Figure 5: Graphical representation for the example *Dereferenced Pointer Resolution*.

```
    return;

}
```

4.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve 100% branch coverage for the module under test `test()`. The generator needs two test cases to achieve complete branch coverage, therefore the test driver contains two test steps. Since no specification of the module under test was given, the test driver contains no assertions. The function `test()` contains no defined or undefined function calls, therefore no stub functions were generated.

First, the module under test is declared. Then auxiliary variables needed to manipulate pointer parameters in order to fulfill the guard conditions as well as variables for passing of the function parameters are declared. In each test step the assignment of pointer parameters as well as auxiliary variables is made according to the calculated values and after the setting is done, the module under test is invoked.

```
/* ===============================
 * Include section
 * ===============================*/


/* ===============================
 * Global or static C declarations and definitions
 * ===============================*/

extern void test(char* p1, char* p2);
```

```
@uut void test(char* p1, char* p2);


/* ==============================
 * Abstract machine declaration.
 * ==============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:

        char p1_rtt_help_pointer;
        char* p1 = &p1_rtt_help_pointer;
        char p2_rtt_help_pointer;
        char* p2 = &p2_rtt_help_pointer;

    char p1_rtt_help_pointer_[10];
    unsigned int p1_rtt_help_pointer_offset_;
    unsigned int p2_rtt_help_pointer_offset_;

    @rttBeginTestStep; // ------------------------------------------------
    {

       p1 = p1_rtt_help_pointer_;
       p2 = p1_rtt_help_pointer_;

       p1_rtt_help_pointer_offset_ = 1;
       p1 += p1_rtt_help_pointer_offset_;
;
       p2_rtt_help_pointer_offset_ = 9;
       p2 += p2_rtt_help_pointer_offset_;
;
       @rttCall(test(p1, p2));
    }
    @rttEndTestStep; // ------------------------------------------------

    @rttBeginTestStep; // ------------------------------------------------
    {

       p1 = p1_rtt_help_pointer_;
       p2 = p1_rtt_help_pointer_;

       p1_rtt_help_pointer_offset_ = 9;
       p1 += p1_rtt_help_pointer_offset_;
;
       p2_rtt_help_pointer_offset_ = 9;
       p2 += p2_rtt_help_pointer_offset_;
;
       @rttCall(test(p1, p2));
    }
    @rttEndTestStep; // ------------------------------------------------

}
```

4.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of two traces (corresponding to the two test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported, in this example it is 100%.

```
SOLUTION FOR FUNCTION test

TRACE 1
TRACE COMPLETED

TRACE
{ (<EMPTYSTATEMENT>;),
  (error_0x40bed618 = 1;),
  (return;) }
  traceState:     CONT_OF_ANOTHER_TRACE
  currentStepNr: 2
  feasible: 1

CONSTRAINT:
(((((((( p1@offset@7 < p2@offset@7) &&
 (p1@baseAddr@7 == p1@baseAddr@4) &&
 (p1@offset@7 == p1@offset@4)) &&
 (p2@baseAddr@7 == p2@baseAddr@6) &&
 (p2@offset@7 == p2@offset@6)) &&
 (p1@offset@7 < 10)) &&
 (p2@offset@7 < 10)) &&
 (p1@baseAddr@7 == p2@baseAddr@7)) &&
 (p1@baseAddr@7 != 0)) &&
 (p2@baseAddr@7 != 0))
SOLUTION:
    p1@baseAddr@4 = (ConcreteLattice<unsigned int>, 2147483648)
    p1@offset@4 = (ConcreteLattice<unsigned int>, 1)
    p2@baseAddr@6 = (ConcreteLattice<unsigned int>, 2147483648)
    p2@offset@6 = (ConcreteLattice<unsigned int>, 9)
    p1@baseAddr@7 = (ConcreteLattice<unsigned int>, 2147483648)
    p1@offset@7 = (ConcreteLattice<unsigned int>, 1)
    p2@baseAddr@7 = (ConcreteLattice<unsigned int>, 2147483648)
    p2@offset@7 = (ConcreteLattice<unsigned int>, 9)


TRACE 2
TRACE COMPLETED

TRACE
{ (<EMPTYSTATEMENT>;),
  (return;) }
  traceState:     CONT_AFTER_SOLVING
  currentStepNr: 1
  feasible: 1

CONSTRAINT:
(((((((( p1@offset@7 >= p2@offset@7) &&
 (p1@baseAddr@7 == p1@baseAddr@4) &&
 (p1@offset@7 == p1@offset@4)) &&
 (p2@baseAddr@7 == p2@baseAddr@6) &&
 (p2@offset@7 == p2@offset@6)) &&
```

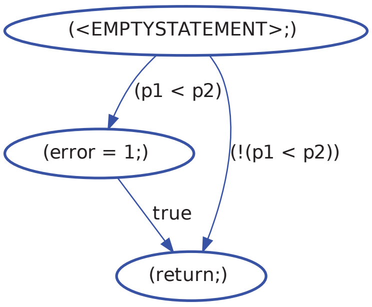


Figure 6: Graphical representation for the example *Pointer Resolution*.

```
 (p1@offset@7 < 10)) &&
 (p2@offset@7 < 10)) &&
 (p1@baseAddr@7 == p2@baseAddr@7)) &&
 (p1@baseAddr@7 != 0)) &&
 (p2@baseAddr@7 != 0))
SOLUTION:
    p1@baseAddr@4 = (ConcreteLattice<unsigned int>, 2147483648)
    p1@offset@4 = (ConcreteLattice<unsigned int>, 9)
    p2@baseAddr@6 = (ConcreteLattice<unsigned int>, 2147483648)
    p2@offset@6 = (ConcreteLattice<unsigned int>, 9)
    p1@baseAddr@7 = (ConcreteLattice<unsigned int>, 2147483648)
    p1@offset@7 = (ConcreteLattice<unsigned int>, 9)
    p2@baseAddr@7 = (ConcreteLattice<unsigned int>, 2147483648)
    p2@offset@7 = (ConcreteLattice<unsigned int>, 9)


Operation test() is covered.

Covered:
Total transitions:        100%
Transitions with guards:  100%
```

4.4 Graphical Output

Figure 6 shows the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue, which indicates that all of them were successfully covered.

5 Address Operation Resolution

This example corresponds to the example discussed in Section 5.7.3 and demonstrates how CTGEN handles address operations.

5.1 Analyzed Code

The source code listed below contains the implementation of the module under test `test_address()`. The function sets two pointers `p1` and `p2` to elements of the input array `a[10]` and reaches the line with an "error" in case when the guard condition `(p1 > p2)` is evaluated to true.

```
int a[10];
void test_address()
{
    int *p1 = &a[1];
    int *p2 = &a[4];

    if(p1 > p2){
        int error = 1;
    }
    return;
}
```

5.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve the maximal possible branch coverage for the module under test `test_address()`. The generator generated one test case, since the UUT contains only two possible paths through it and one of them is infeasible. Since no specification of the module under test was given, the test driver contains no assertions. Function `test()` contains no defined or undefined function calls, therefore no stub functions are generated.

First, the module under test is declared. Since the guard condition of the only `if`-statement of the UUT does not depend on any input, the generated test driver does not contain any settings for inputs.

```
/* ===============================
 * Include section
 * ===============================*/


/* ===============================
 * Structures
 * ===============================*/


/* ===============================
 * Global or static C declarations and definitions
 * ===============================*/

extern void test_address();
@uut void test_address();

extern int a[10];

/* ===============================
 * Abstract machine declaration.
 * ===============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:
```

```
 @PROCESS:

     @rttBeginTestStep; // ------------------------------------------------
     {
        @rttCall(test_address());
     }
     @rttEndTestStep; // --------------------------------------------------
}
```

5.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of one trace (corresponding to the one test step in the test driver). The trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported, in this example it is as expected 50%.

```
SOLUTION FOR FUNCTION test_address

TRACE 1
TRACE COMPLETED

TRACE
{ (p1 = (&a[1]);),
  (p2 = (&a[4]);),
  (return;) }
  traceState:     CONT_AFTER_SOLVING
  currentStepNr: 2
  feasible: 1

CONSTRAINT:
(((((((((((p1@offset@7 <= p2@offset@7) &&
 (p1@baseAddr@7 == a@baseAddr@5) &&
 (a@offset@5 == 0) &&
 (a@offset@5 < 10) &&
 (p1@offset@7 == (a@offset@5 + 1))) &&
 (p2@baseAddr@7 == a@baseAddr@6) &&
 (a@offset@6 == 0) &&
 (a@offset@6 < 10) &&
 (p2@offset@7 == (a@offset@6 + 4))) &&
 (a@offset@5 == a@offset@0)) &&
 (a@offset@6 == a@offset@0)) &&
 (a@offset@5 < 10)) &&
 (a@offset@0 < 10)) &&
 (a@offset@6 < 10)) &&
 (p1@baseAddr@7 == p2@baseAddr@7)) &&
 (p2@baseAddr@7 == a@baseAddr@5)) &&
 (a@baseAddr@5 == a@baseAddr@0)) &&
 (a@baseAddr@0 == a@baseAddr@6))
SOLUTION:
   a@baseAddr@0 = (ConcreteLattice<unsigned int>, 4294967295)
   a@offset@0 = (ConcreteLattice<unsigned int>, 0)
   a@baseAddr@5 = (ConcreteLattice<unsigned int>, 4294967295)
   a@offset@5 = (ConcreteLattice<unsigned int>, 0)
   a@baseAddr@6 = (ConcreteLattice<unsigned int>, 4294967295)
   a@offset@6 = (ConcreteLattice<unsigned int>, 0)
```

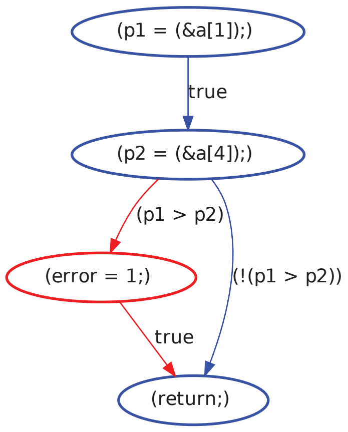


Figure 7: Graphical representation for the example *Address Operation Resolution*.

```
    p1@baseAddr@7 = (ConcreteLattice<unsigned int>, 4294967295)
    p1@offset@7 = (ConcreteLattice<unsigned int>, 1)
    p2@baseAddr@7 = (ConcreteLattice<unsigned int>, 4294967295)
    p2@offset@7 = (ConcreteLattice<unsigned int>, 4)


MAX SIZE IS STILL NOT REACHED
NO EXPANSION IS POSSIBLE ANY MORE AND NO NEW TRACE
 Operation test_address() is not covered.
 Uncovered transitions:
 [id=1] —— [ (p1 > p2) ]  ——> error = 1;
 [id=3] —— [ true ]  ——> return;

 Covered:
 Total transitions:         50%
 Transitions with guards:  50%
```

5.4 Graphical Output

Figure 7 shows the graphical representation of the CFG corresponding to the module under test. All covered nodes and edges of the CFG are drawn blue, all uncovered ones red.

6 Structure Access Resolution

This example corresponds to the example discussed in Section 5.8 and demonstrates the technique developed for the resolution of structure accesses.

6.1 Analyzed Code

The source code listed below contains the implementation of the module under test `structAccess()`. The function receives a parameter `p1` of type `structType_t` as input. The module under test contains a couple of `if` statements which demonstrate different aspects of the algorithm developed for the resolution of structure accesses: line 14 contains the evaluation of a defined structure field, line 16 contains the evaluation of an undefined structure field and line 25 demonstrates a nested structure access.

```
 1  typedef struct{
 2      int f1;
 3      int f2;
 4  }nestedStructType_t;
 5
 6  typedef struct{
 7    int field1;
 8    int field2;
 9    nestedStructType_t field3;
10  }structType_t;
11
12  int structAccess(structType_t p1){
13    p1.field1 = 4;
14    if(p1.field1 < 0){
15        return 1;
16    } else if(p1.field2 > 0){
17        return 2;
18    }
19
20    nestedStructType_t tmp;
21    tmp.f1 = 5;
22    tmp.f2 = 7;
23    p1.field3 = tmp;
24
25    if(p1.field3.f1 == 5){
26        return 3;
27    }
28    return 4;
29  }
```

The analysis shows, that lines 15 and 28 of the module under test are unreachable since the guard conditions `(p1.field1 < 0)` and `(p1.field3.f1 != 5)` are infeasible.

6.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve the maximal possible branch coverage for the module under test `structAccess()`. The generator produced two test cases, therefore the test driver contains two test steps. Since no specification of the module under test was given, the test driver contains no assertions. The function `structAccess()` contains no defined or undefined function calls, therefore no stub functions are generated.

First, the module under test is declared. Then a variable needed for passing the parameter to the module under test as well as an auxiliary variable for the return value of the function are declared. In each test step the assignment of the parameter is made according to the calculated values and after the setting is done, the module under test is invoked.

```
/* ==============================
 * Include section
 * ==============================*/

/* ==============================
 * Global or static C declarations and definitions
 * ==============================*/

extern int structAccess(structType_t p1);
@uut int structAccess(structType_t p1);


/* ==============================
 * Abstract machine declaration.
 * ==============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:

    int __rtt_return;
        structType_t p1;


    @rttBeginTestStep; // ----------------------------------------------------
    {
       p1.field2 = 1073741824;


       @rttCall(__rtt_return = structAccess(p1));
    }
    @rttEndTestStep; // ------------------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------------
    {
       p1.field2 = -2147483648;


       @rttCall(__rtt_return = structAccess(p1));
    }
    @rttEndTestStep; // ------------------------------------------------------

}
```

6.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of two traces (corresponding to the two test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported. In this example, since the module under test contains unreachable lines of code, the coverage is 66% for transitions with guards and 71% for total transitions. Besides this statistics the uncovered transitions are listed.

```
SOLUTION FOR FUNCTION structAccess
```

```
TRACE 3
TRACE COMPLETED

TRACE
{ (p1.field1 = 4;),
  (D_1724 = p1.field1;),
  (D_1728 = p1.field2;),
  (D_1727 = 2;),
  (return = D_1727;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 4
  feasible: 1

CONSTRAINT:
(((D_1724@12 >= 0) &&
 (D_1724@12 == 4)) &&
 (D_1728@13 > 0) &&
 (D_1728@13 == ((int) p1.field2@0)))
SOLUTION:
   p1.field2@0 = (ConcreteLattice<signed int>, 1073741824)
   D_1724@12 = (ConcreteLattice<signed int>, 4)
   D_1728@13 = (ConcreteLattice<signed int>, 1073741824)


TRACE 6
TRACE COMPLETED

TRACE
{ (p1.field1 = 4;),
  (D_1724 = p1.field1;),
  (D_1728 = p1.field2;),
  (tmp_0x40bed784.f1 = 5;),
  (tmp_0x40bed784.f2 = 7;),
  (p1.field3 = tmp_0x40bed784;),
  (D_1731 = p1.field3.f1;),
  (D_1727 = 3;),
  (return = D_1727;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 8
  feasible: 1

CONSTRAINT:
((((D_1724@12 >= 0) &&
 (D_1724@12 == 4)) &&
 (D_1728@13 <= 0) &&
 (D_1728@13 == ((int) p1.field2@0))) &&
 (D_1731@26 == 5) &&
 (D_1731@26 == 5))
SOLUTION:
   p1.field2@0 = (ConcreteLattice<signed int>, −2147483648)
   D_1724@12 = (ConcreteLattice<signed int>, 4)
   D_1728@13 = (ConcreteLattice<signed int>, −2147483648)
   D_1731@26 = (ConcreteLattice<signed int>, 5)


MAX SIZE IS STILL NOT REACHED
NO EXPANSION IS POSSIBLE ANY MORE AND NO NEW TRACE
Operation structAccess() is not covered.
Uncovered transitions:
[id=4] —— [ (D_1724 < 0) ] ——> D_1727 = 1;
[id=6] —— [ true ] ——> return = D_1727;
```

```
[id=11] ---- [ (!(D_1731 == 5)) ] ---> D_1727 = 4;
[id=13] ---- [ true ] ---> return = D_1727;

Covered:
Total transitions:       71%
Transitions with guards: 66%
```

6.4 Graphical Output

Figure 8 shows the graphical representation of the CFG corresponding to the module under test. All covered nodes and edges of this CFG are drawn blue, all uncovered are drawn red. As expected, nodes corresponding to line 15 (`return 1;`) and to line 28 (`return 4;`) as well as their incoming and outgoing edges are marked as uncovered.

7 Pointer Structure Access Resolution

This example corresponds to the example discussed in Section 5.8.1 and demonstrates the technique developed for the resolution of pointer structure accesses.

7.1 Analyzed Code

The source code listed below contains the implementation of the module under test `getAge()`. The function receives a pointer parameter `p` of type `person_t*` as input. The module under test demonstrates pointer access to a nested undefined structure field. The ranges of the global variables defining the current date are restricted by a precondition, this is the only difference to the example discussed in Section 5.8.1.

```
#include "ctgen_annotation.h"

unsigned int CURRENT_DAY;
unsigned int CURRENT_MONTH;
unsigned int CURRENT_YEAR;

struct birthday_t
{
    unsigned int day;
    unsigned int month;
    unsigned int year;
};


struct person_t {
    float weight;
    int height;
    bool  isMale;
    birthday_t birthday;
};

int getAge(person_t *p)
{
    __rtt_precondition(0 < CURRENT_DAY && CURRENT_DAY < 32 &&
                       0 < CURRENT_MONTH && CURRENT_MONTH < 13 &&
                       2000 < CURRENT_YEAR && CURRENT_YEAR < 2014);
```

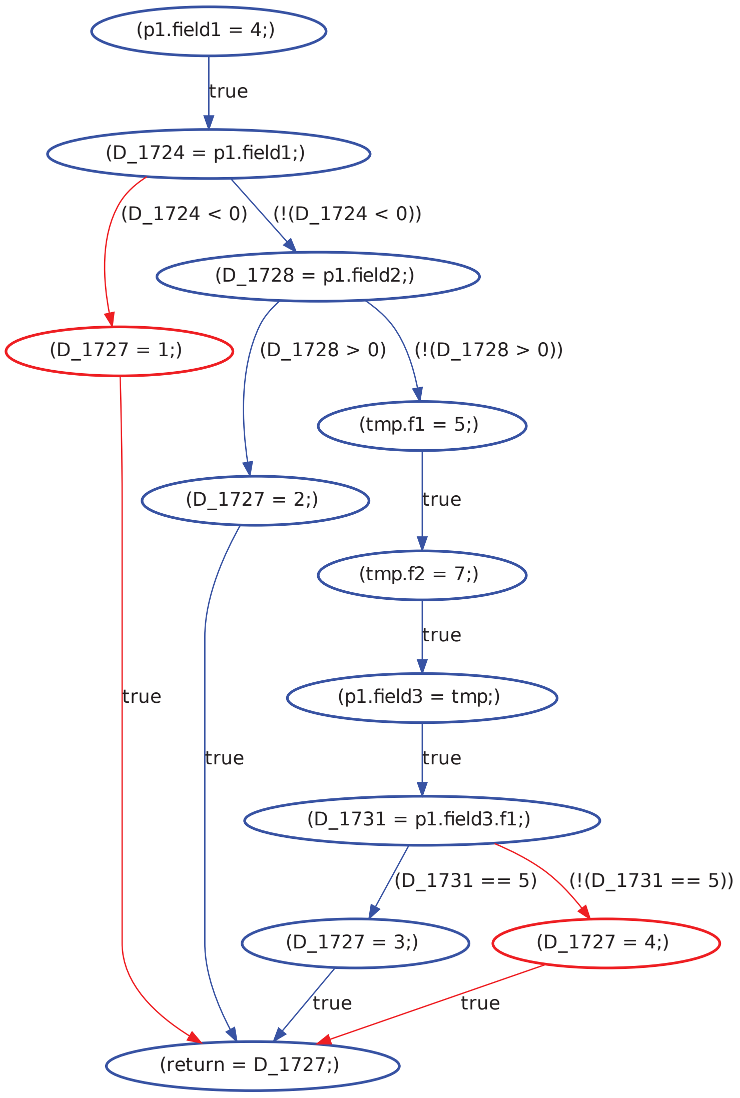


Figure 8: Graphical representation for the example *Structure Access Resolution*.

```
    int age;

    if(CURRENT_MONTH > p->birthday.month &&
       CURRENT_DAY > p->birthday.day){
        age = CURRENT_YEAR - p->birthday.year;
    } else {
        age = CURRENT_YEAR - p->birthday.year - 1;
    }

    return age;
}
```

7.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve 100% branch coverage for the module under test `getAge()`. The generator produced three test cases, therefore the test driver contains three test steps.

First, the module under test is declared. Then a variable required to pass the parameter to the module under test as well as auxiliary variable needed for the initialization of the pointer parameter and an auxiliary variable for the return value of the function are declared. In each test step the assignment of the parameter and global variables is made according to the calculated values and after the setting is done, the module under test is invoked.

```
/* ================================
 * Include section
 * ================================*/


/* ================================
 * Structures
 * ================================*/

struct birthday_t {
unsigned int day;
unsigned int month;
unsigned int year;
};
struct person_t {
float weight;
int height;
bool isMale;
birthday_t* birthday;
};

/* ================================
 * Global or static C declarations and definitions
 * ================================*/

extern int getAge(person_t* p);
@uut int getAge(person_t* p);

extern unsigned int CURRENT_DAY;
extern unsigned int CURRENT_MONTH;
extern unsigned int CURRENT_YEAR;

/* ================================
 * Abstract machine declaration.
 * ==============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:

    int __rtt_return;
    person_t p_PointsTo;
    birthday_t p_PointsTo_birthday_PointsTo;
    p_PointsTo.birthday = &p_PointsTo_birthday_PointsTo;
    person_t* p = &p_PointsTo;


    @rttBeginTestStep; // ------------------------------------------------
    {
       CURRENT_DAY = 31;
       CURRENT_DAY = 31;
       CURRENT_DAY = 31;
       CURRENT_DAY = 31;

       CURRENT_MONTH = 7;
       CURRENT_MONTH = 7;
       CURRENT_MONTH = 7;
       CURRENT_MONTH = 7;

       CURRENT_YEAR = 2007;
       CURRENT_YEAR = 2007;
       CURRENT_YEAR = 2007;

       p_PointsTo_birthday_PointsTo.month = 3;
       p_PointsTo_birthday_PointsTo.day = 15;


       @rttCall(__rtt_return = getAge(p));
    }
    @rttEndTestStep; // --------------------------------------------------

    @rttBeginTestStep; // ------------------------------------------------
    {
       CURRENT_DAY = 16;
       CURRENT_DAY = 16;
       CURRENT_DAY = 16;

       CURRENT_MONTH = 12;
       CURRENT_MONTH = 12;
       CURRENT_MONTH = 12;
       CURRENT_MONTH = 12;

       CURRENT_YEAR = 2008;
       CURRENT_YEAR = 2008;
       CURRENT_YEAR = 2008;

       p_PointsTo_birthday_PointsTo.month = 12;


       @rttCall(__rtt_return = getAge(p));
    }
    @rttEndTestStep; // ------------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------
    {
       CURRENT_DAY = 31;
       CURRENT_DAY = 31;
       CURRENT_DAY = 31;
       CURRENT_DAY = 31;

       CURRENT_MONTH = 7;
       CURRENT_MONTH = 7;
       CURRENT_MONTH = 7;
       CURRENT_MONTH = 7;

       CURRENT_YEAR = 2007;
       CURRENT_YEAR = 2007;
       CURRENT_YEAR = 2007;

       p_PointsTo_birthday_PointsTo.month = 3;
       p_PointsTo_birthday_PointsTo.day = 31;


       @rttCall(__rtt_return = getAge(p));
    }
    @rttEndTestStep; // ------------------------------------------------

}
```

7.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of three traces (corresponding to the three test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported, in this example it is 100%.

```
SOLUTION FOR FUNCTION getAge

TRACE 8
TRACE COMPLETED

TRACE
{ (__rtt_precondition_begin__();),
  (CURRENT_DAY_4 = CURRENT_DAY;),
  (CURRENT_DAY_5 = CURRENT_DAY;),
  (CURRENT_MONTH_6 = CURRENT_MONTH;),
  (CURRENT_MONTH_7 = CURRENT_MONTH;),
  (CURRENT_YEAR_8 = CURRENT_YEAR;),
  (CURRENT_YEAR_9 = CURRENT_YEAR;),
  (__rtt_precondition_end__();),
  (D_1818 = p->birthday;),
  (D_1819 = D_1818->month;),
  (CURRENT_MONTH_10 = CURRENT_MONTH;),
  (D_1822 = p->birthday;),
  (D_1823 = D_1822->day;),
  (CURRENT_DAY_11 = CURRENT_DAY;),
  (CURRENT_YEAR_12 = CURRENT_YEAR;) ,
  (D_1827 = p->birthday;) ,
  (D_1828 = D_1827->year;) ,
  (D_1829 = (CURRENT_YEAR_12 - D_1828);) ,
  (age_0x40bf0548 = ((int) D_1829);) ,
  (D_1835 = age_0x40bf0548;) ,
  (return = D_1835;) }
  traceState:      CONT_OF_ANOTHER_TRACE
  currentStepNr: 20
  feasible: 1

CONSTRAINT:
(((((((((CURRENT_DAY_4@36 != 0) &&
 (CURRENT_DAY_4@36 == ((unsigned int) CURRENT_DAY@35))) &&
 (CURRENT_DAY@35 == CURRENT_DAY@0)) &&
 (CURRENT_DAY_5@37 <= 31) &&
 (CURRENT_DAY_5@37 == ((unsigned int) CURRENT_DAY@36)) &&
 (CURRENT_DAY@36 == CURRENT_DAY@0)) &&
 (CURRENT_MONTH_6@38 != 0) &&
 (CURRENT_MONTH_6@38 == ((unsigned int) CURRENT_MONTH@37)) &&
 (CURRENT_MONTH@37 == CURRENT_MONTH@0)) &&
 (CURRENT_MONTH_7@39 <= 12) &&
 (CURRENT_MONTH_7@39 == ((unsigned int) CURRENT_MONTH@38)) &&
 (CURRENT_MONTH@38 == CURRENT_MONTH@0)) &&
 (CURRENT_YEAR_8@40 > 2000) &&
 (CURRENT_YEAR_8@40 == ((unsigned int) CURRENT_YEAR@39)) &&
 (CURRENT_YEAR@39 == CURRENT_YEAR@0)) &&
 (CURRENT_YEAR_9@41 <= 2013) &&
 (CURRENT_YEAR_9@41 == ((unsigned int) CURRENT_YEAR@40)) &&
 (CURRENT_YEAR@40 == CURRENT_YEAR@0)) &&
 (D_1819@44 < CURRENT_MONTH_10@44) &&
 (D_1819@44 == ((unsigned int) p@PointsTo_birthday@PointsTo.month@0)) &&
 (CURRENT_MONTH_10@44 == ((unsigned int) CURRENT_MONTH@43)) &&
 (CURRENT_MONTH@43 == CURRENT_MONTH@0)) &&
 (D_1823@47 < CURRENT_DAY_11@47) &&
 (D_1823@47 == ((unsigned int) p@PointsTo_birthday@PointsTo.day@0)) &&
 (CURRENT_DAY_11@47 == ((unsigned int) CURRENT_DAY@46)) &&
 (CURRENT_DAY@46 == CURRENT_DAY@0))
SOLUTION:
   CURRENT_DAY@0 = (ConcreteLattice<unsigned int>, 31)
   CURRENT_MONTH@0 = (ConcreteLattice<unsigned int>, 7)
   CURRENT_YEAR@0 = (ConcreteLattice<unsigned int>, 2007)
   p@PointsTo_birthday@PointsTo.month@0 = (ConcreteLattice<unsigned int>, 3)
   p@PointsTo_birthday@PointsTo.day@0 = (ConcreteLattice<unsigned int>, 15)
   CURRENT_DAY@35 = (ConcreteLattice<unsigned int>, 31)
   CURRENT_DAY@36 = (ConcreteLattice<unsigned int>, 31)
   CURRENT_DAY_4@36 = (ConcreteLattice<unsigned int>, 31)
   CURRENT_DAY_5@37 = (ConcreteLattice<unsigned int>, 31)
   CURRENT_MONTH@37 = (ConcreteLattice<unsigned int>, 7)
   CURRENT_MONTH@38 = (ConcreteLattice<unsigned int>, 7)
   CURRENT_MONTH_6@38 = (ConcreteLattice<unsigned int>, 7)
   CURRENT_MONTH_7@39 = (ConcreteLattice<unsigned int>, 7)
   CURRENT_YEAR@39 = (ConcreteLattice<unsigned int>, 2007)
   CURRENT_YEAR@40 = (ConcreteLattice<unsigned int>, 2007)
   CURRENT_YEAR_8@40 = (ConcreteLattice<unsigned int>, 2007)
   CURRENT_YEAR_9@41 = (ConcreteLattice<unsigned int>, 2007)
   CURRENT_MONTH@43 = (ConcreteLattice<unsigned int>, 7)
   CURRENT_MONTH_10@44 = (ConcreteLattice<unsigned int>, 7)
   D_1819@44 = (ConcreteLattice<unsigned int>, 3)
   CURRENT_DAY@46 = (ConcreteLattice<unsigned int>, 31)
   CURRENT_DAY_11@47 = (ConcreteLattice<unsigned int>, 31)
   D_1823@47 = (ConcreteLattice<unsigned int>, 15)
```

```
TRACE 10
TRACE COMPLETED

TRACE
{ (__rtt_precondition_begin__();),
  (CURRENT_DAY_4 = CURRENT_DAY;),
  (CURRENT_DAY_5 = CURRENT_DAY;),
  (CURRENT_MONTH_6 = CURRENT_MONTH;),
  (CURRENT_MONTH_7 = CURRENT_MONTH;),
  (CURRENT_YEAR_8 = CURRENT_YEAR;),
  (CURRENT_YEAR_9 = CURRENT_YEAR;),
  (__rtt_precondition_end__();),
  (D_1818 = p->birthday;),
  (D_1819 = D_1818->month;),
  (CURRENT_MONTH_10 = CURRENT_MONTH;),
  (CURRENT_YEAR_13 = CURRENT_YEAR;),
  (D_1831 = p->birthday;),
  (D_1832 = D_1831->year;),
  (D_1833 = (CURRENT_YEAR_13 - D_1832);),
  (D_1834 = (D_1833 + 4294967295);),
  (age_0x40bf0548 = ((int) D_1834);),
  (D_1835 = age_0x40bf0548;),
  (return = D_1835;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 18
  feasible: 1

CONSTRAINT:
((((((((CURRENT_DAY_4@36 != 0) &&
 (CURRENT_DAY_4@36 == ((unsigned int) CURRENT_DAY@35))) &&
 (CURRENT_DAY@35 == CURRENT_DAY@0)) &&
 (CURRENT_DAY_5@37 <= 31) &&
 (CURRENT_DAY_5@37 == ((unsigned int) CURRENT_DAY@36)) &&
 (CURRENT_DAY@36 == CURRENT_DAY@0)) &&
 (CURRENT_MONTH_6@38 != 0) &&
 (CURRENT_MONTH_6@38 == ((unsigned int) CURRENT_MONTH@37)) &&
 (CURRENT_MONTH@37 == CURRENT_MONTH@0)) &&
 (CURRENT_MONTH_7@39 <= 12) &&
 (CURRENT_MONTH_7@39 == ((unsigned int) CURRENT_MONTH@38)) &&
 (CURRENT_MONTH@38 == CURRENT_MONTH@0)) &&
 (CURRENT_YEAR_8@40 > 2000) &&
 (CURRENT_YEAR_8@40 == ((unsigned int) CURRENT_YEAR@39)) &&
 (CURRENT_YEAR@39 == CURRENT_YEAR@0)) &&
 (CURRENT_YEAR_9@41 <= 2013) &&
 (CURRENT_YEAR_9@41 == ((unsigned int) CURRENT_YEAR@40)) &&
 (CURRENT_YEAR@40 == CURRENT_YEAR@0)) &&
 (D_1819@44 >= CURRENT_MONTH_10@44) &&
 (D_1819@44 == ((unsigned int) p@PointsTo_birthday@PointsTo.month@0)) &&
 (CURRENT_MONTH_10@44 == ((unsigned int) CURRENT_MONTH@43)) &&
 (CURRENT_MONTH@43 == CURRENT_MONTH@0))
SOLUTION:
   CURRENT_DAY@0 = (ConcreteLattice<unsigned int>, 16)
   CURRENT_MONTH@0 = (ConcreteLattice<unsigned int>, 12)
   CURRENT_YEAR@0 = (ConcreteLattice<unsigned int>, 2008)
   p@PointsTo_birthday@PointsTo.month@0 = (ConcreteLattice<unsigned int>, 12)
   CURRENT_DAY@35 = (ConcreteLattice<unsigned int>, 16)
   CURRENT_DAY@36 = (ConcreteLattice<unsigned int>, 16)
   CURRENT_DAY_4@36 = (ConcreteLattice<unsigned int>, 16)
   CURRENT_DAY_5@37 = (ConcreteLattice<unsigned int>, 16)
   CURRENT_MONTH@37 = (ConcreteLattice<unsigned int>, 12)
```

```
    CURRENT_MONTH@38 = (ConcreteLattice<unsigned int>, 12)
    CURRENT_MONTH_6@38 = (ConcreteLattice<unsigned int>, 12)
    CURRENT_MONTH_7@39 = (ConcreteLattice<unsigned int>, 12)
    CURRENT_YEAR@39 = (ConcreteLattice<unsigned int>, 2008)
    CURRENT_YEAR@40 = (ConcreteLattice<unsigned int>, 2008)
    CURRENT_YEAR_8@40 = (ConcreteLattice<unsigned int>, 2008)
    CURRENT_YEAR_9@41 = (ConcreteLattice<unsigned int>, 2008)
    CURRENT_MONTH@43 = (ConcreteLattice<unsigned int>, 12)
    CURRENT_MONTH_10@44 = (ConcreteLattice<unsigned int>, 12)
    D_1819@44 = (ConcreteLattice<unsigned int>, 12)


TRACE 11
TRACE COMPLETED

TRACE
{ (__rtt_precondition_begin__();),
  (CURRENT_DAY_4 = CURRENT_DAY;),
  (CURRENT_DAY_5 = CURRENT_DAY;),
  (CURRENT_MONTH_6 = CURRENT_MONTH;),
  (CURRENT_MONTH_7 = CURRENT_MONTH;),
  (CURRENT_YEAR_8 = CURRENT_YEAR;),
  (CURRENT_YEAR_9 = CURRENT_YEAR;),
  (__rtt_precondition_end__();),
  (D_1818 = p->birthday;),
  (D_1819 = D_1818->month;),
  (CURRENT_MONTH_10 = CURRENT_MONTH;),
  (D_1822 = p->birthday;),
  (D_1823 = D_1822->day;),
  (CURRENT_DAY_11 = CURRENT_DAY;),
  (CURRENT_YEAR_13 = CURRENT_YEAR;) }
  traceState:    CONT_AFTER_SOLVING
  currentStepNr: 14
  feasible: 1

CONSTRAINT:
(((((((((((CURRENT_DAY_4@36 != 0) &&
 (CURRENT_DAY_4@36 == ((unsigned int) CURRENT_DAY@35))) &&
 (CURRENT_DAY@35 == CURRENT_DAY@0)) &&
 (CURRENT_DAY_5@37 <= 31) &&
 (CURRENT_DAY_5@37 == ((unsigned int) CURRENT_DAY@36)) &&
 (CURRENT_DAY@36 == CURRENT_DAY@0)) &&
 (CURRENT_MONTH_6@38 != 0) &&
 (CURRENT_MONTH_6@38 == ((unsigned int) CURRENT_MONTH@37)) &&
 (CURRENT_MONTH@37 == CURRENT_MONTH@0)) &&
 (CURRENT_MONTH_7@39 <= 12) &&
 (CURRENT_MONTH_7@39 == ((unsigned int) CURRENT_MONTH@38)) &&
 (CURRENT_MONTH@38 == CURRENT_MONTH@0)) &&
 (CURRENT_YEAR_8@40 > 2000) &&
 (CURRENT_YEAR_8@40 == ((unsigned int) CURRENT_YEAR@39)) &&
 (CURRENT_YEAR@39 == CURRENT_YEAR@0)) &&
 (CURRENT_YEAR_9@41 <= 2013) &&
 (CURRENT_YEAR_9@41 == ((unsigned int) CURRENT_YEAR@40)) &&
 (CURRENT_YEAR@40 == CURRENT_YEAR@0)) &&
 (D_1819@44 < CURRENT_MONTH_10@44) &&
 (D_1819@44 == ((unsigned int) p@PointsTo_birthday@PointsTo.month@0)) &&
 (CURRENT_MONTH_10@44 == ((unsigned int) CURRENT_MONTH@43)) &&
 (CURRENT_MONTH@43 == CURRENT_MONTH@0)) &&
 (D_1823@47 >= CURRENT_DAY_11@47) &&
 (D_1823@47 == ((unsigned int) p@PointsTo_birthday@PointsTo.day@0)) &&
 (CURRENT_DAY_11@47 == ((unsigned int) CURRENT_DAY@46)) &&
 (CURRENT_DAY@46 == CURRENT_DAY@0))
```

```
SOLUTION:
    CURRENT_DAY@0 = (ConcreteLattice<unsigned int>, 31)
    CURRENT_MONTH@0 = (ConcreteLattice<unsigned int>, 7)
    CURRENT_YEAR@0 = (ConcreteLattice<unsigned int>, 2007)
    p@PointsTo_birthday@PointsTo.month@0 = (ConcreteLattice<unsigned int>, 3)
    p@PointsTo_birthday@PointsTo.day@0 = (ConcreteLattice<unsigned int>, 31)
    CURRENT_DAY@35 = (ConcreteLattice<unsigned int>, 31)
    CURRENT_DAY@36 = (ConcreteLattice<unsigned int>, 31)
    CURRENT_DAY_4@36 = (ConcreteLattice<unsigned int>, 31)
    CURRENT_DAY_5@37 = (ConcreteLattice<unsigned int>, 31)
    CURRENT_MONTH@37 = (ConcreteLattice<unsigned int>, 7)
    CURRENT_MONTH@38 = (ConcreteLattice<unsigned int>, 7)
    CURRENT_MONTH_6@38 = (ConcreteLattice<unsigned int>, 7)
    CURRENT_MONTH_7@39 = (ConcreteLattice<unsigned int>, 7)
    CURRENT_YEAR@39 = (ConcreteLattice<unsigned int>, 2007)
    CURRENT_YEAR@40 = (ConcreteLattice<unsigned int>, 2007)
    CURRENT_YEAR_8@40 = (ConcreteLattice<unsigned int>, 2007)
    CURRENT_YEAR_9@41 = (ConcreteLattice<unsigned int>, 2007)
    CURRENT_MONTH@43 = (ConcreteLattice<unsigned int>, 7)
    CURRENT_MONTH_10@44 = (ConcreteLattice<unsigned int>, 7)
    D_1819@44 = (ConcreteLattice<unsigned int>, 3)
    CURRENT_DAY@46 = (ConcreteLattice<unsigned int>, 31)
    CURRENT_DAY_11@47 = (ConcreteLattice<unsigned int>, 31)
    D_1823@47 = (ConcreteLattice<unsigned int>, 31)


Operation getAge() is covered.

Covered:
Total transitions:        100%
Transitions with guards:  100%
```

7.4 Graphical Output

Figure 9 shows the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue, which indicates that all of them are successfully covered.

8 Processing Bit Fields

This example corresponds to the example discussed in Section 5.9 and demonstrates the technique developed for the handling of bit fields.

8.1 Analyzed Code

The source code listed below contains the implementation of the module under test `test()`. The function evaluates the global variable `globalBF`, which consists of 12 bit fields. The module under test demonstrates how the generator processes bit fields.

```c
typedef unsigned char uint8_t;
typedef struct bitfield_t {
    uint8_t bit1:1;
    uint8_t bit2:1;
    uint8_t bit3:1;
    uint8_t bit4:1;
    uint8_t bit5:1;
```

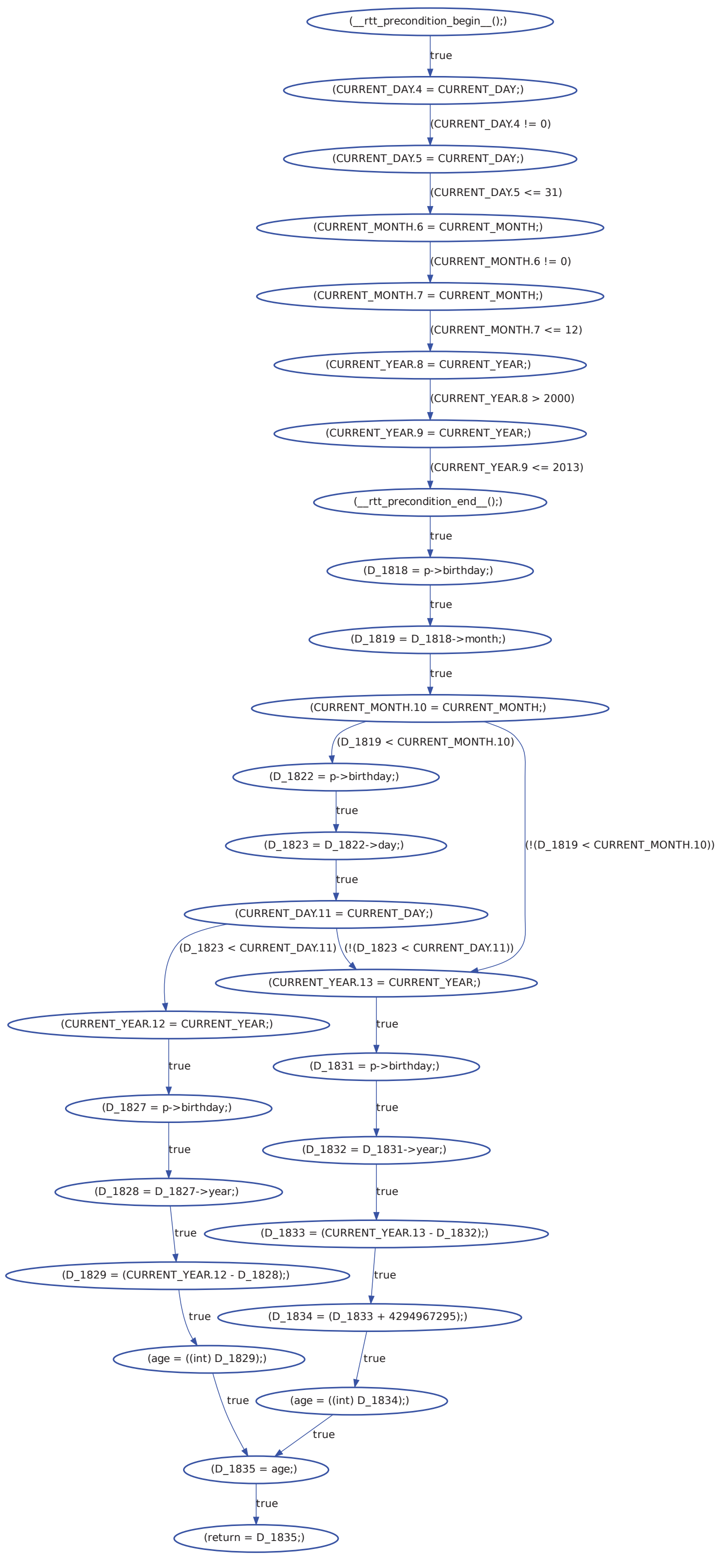

(__rtt_precondition_begin__();)
true
(CURRENT_DAY.4 = CURRENT_DAY;)
(CURRENT_DAY.4 != 0)
(CURRENT_DAY.5 = CURRENT_DAY;)
(CURRENT_DAY.5 <= 31)
(CURRENT_MONTH.6 = CURRENT_MONTH;)
(CURRENT_MONTH.6 != 0)
(CURRENT_MONTH.7 = CURRENT_MONTH;)
(CURRENT_MONTH.7 <= 12)
(CURRENT_YEAR.8 = CURRENT_YEAR;)
(CURRENT_YEAR.8 > 2000)
(CURRENT_YEAR.9 = CURRENT_YEAR;)
(CURRENT_YEAR.9 <= 2013)
(__rtt_precondition_end__();)
true
(D_1818 = p->birthday;)
true
(D_1819 = D_1818->month;)
true
(CURRENT_MONTH.10 = CURRENT_MONTH;)
(D_1819 < CURRENT_MONTH.10)
(!(D_1819 < CURRENT_MONTH.10))
(D_1822 = p->birthday;)
true
(D_1823 = D_1822->day;)
true
(CURRENT_DAY.11 = CURRENT_DAY;)
(D_1823 < CURRENT_DAY.11)
(!(D_1823 < CURRENT_DAY.11))
(CURRENT_YEAR.13 = CURRENT_YEAR;)
(CURRENT_YEAR.12 = CURRENT_YEAR;)
true
(D_1831 = p->birthday;)
true
(D_1827 = p->birthday;)
true
(D_1832 = D_1831->year;)
true
(D_1828 = D_1827->year;)
true
(D_1833 = (CURRENT_YEAR.13 - D_1832);)
true
(D_1829 = (CURRENT_YEAR.12 - D_1828);)
true
(D_1834 = (D_1833 + 4294967295);)
true
(age = ((int) D_1829);)
true
(age = ((int) D_1834);)
true
(D_1835 = age;)
true
(return = D_1835;)

Figure 9: Graphical representation for the example *Pointer Structure Access Resolution*.

```
    uint8_t bit6:1;
    uint8_t bit7:1;
    uint8_t bit8:1;
    uint8_t bit9:1;
    uint8_t bit10:1;
    uint8_t bit11:1;
    uint8_t bit12:1;
} bitfield_t;

bitfield_t globalBF;

int test()
{
    int retval = 0;
    globalBF.bit11 = 1;
    globalBF.bit5 = 0;
    if(globalBF.bit11 && globalBF.bit5){
        retval = 1;
    } else if(globalBF.bit2){
        retval = 2;
    }

    return retval;
}
```

8.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve the maximal possible branch coverage for the module under test `test()`. The generator produced two test cases, therefore the test driver contains two test steps. Since no specification for the module under test was given, the test driver contains no assertions. The function `test()` contains no defined or undefined function calls, therefore no stub functions are generated.

First, the module under test and the global variable `globalBF` are declared. In each test step the assignment of the global variable is made according to the calculated values and after the setting is done, the module under test is invoked.

```
/* ==============================
 * Include section
 * ==============================*/


/* ==============================
 * Structures
 * ==============================*/

struct bitfield_t {
uint8_t bit1;
uint8_t bit2;
uint8_t bit3;
uint8_t bit4;
uint8_t bit5;
uint8_t bit6;
uint8_t bit7;
uint8_t bit8;
uint8_t bit9;
uint8_t bit10;
uint8_t bit11;
```

```
uint8_t bit12;
};

/* ==============================
 * Global or static C declarations and definitions
 * ==============================*/

extern int test();
@uut int test();

extern bitfield_t globalBF;

/* ==============================
 * Abstract machine declaration.
 * ==============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:

    int __rtt_return;


    @rttBeginTestStep; // ----------------------------------------------
    {
       globalBF.bit12 = 31
;
       globalBF.bit9 = 0;
       globalBF.bit10 = 0;
       globalBF.bit11 = 1;
       globalBF.bit12 = 31;

       globalBF.bit1 = 0;
       globalBF.bit2 = 1;
       globalBF.bit3 = 0;
       globalBF.bit4 = 0;
       globalBF.bit5 = 0;
       globalBF.bit6 = 0;
       globalBF.bit7 = 0;
       globalBF.bit8 = 0;



       @rttCall(__rtt_return = test());
    }
    @rttEndTestStep; // ----------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------
    {
       globalBF.bit12 = 31
;
       globalBF.bit9 = 0;
       globalBF.bit10 = 0;
       globalBF.bit11 = 1;
       globalBF.bit12 = 31;

       globalBF.bit1 = 0;
```

```
        globalBF.bit2 = 0;
        globalBF.bit3 = 0;
        globalBF.bit4 = 0;
        globalBF.bit5 = 0;
        globalBF.bit6 = 0;
        globalBF.bit7 = 0;
        globalBF.bit8 = 0;

        @rttCall(__rtt_return = test());
    }
    @rttEndTestStep; // ------------------------------------------------

}
```

8.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of two traces (corresponding to the two test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported. In this example, since the module under test contains unreachable branches, the coverage is 66% for transitions with guards and 80% for the total transitions. Furthermore, the uncovered transitions are listed.

```
SOLUTION FOR FUNCTION test

TRACE 4
TRACE COMPLETED

TRACE
{ (retval_0x40bee8f0 = 0;),
  (globalBF.bit11 = 1;),
  (globalBF.bit5 = 0;),
  (D_1729 = BIT_FIELD_REF(globalBF, 8, 8);),
  (D_1730 = (D_1729 & 4);),
  (D_1732 = BIT_FIELD_REF(globalBF, 8, 0);),
  (D_1733 = (D_1732 & 16);),
  (D_1735 = BIT_FIELD_REF(globalBF, 8, 0);),
  (D_1736 = (D_1735 & 2);),
  (retval_0x40bee8f0 = 2;),
  (D_1740 = retval_0x40bee8f0;),
  (return = D_1740;) }
  traceState:     CONT_OF_ANOTHER_TRACE
  currentStepNr: 11
  feasible: 1

CONSTRAINT:
(((((D_1730@22 != 0) &&
 (D_1730@22 == ((unsigned char) (D_1729@21 & 4)))) &&
 ((D_1729@21 & 248) == (globalBF.bit12@0 << 3)) &&
 ((D_1729@21 & 4) == 4) &&
 (D_1729@21 == ((unsigned char) rttExtract(globalBF@17, 15, 8)))) &&
 (D_1733@24 == 0) &&
 (D_1733@24 == ((unsigned char) (D_1732@23 & 16))) &&
 ((D_1732@23 & 16) == 0) &&
 (D_1732@23 == ((unsigned char) rttExtract(globalBF@17, 7, 0)))) &&
 (D_1736@26 != 0) &&
 (D_1736@26 == ((unsigned char) (D_1735@25 & 2))) &&
  ((D_1735@25 & 16) == 0) &&
  (D_1735@25 == ((unsigned char) rttExtract(globalBF@17, 7, 0))))
SOLUTION:
    globalBF.bit12@0 = (ConcreteLattice<signed short int>, 31)
        globalBF.bit9@21 = 0;
        globalBF.bit10@21 = 0;
        globalBF.bit11@21 = 1;
        globalBF.bit12@21 = 31;

    D_1730@22 = (ConcreteLattice<signed short int>, 4)
        globalBF.bit1@23 = 0;
        globalBF.bit2@23 = 1;
        globalBF.bit3@23 = 0;
        globalBF.bit4@23 = 0;
        globalBF.bit5@23 = 0;
        globalBF.bit6@23 = 0;
        globalBF.bit7@23 = 0;
        globalBF.bit8@23 = 0;
    D_1733@24 = (ConcreteLattice<signed short int>, 0)
        globalBF.bit1@25 = 0;
        globalBF.bit2@25 = 1;
        globalBF.bit3@25 = 0;
        globalBF.bit4@25 = 0;
        globalBF.bit5@25 = 0;
        globalBF.bit6@25 = 0;
        globalBF.bit7@25 = 0;
        globalBF.bit8@25 = 0;
    D_1736@26 = (ConcreteLattice<signed short int>, 2)

TRACE 7
TRACE COMPLETED

TRACE
{ (retval_0x40bee8f0 = 0;),
   (globalBF.bit11 = 1;),
   (globalBF.bit5 = 0;),
   (D_1729 = BIT_FIELD_REF(globalBF, 8, 8);),
   (D_1730 = (D_1729 & 4);),
   (D_1732 = BIT_FIELD_REF(globalBF, 8, 0);),
   (D_1733 = (D_1732 & 16);),
   (D_1735 = BIT_FIELD_REF(globalBF, 8, 0);),
   (D_1736 = (D_1735 & 2);),
   (D_1740 = retval_0x40bee8f0;),
   (return = D_1740;) }
   traceState:     CONT_OF_ANOTHER_TRACE
   currentStepNr: 10
   feasible: 1

CONSTRAINT:
(((((D_1730@22 != 0) &&
  (D_1730@22 == ((unsigned char) (D_1729@21 & 4)))) &&
  ((D_1729@21 & 248) == (globalBF.bit12@0 << 3)) &&
  ((D_1729@21 & 4) == 4) &&
  (D_1729@21 == ((unsigned char) rttExtract(globalBF@17, 15, 8)))) &&
  (D_1733@24 == 0) &&
  (D_1733@24 == ((unsigned char) (D_1732@23 & 16))) &&
  ((D_1732@23 & 16) == 0) &&
  (D_1732@23 == ((unsigned char) rttExtract(globalBF@17, 7, 0)))) &&
  (D_1736@26 == 0) &&
  (D_1736@26 == ((unsigned char) (D_1735@25 & 2))) &&
  ((D_1735@25 & 16) == 0) &&
  (D_1735@25 == ((unsigned char) rttExtract(globalBF@17, 7, 0))))
```

```
SOLUTION:
   globalBF.bit12@0 = (ConcreteLattice<signed short int>, 31)
       globalBF.bit9@21 = 0;
       globalBF.bit10@21 = 0;
       globalBF.bit11@21 = 1;
       globalBF.bit12@21 = 31;

   D_1730@22 = (ConcreteLattice<signed short int>, 4)
       globalBF.bit1@23 = 0;
       globalBF.bit2@23 = 0;
       globalBF.bit3@23 = 0;
       globalBF.bit4@23 = 0;
       globalBF.bit5@23 = 0;
       globalBF.bit6@23 = 0;
       globalBF.bit7@23 = 0;
       globalBF.bit8@23 = 0;
   D_1733@24 = (ConcreteLattice<signed short int>, 0)
       globalBF.bit1@25 = 0;
       globalBF.bit2@25 = 0;
       globalBF.bit3@25 = 0;
       globalBF.bit4@25 = 0;
       globalBF.bit5@25 = 0;
       globalBF.bit6@25 = 0;
       globalBF.bit7@25 = 0;
       globalBF.bit8@25 = 0;
   D_1736@26 = (ConcreteLattice<signed short int>, 0)

MAX SIZE IS STILL NOT REACHED
NO EXPANSION IS POSSIBLE ANY MORE AND NO NEW TRACE
 Operation test() is not covered.
 Uncovered transitions:
 [id=8] --- [ (!(D_1730 != 0)) ]  ---> D_1735 = BIT_FIELD_REF(globalBF, 8, 0);
 [id=9] --- [ (D_1733 != 0) ]  ---> retval_0x40bee8f0 = 1;
 [id=11] --- [ true ]  ---> D_1740 = retval_0x40bee8f0;

 Covered:
 Total transitions:        80%
 Transitions with guards: 66%
```

8.4 Graphical Output

Figure 10 shows the graphical representation of the CFG corresponding to the module under test. All covered nodes and edges of this CFG are drawn blue, all uncovered ones are drawn red. As expected, the node corresponding to line 25 (`retval = 1;`) as well as its incoming and outgoing edges are marked as uncovered. The edge corresponding to the evaluation of the bitfield `globalBF.bit11` to zero is also marked as uncovered.

9 Processing Unions (Example 1)

This example corresponds to the example discussed in Section 5.10 and demonstrates the technique developed for the handling of unions, in particular the case of access to a smaller union member after the assignment of a bigger one.

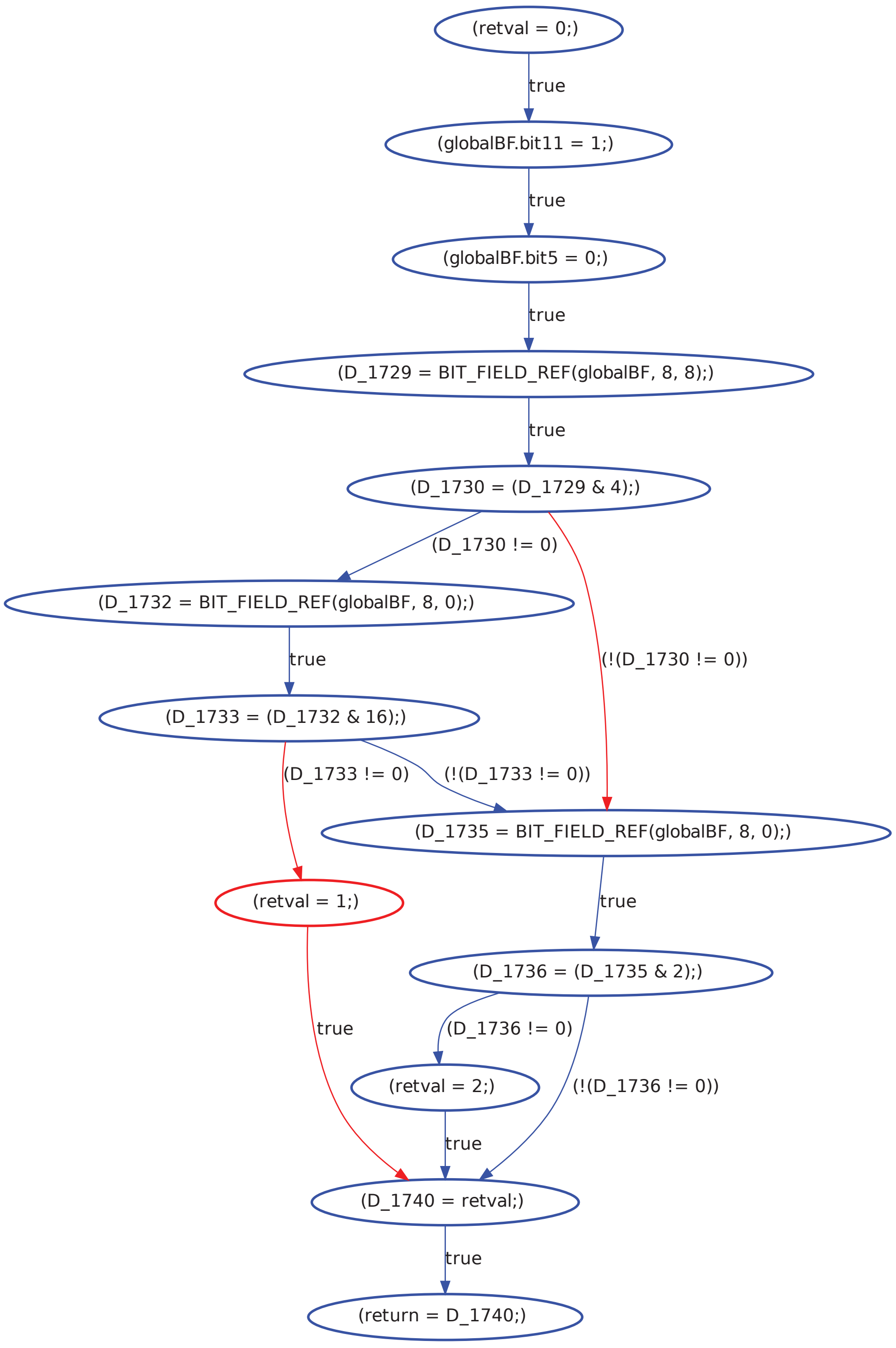


Figure 10: Graphical representation for the example *Bitfields*.

9.1 Analyzed Code

The source code listed below contains the implementation of the module under test `test_sym1()`. By the example of the module under test we demonstrate the algorithm developed for the resolution of union access.

```
typedef union {
    unsigned short  c2u16;
    unsigned char c2u8[2];
} union_u16;

union_u16 globalV;
int test_sym1(unsigned short x)
{
    globalV.c2u16 = x;
    if(globalV.c2u8[0] == 0xff && globalV.c2u8[1] == 85){
        return 1;
    }
    return 0;
}
```

9.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve the maximal possible branch coverage for the module under test `test_sym1()`. The generator produced three test cases, therefore the test driver contains three test steps. The test driver assigns the input parameter `x` according to the solution calculated by the solver in each test step before the call to the module under test. Since no specification of the module under test was given, the test driver contains no assertions. The function `test_sym1()` contains no defined or undefined function calls, therefore no stub functions are generated.

```
/* ===============================
 * Include section
 * ===============================*/
/* ===============================
 * Structures
 * ===============================*/
/* ===============================
 * Global or static C declarations and definitions
 * ===============================*/

extern int test_sym1(short unsigned int x);
@uut int test_sym1(short unsigned int x);

/* ===============================
 * Abstract machine declaration.
 * ===============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:
    int __rtt_return;
    short unsigned int x;


    @rttBeginTestStep; // -----------------------------------------------
    {
       x = 22015;
       @rttCall(__rtt_return = test_sym1(x));
    }
    @rttEndTestStep; // ------------------------------------------------

    @rttBeginTestStep; // -----------------------------------------------
    {
       x = 191;
       @rttCall(__rtt_return = test_sym1(x));
    }
    @rttEndTestStep; // ------------------------------------------------

    @rttBeginTestStep; // -----------------------------------------------
    {
       x = 54783;
       @rttCall(__rtt_return = test_sym1(x));
    }
    @rttEndTestStep; // ------------------------------------------------

}
```

9.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of three traces (corresponding to the three test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported. In this example the coverage is 100%.

```
SOLUTION FOR FUNCTION test_sym1

TRACE 2
TRACE COMPLETED

TRACE
{ (globalV.c2u16 = x;),
  (D_1732 = globalV.c2u8[0];),
  (D_1735 = globalV.c2u8[1];),
  (D_1738 = 1;),
  (return = D_1738;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 4
  feasible: 1

CONSTRAINT:
((((D_1732@11 == 255) &&
 (D_1732@11 == ((unsigned char) ((unsigned char) ((x@6 >> 0) & 255))))) &&
 (x@6 == ((short unsigned int) x@0))) &&
 (D_1735@12 == 85) &&
 (D_1735@12 == ((unsigned char) ((unsigned char) ((x@6 >> 8) & 255)))))
SOLUTION:
    x@0 = (ConcreteLattice<unsigned short int>, 22015)
    x@6 = (ConcreteLattice<unsigned short int>, 22015)
    D_1732@11 = (ConcreteLattice<signed short int>, 255)
    D_1735@12 = (ConcreteLattice<signed short int>, 85)


TRACE 4
TRACE COMPLETED

TRACE
{ (globalV.c2u16 = x;),
   (D_1732 = globalV.c2u8[0];),
   (D_1738 = 0;),
   (return = D_1738;) }
   traceState:     CONT_OF_ANOTHER_TRACE
   currentStepNr: 3
   feasible: 1

CONSTRAINT:
(((D_1732@11 != 255) &&
 (D_1732@11 == ((unsigned char) ((unsigned char) ((x@6 >> 0) & 255))))) &&
 (x@6 == ((short unsigned int) x@0)))
SOLUTION:
    x@0 = (ConcreteLattice<unsigned short int>, 191)
    x@6 = (ConcreteLattice<unsigned short int>, 191)
    D_1732@11 = (ConcreteLattice<signed short int>, 191)


TRACE 5
TRACE COMPLETED

TRACE
{ (globalV.c2u16 = x;),
   (D_1732 = globalV.c2u8[0];),
   (D_1735 = globalV.c2u8[1];),
   (D_1738 = 0;) }
   traceState:     CONT_AFTER_SOLVING
   currentStepNr: 3
   feasible: 1

CONSTRAINT:
((((D_1732@11 == 255) &&
 (D_1732@11 == ((unsigned char) ((unsigned char) ((x@6 >> 0) & 255))))) &&
 (x@6 == ((short unsigned int) x@0))) &&
 (D_1735@12 != 85) &&
 (D_1735@12 == ((unsigned char) ((unsigned char) ((x@6 >> 8) & 255)))))
SOLUTION:
    x@0 = (ConcreteLattice<unsigned short int>, 54783)
    x@6 = (ConcreteLattice<unsigned short int>, 54783)
    D_1732@11 = (ConcreteLattice<signed short int>, 255)
    D_1735@12 = (ConcreteLattice<signed short int>, 213)


Operation test_sym1() is covered.

Covered:
Total transitions:        100%
Transitions with guards: 100%
```

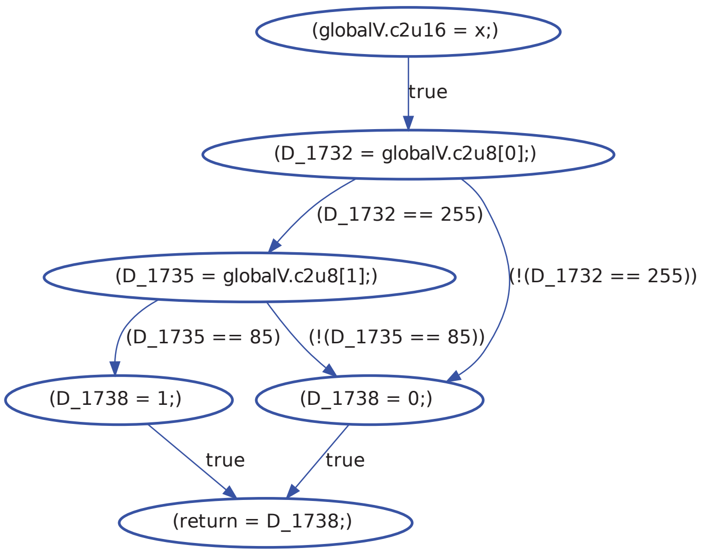


Figure 11: Graphical representation for the example *Unions (1)*.

9.4 Graphical Output

Figure 11 shows the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue since the generator was able to achieve 100% branch coverage.

10 Processing Unions (Example 2)

This example corresponds to the example discussed in Section 5.10 and demonstrates the technique developed for the handling of unions, in particular the case of access to a bigger member after the assignment of small ones.

10.1 Analyzed Code

The source code listed below contains the implementation of the module under test `test_sym2()`. By the example of the module under test we demonstrate the algorithm developed for the resolution of union access.

```
typedef union {
    unsigned short  c2u16;
    unsigned char c2u8[2];
} union_u16;

union_u16 globalV;
int test_sym2(unsigned char x, unsigned char y)
```

```
{
    globalV.c2u8[0] = x;
    globalV.c2u8[1] = y;

    if(globalV.c2u16 == 0x5555){
        return 1;
    }
    return 0;

}
```

10.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve the maximal possible branch coverage for the module under test `test_sym2()`. The generator produced two test cases, therefore the test driver contains two test steps. The test driver makes assignments of the parameters `x` and `y` according to the solution calculated by the solver in each test step before the call to the module under test. Since no specification of the module under test was given, the test driver contains no assertions. The function `test_sym2()` contains no defined or undefined function calls, therefore no stub functions are generated.

```
/* ===============================
 * Include section
 * ===============================*/
/* ===============================
 * Structures
 * ===============================*/
/* ===============================
 * Global or static C declarations and definitions
 * ===============================*/

extern int test_sym2(unsigned char x, unsigned char y);
@uut int test_sym2(unsigned char x, unsigned char y);

/* ===============================
 * Abstract machine declaration.
 * ===============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:

    int __rtt_return;
    unsigned char x;
    unsigned char y;

    @rttBeginTestStep; // ---------------------------------------------
    {
       x = 85;
       y = 85;
       @rttCall(__rtt_return = test_sym2(x, y));
    }
    @rttEndTestStep; // -----------------------------------------------
    @rttBeginTestStep; // ----------------------------------------------------------
    {
      x = 85;
      y = 213;
      @rttCall(__rtt_return = test_sym2(x, y));
    }
    @rttEndTestStep; // ------------------------------------------------------------

}
```

10.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of two traces (corresponding to the two test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported. In this example the coverage is 100%.

```
SOLUTION FOR FUNCTION test_sym2

TRACE 1
TRACE COMPLETED

TRACE
{ (globalV.c2u8[0] = x;),
  (globalV.c2u8[1] = y;),
  (D_1727 = globalV.c2u16;),
  (D_1730 = 1;),
  (return = D_1730;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 4
  feasible: 1

CONSTRAINT:
((((D_1727@15 == 21845) &&
 (D_1727@15 == ((short unsigned int) (((short unsigned int) (x@6 << 0)) | ((short unsigned
      int) (y@10 << 8)))))) &&
 (x@6 == ((unsigned char) x@0))) &&
 (y@10 == ((unsigned char) y@0)))
SOLUTION:
   x@0 = (ConcreteLattice<signed short int>, 85)
   y@0 = (ConcreteLattice<signed short int>, 85)
   x@6 = (ConcreteLattice<signed short int>, 85)
   y@10 = (ConcreteLattice<signed short int>, 85)
   D_1727@15 = (ConcreteLattice<unsigned short int>, 21845)


TRACE 3
TRACE COMPLETED

TRACE
{ (globalV.c2u8[0] = x;),
  (globalV.c2u8[1] = y;),
  (D_1727 = globalV.c2u16;),
  (D_1730 = 0;),
  (return = D_1730;) }
```

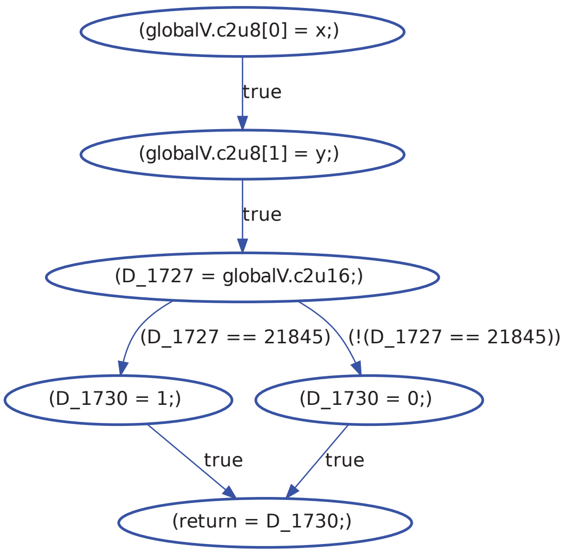


Figure 12: Graphical representation for the example *Unions (2)*.

```
   traceState:     CONT_OF_ANOTHER_TRACE
   currentStepNr: 4
   feasible: 1

CONSTRAINT:
((((D_1727@15 != 21845) &&
 (D_1727@15 == ((short unsigned int) (((short unsigned int) (x@6 << 0)) | ((short unsigned
       int) (y@10 << 8)))))) &&
 (x@6 == ((unsigned char) x@0))) &&
 (y@10 == ((unsigned char) y@0)))
SOLUTION:
    x@0 = (ConcreteLattice<signed short int>, 85)
    y@0 = (ConcreteLattice<signed short int>, 213)
    x@6 = (ConcreteLattice<signed short int>, 85)
    y@10 = (ConcreteLattice<signed short int>, 213)
    D_1727@15 = (ConcreteLattice<unsigned short int>, 54613)


Operation test_sym2() is covered.

Covered:
Total transitions:         100%
Transitions with guards: 100%
```

10.4 Graphical Output

Figure 12 demonstrates the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue since the generator was able to achieve 100% branch coverage.

11 Pointer Union Access Resolution

This example corresponds to the example discussed in Section 5.10.3 and demonstrates the technique developed for the resolution of pointer union accesses.

11.1 Analyzed Code

The source code listed below contains the implementation of the module under test `union_ptr1()`. The function receives a parameter `x` of type `unsigned short` and a global variable `globalP` of a union pointer type as input. The module under test demonstrates pointer access to a union member.

```
typedef union {
    unsigned short  c2u16;
    unsigned char c2u8[2];
} union_u16;

union_u16 globalV;
union_u16 *globalP = &globalV;

int union_ptr1(unsigned short x)
{
    globalP->c2u16 = x;
    if(globalP->c2u8[0] == 0xff && globalP->c2u8[1] == 85){
        return 1;
    }
    return 0;
}
```

11.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve 100% branch coverage for the module under test `union_ptr1()`. The generator produced three test cases, therefore the test driver contains three test steps. Since no specification for the module under test was given, the test driver contains no assertions. The function `union_ptr1()` contains no defined or undefined function calls, therefore no stub functions are generated.

First, the module under test is declared. Then a variable required to pass the parameter to the module under test as well as an auxiliary variable for the return value of the function are declared. In each test step the assignment of the parameter is made according to the calculated values and after the setting is done, the module under test is invoked.

```
/* ==============================
 * Include section
 * ==============================*/


/* ==============================
 * Structures
 * ==============================*/
```

```
/* ==============================
 * Global or static C declarations and definitions
 * ==============================*/

extern int union_ptr1(short unsigned int x);
@uut int union_ptr1(short unsigned int x);


/* ==============================
 * Abstract machine declaration.
 * ==============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:

    int __rtt_return;
    short unsigned int x;


    @rttBeginTestStep; // ----------------------------------------------------------------
    {
       x = 22015;
       @rttCall(__rtt_return = union_ptr1(x));
    }
    @rttEndTestStep; // ----------------------------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------------------------
    {
       x = 191;
       @rttCall(__rtt_return = union_ptr1(x));
    }
    @rttEndTestStep; // ----------------------------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------------------------
    {
       x = 54783;
       @rttCall(__rtt_return = union_ptr1(x));
    }
    @rttEndTestStep; // ----------------------------------------------------------------

}
```

11.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of three traces (corresponding to the three test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is shown.

At the bottom of the solution file the statistics of the achieved coverage is reported. In this example it is 100%.

```
SOLUTION FOR FUNCTION union_ptr1
```

```
TRACE 2
TRACE COMPLETED

TRACE
{ (globalP_2 = globalP;),
  (globalP_2->c2u16 = x;),
  (globalP_3 = globalP;),
  (D_1754 = globalP_3->c2u8[0];),
  (globalP_4 = globalP;),
  (D_1758 = globalP_4->c2u8[1];),
  (D_1761 = 1;),
  (return = D_1761;) }
  traceState:     CONT_OF_ANOTHER_TRACE
  currentStepNr: 7
  feasible: 1

CONSTRAINT:
((((D_1754@23 == 255) &&
 (D_1754@23 == ((unsigned char) ((unsigned char) ((x@17 >> 0) & 255))))) &&
 (x@17 == ((short unsigned int) x@0))) &&
 (D_1758@25 == 85) &&
 (D_1758@25 == ((unsigned char) ((unsigned char) ((x@17 >> 8) & 255)))))
SOLUTION:
   x@0 = (ConcreteLattice<unsigned short int>, 22015)
   x@17 = (ConcreteLattice<unsigned short int>, 22015)
   D_1754@23 = (ConcreteLattice<signed short int>, 255)
   D_1758@25 = (ConcreteLattice<signed short int>, 85)


TRACE 4
TRACE COMPLETED

TRACE
{ (globalP_2 = globalP;),
  (globalP_2->c2u16 = x;),
  (globalP_3 = globalP;),
  (D_1754 = globalP_3->c2u8[0];),
  (D_1761 = 0;),
  (return = D_1761;) }
  traceState:     CONT_OF_ANOTHER_TRACE
  currentStepNr: 5
  feasible: 1

CONSTRAINT:
(((D_1754@23 != 255) &&
 (D_1754@23 == ((unsigned char) ((unsigned char) ((x@17 >> 0) & 255))))) &&
 (x@17 == ((short unsigned int) x@0)))
SOLUTION:
   x@0 = (ConcreteLattice<unsigned short int>, 191)
   x@17 = (ConcreteLattice<unsigned short int>, 191)
   D_1754@23 = (ConcreteLattice<signed short int>, 191)


TRACE 5
TRACE COMPLETED

TRACE
{ (globalP_2 = globalP;),
  (globalP_2->c2u16 = x;),
  (globalP_3 = globalP;),
  (D_1754 = globalP_3->c2u8[0];),
    (globalP_4 = globalP;),
    (D_1758 = globalP_4->c2u8[1];),
    (D_1761 = 0;) }
    traceState:     CONT_AFTER_SOLVING
    currentStepNr: 6
    feasible: 1

CONSTRAINT:
((((D_1754@23 == 255) &&
 (D_1754@23 == ((unsigned char) ((unsigned char) ((x@17 >> 0) & 255))))) &&
 (x@17 == ((short unsigned int) x@0))) &&
 (D_1758@25 != 85) &&
 (D_1758@25 == ((unsigned char) ((unsigned char) ((x@17 >> 8) & 255)))))
SOLUTION:
    x@0 = (ConcreteLattice<unsigned short int>, 54783)
    x@17 = (ConcreteLattice<unsigned short int>, 54783)
    D_1754@23 = (ConcreteLattice<signed short int>, 255)
    D_1758@25 = (ConcreteLattice<signed short int>, 213)


Operation union_ptr1() is covered.

Covered:
Total transitions:        100%
Transitions with guards: 100%
```

11.4 Graphical Output

Figure 13 shows the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue, which indicates that all of them were successfully covered.

12 Processing Input Arrays

This example corresponds to the example discussed in Section 5.11.1 and demonstrates the technique developed for handling input arrays.

12.1 Analyzed Code

The source code listed below contains the implementation of the module under test `test()`. The module under test `test()` compares two elements of the integer array `a[]` and returns *true*, if the element `a[x]` is greater then the element `a[y]` and *false* otherwise. The array `a[]` as well as the indices `x` and `y` are inputs. To ensure, that the passed values of `x` and `y` are within the array bounds we specified a precondition.

```
#include "ctgen_annotation.h"
#define N 2
typedef int my_array[N];
int test(my_array a, unsigned int x, unsigned int y)
{
    __rtt_precondition(x < N && y < N);

    int retval = 0;
    if(a[x] > a[y]){
        retval = 1;
```

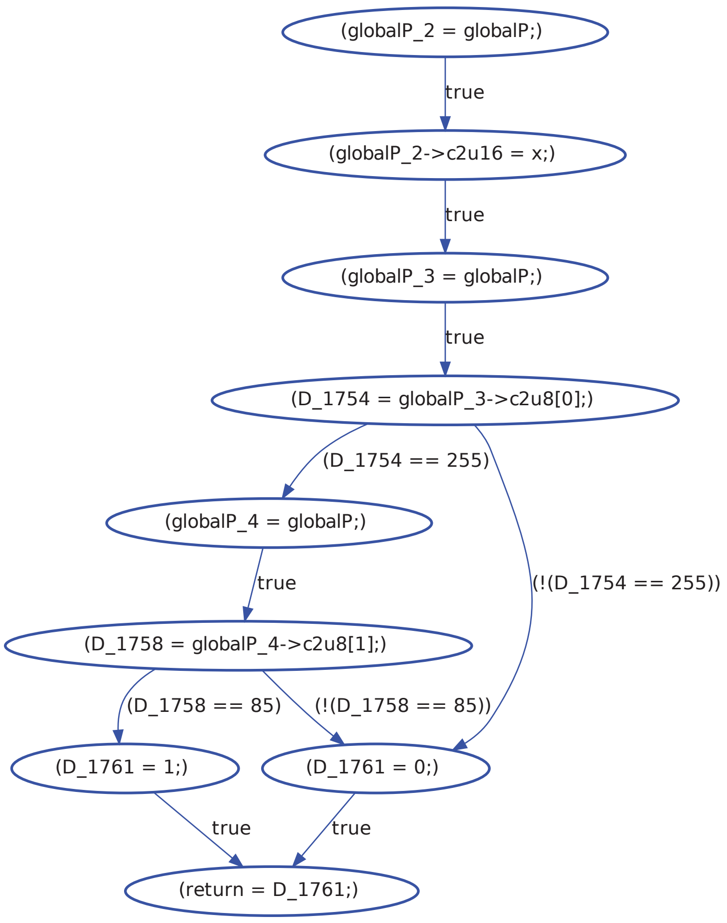


Figure 13: Graphical representation for the example *Pointer Union Access*.

```
    } else {
        retval = 0;
    }
    return retval;
}
```

12.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve 100% branch coverage for the module under test `test()`. The generator needs two test cases to achieve complete branch coverage, therefore the test driver contains two test steps.

First, the module under test is declared. Then the auxiliary array needed to manipulate array parameters in order to fulfill the guard conditions as well as variables for passing of the function parameters are declared. In each test step the assignment of array and other parameters is made according to the calculated values. After the setting is done, the module under test is invoked.

```
/* ===============================
 * Include section
 * ===============================*/
/* ===============================
 * Structures
 * ===============================*/
/* ===============================
 * Global or static C declarations and definitions
 * ===============================*/

extern int test(int* a, unsigned int x, unsigned int y);
@uut int test(int* a, unsigned int x, unsigned int y);

/* ===============================
 * Abstract machine declaration.
 * ===============================*/

@abstract machine unit_test(){

 @INIT:
 @FINIT:
 @PROCESS:

    int __rtt_return;
        int a_PointsTo;
        int* a = &a_PointsTo;
        unsigned int x;
        unsigned int y;

    int a_rtt_array[100];

    @rttBeginTestStep; // ------------------------------------------------
    {
      y = 0;
      x = 1;
      a_rtt_array[0] = -513;
      a = a_rtt_array;
      a_rtt_array[1] = -1;
      a = a_rtt_array;
      @rttCall(__rtt_return = test(a, x, y));
    }
    @rttEndTestStep; // ------------------------------------------------
    @rttBeginTestStep; // ----------------------------------------------------------
    {
       y = 0;
       x = 0;
       a_rtt_array[0] = -1;
       a = a_rtt_array;
       @rttCall(__rtt_return = test(a, x, y));
    }
    @rttEndTestStep; // ------------------------------------------------------------

}
```

12.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of two traces (corresponding to the two test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported. In this example it is 100%.

```
SOLUTION FOR FUNCTION test

TRACE 3
TRACE COMPLETED

TRACE
{ (__rtt_precondition_begin__();),
  (<EMPTYSTATEMENT>;),
  (__rtt_precondition_end__();),
  (retval_0x40bf03a8 = 0;),
  (D_1763 = (x * 4);),
  (D_1764 = (a + D_1763);),
  (D_1765 = (*D_1764);),
  (D_1766 = (y * 4);),
  (D_1767 = (a + D_1766);),
  (D_1768 = (*D_1767);),
  (retval_0x40bf03a8 = 1;),
  (D_1772 = retval_0x40bf03a8;),
  (return = D_1772;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 12
  feasible: 1

CONSTRAINT:
((((x@14 <= 1) &&
 (x@14 == ((unsigned int) x@0))) &&
 (y@14 <= 1) &&
 (y@14 == ((unsigned int) y@0))) &&
 (D_1765@21 > D_1768@21) &&
 (rttTgenIdxExp0@17 == ((8 * (x@0 * 4)) / 32)) &&
 (0 <= (rttTgenIdxExp0@17 * 32)) &&
 ((rttTgenIdxExp0@17 * 32) < 3200) &&
 (D_1765@21 == rttArrayRead(a@ARRAY_ACCESS@0, rttTgenIdxExp0@17)) &&
 (rttTgenIdxExp0@20 == ((8 * (y@0 * 4)) / 32)) &&
 (0 <= (rttTgenIdxExp0@20 * 32)) &&
 ((rttTgenIdxExp0@20 * 32) < 3200) &&
 (D_1768@21 == rttArrayRead(a@ARRAY_ACCESS@0, rttTgenIdxExp0@20)))
SOLUTION:
   x@0 = (ConcreteLattice<unsigned int>, 1)
   y@0 = (ConcreteLattice<unsigned int>, 0)
   a@ARRAY_ACCESS[0]@0 = (ConcreteLattice<signed int>, -513)
   a@ARRAY_ACCESS[1]@0 = (ConcreteLattice<signed int>, -1)
   x@14 = (ConcreteLattice<unsigned int>, 1)
   y@14 = (ConcreteLattice<unsigned int>, 0)
   rttTgenIdxExp0@17 = (ConcreteLattice<unsigned int>, 1)
   rttTgenIdxExp0@20 = (ConcreteLattice<unsigned int>, 0)
   D_1765@21 = (ConcreteLattice<signed int>, -1)
   D_1768@21 = (ConcreteLattice<signed int>, -513)

TRACE 5
TRACE COMPLETED

TRACE
{ (__rtt_precondition_begin__();),
  (<EMPTYSTATEMENT>;),
  (__rtt_precondition_end__();),
  (retval_0x40bf03a8 = 0;),
  (D_1763 = (x * 4);),
  (D_1764 = (a + D_1763);),
  (D_1765 = (*D_1764);),
  (D_1766 = (y * 4);),
  (D_1767 = (a + D_1766);),
  (D_1768 = (*D_1767);),
  (retval_0x40bf03a8 = 0;),
  (D_1772 = retval_0x40bf03a8;),
  (return = D_1772;) }
  traceState:      CONT_OF_ANOTHER_TRACE
  currentStepNr: 12
  feasible: 1

CONSTRAINT:
((((x@14 <= 1) &&
 (x@14 == ((unsigned int) x@0))) &&
 (y@14 <= 1) &&
 (y@14 == ((unsigned int) y@0))) &&
 (D_1765@21 <= D_1768@21) &&
 (rttTgenIdxExp0@17 == ((8 * (x@0 * 4)) / 32)) &&
 (0 <= (rttTgenIdxExp0@17 * 32)) &&
 ((rttTgenIdxExp0@17 * 32) < 3200) &&
 (D_1765@21 == rttArrayRead(a@ARRAY_ACCESS@0, rttTgenIdxExp0@17)) &&
 (rttTgenIdxExp0@20 == ((8 * (y@0 * 4)) / 32)) &&
 (0 <= (rttTgenIdxExp0@20 * 32)) &&
 ((rttTgenIdxExp0@20 * 32) < 3200) &&
 (D_1768@21 == rttArrayRead(a@ARRAY_ACCESS@0, rttTgenIdxExp0@20)))
SOLUTION:
   x@0 = (ConcreteLattice<unsigned int>, 0)
   y@0 = (ConcreteLattice<unsigned int>, 0)
   a@ARRAY_ACCESS[0]@0 = (ConcreteLattice<signed int>, -1)
   x@14 = (ConcreteLattice<unsigned int>, 0)
   y@14 = (ConcreteLattice<unsigned int>, 0)
   rttTgenIdxExp0@17 = (ConcreteLattice<unsigned int>, 0)
   rttTgenIdxExp0@20 = (ConcreteLattice<unsigned int>, 0)
   D_1765@21 = (ConcreteLattice<signed int>, -1)
   D_1768@21 = (ConcreteLattice<signed int>, -1)

Operation test() is covered.

Covered:
```

```
Total transitions:         100%
Transitions with guards: 100%
```

12.4 Graphical Output

Figure 14 shows the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue, which indicates that all of them were successfully covered.

13 Processing Global Arrays

This example corresponds to the example discussed in Section 5.11.2 and demonstrates the technique developed for handling global arrays.

13.1 Analyzed Code

The source code listed below contains the implementation of the module under test `example()`. The module under test `example()` receives a global integer array `aG[]` and an integer parameter `x` as input. It returns *true*, if the element at index `x` is equal to 2 and *false* otherwise.

```
unsigned int aG[10];
int example(unsigned int x)
{
  aG[0] = 1;
  aG[3] = 2;
  if(aG[x] == 2)
    return 1;
  return 0;
}
```

13.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve 100% branch coverage for the module under test `example()`. The generator needs two test cases to achieve complete branch coverage, therefore the test driver contains two test steps. Since no specification of the module under test was given, the test driver contains no assertions. The function `example()` contains no defined or undefined function calls, therefore no stub functions are generated.

First, the module under test is declared. In each test step the assignment of global array elements and other parameters are made according to the calculated values. After the setting is done, the module under test is invoked.

```
/* ===============================
 * Include section
 * ===============================*/
/* ===============================
 * Structures
 * ===============================*/
/* ===============================
 * Global or static C declarations and definitions
 * ===============================*/
```

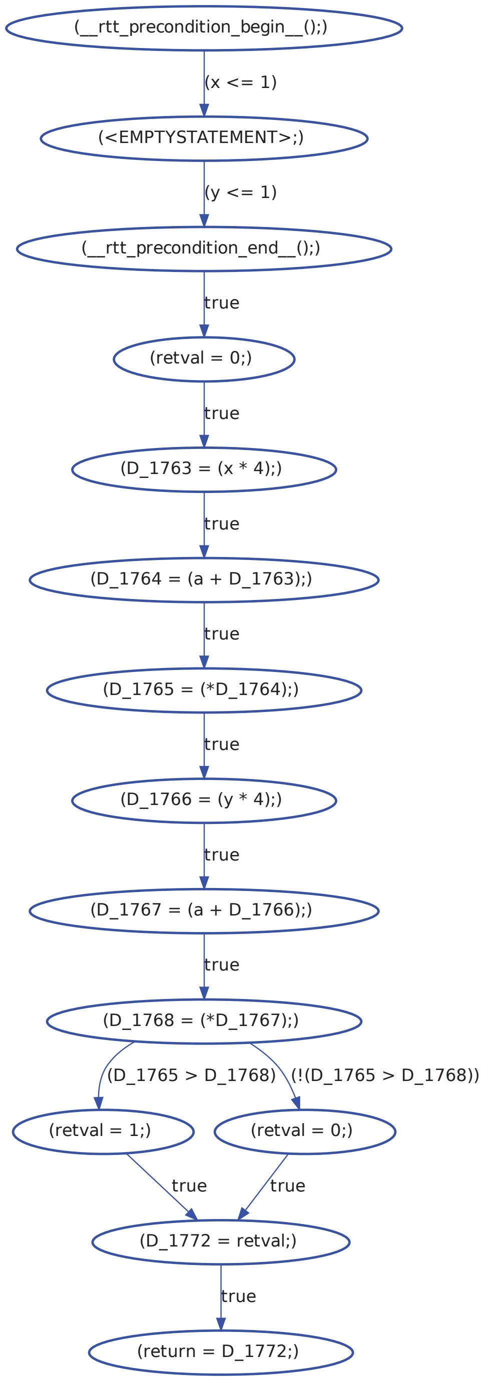


Figure 14: Graphical representation for the example *Input Arrays*.

```
extern int example(unsigned int x);
@uut int example(unsigned int x);

extern unsigned int aG[10];

/* ==============================
 * Abstract machine declaration.
 * ==============================*/

@abstract machine unit_test(){

 @INIT:
{}
 @FINIT:

 @PROCESS:

    int __rtt_return;
    unsigned int x;

    @rttBeginTestStep; // ------------------------------------------------
    {
       aG[1] = 2;
       x = 1;
       @rttCall(__rtt_return = example(x));
    }
    @rttEndTestStep; // ------------------------------------------------

    @rttBeginTestStep; // ------------------------------------------------
    {
       aG[2] = 4294967295;
       x = 2;
       @rttCall(__rtt_return = example(x));
    }
    @rttEndTestStep; // ------------------------------------------------
}
```

13.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of two traces (corresponding to the two test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file the statistics of the achieved coverage is reported. In this example it is 100%.

```
SOLUTION FOR FUNCTION example

TRACE 1
TRACE COMPLETED

TRACE
{ (aG[0] = 1;),
  (aG[3] = 2;),
  (D_1714 = aG[x];),
  (D_1717 = 1;),
  (return = D_1717;) }
  traceState:     CONT_OF_ANOTHER_TRACE
  currentStepNr: 4
  feasible: 1

CONSTRAINT:
(((D_1714@14 == 2) &&
 (((((32 * x@13) < 32) &&
 (0 < ((32 * x@13) + 32)) &&
 (D_1714@14 == 1) || ((((32 * x@13) < 96) &&
 (32 < ((32 * x@13) + 32))) &&
 (D_1714@14 == ((unsigned int) rttArrayRead(aG@ARRAY_ACCESS@0, x@13))))) || ((((32 * x@13)
      < 320) &&
 (128 < ((32 * x@13) + 32))) &&
 (D_1714@14 == ((unsigned int) rttArrayRead(aG@ARRAY_ACCESS@0, x@13))))) || ((((32 * x@13)
      < 128) &&
 (96 < ((32 * x@13) + 32))) &&
 (D_1714@14 == 2)))) &&
 (x@13 == ((unsigned int) x@0)))
SOLUTION:
   aG@ARRAY_ACCESS[1]@0 = (ConcreteLattice<unsigned int>, 2)
   x@0 = (ConcreteLattice<unsigned int>, 1)
   x@13 = (ConcreteLattice<unsigned int>, 1)
   D_1714@14 = (ConcreteLattice<unsigned int>, 2)

TRACE 3
TRACE COMPLETED

TRACE
{ (aG[0] = 1;),
  (aG[3] = 2;),
  (D_1714 = aG[x];),
  (D_1717 = 0;),
  (return = D_1717;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 4
  feasible: 1

CONSTRAINT:
(((D_1714@14 != 2) &&
 (((((32 * x@13) < 32) &&
 (0 < ((32 * x@13) + 32)) &&
 (D_1714@14 == 1) || ((((32 * x@13) < 96) &&
 (32 < ((32 * x@13) + 32))) &&
 (D_1714@14 == ((unsigned int) rttArrayRead(aG@ARRAY_ACCESS@0, x@13))))) || ((((32 * x@13)
      < 320) &&
 (128 < ((32 * x@13) + 32))) &&
 (D_1714@14 == ((unsigned int) rttArrayRead(aG@ARRAY_ACCESS@0, x@13))))) || ((((32 * x@13)
      < 128) &&
 (96 < ((32 * x@13) + 32))) &&
 (D_1714@14 == 2)))) &&
 (x@13 == ((unsigned int) x@0)))
SOLUTION:
   x@0 = (ConcreteLattice<unsigned int>, 2)
   aG@ARRAY_ACCESS[2]@0 = (ConcreteLattice<unsigned int>, 4294967295)
   x@13 = (ConcreteLattice<unsigned int>, 2)
   D_1714@14 = (ConcreteLattice<unsigned int>, 4294967295)

Operation example() is covered.

Covered:
Total transitions:        100%
Transitions with guards: 100%
```

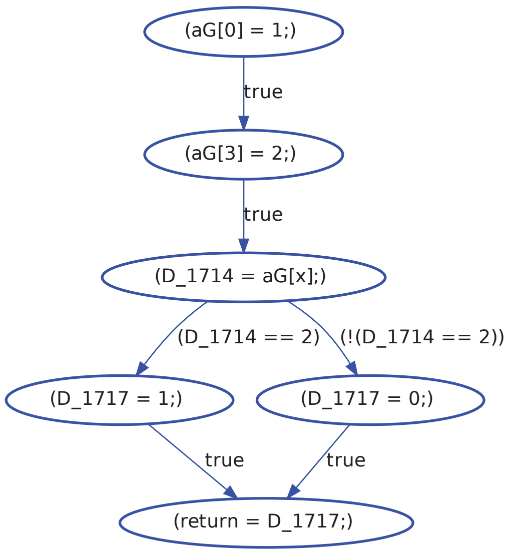


Figure 15: Graphical representation for the example *Global Arrays*.

13.4 Graphical Output

Figure 15 shows the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue, which indicates that all of them were successfully covered.

14 Processing defined Functions

This example corresponds to the example discussed in Section 5.12.1 and demonstrates the technique developed for processing defined function calls.

14.1 Analyzed Code

The source code listed below contains the implementation of two functions: the module under test `test()` and the function `foo()` which is invoked by the module under test. The function `test()` receives two integer parameters `a` and `b` as input. In order to achieve 100% branch coverage of the module under test the called function `foo()` must return once a positive and once a negative value.

```
int foo(unsigned int a, int b)
{
    switch(a){
    case 0:
        return b;
    case 1:
        return -b;
    case 2:
        return 0;
    default:
        return -1;
    }
    return -1;
}

int test(unsigned int a, int b)
{
    if(foo(a, b) > 0){
        return 1;
    }
    return 0;
}
```

14.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve 100% branch coverage for the module under test `test()`. The generator produced two test cases, therefore the test driver contains two test steps. Since no specification of the module under test was given, the test driver contains no assertions. The function `test()` contains only one defined function call, which was symbolically executed as discussed in Section 5.12.1, therefore no stub functions were generated.

First, the module under test is declared. Then variables required to pass the parameters to the module under test as well as an auxiliary variable for the return value of the function are declared. In each test step the assignment of the parameters is made according to the calculated values. After the setting is done the module under test is invoked.

```
/* ===============================
 * Include section
 * ===============================*/


/* ===============================
 * Global or static C declarations and definitions
 * ===============================*/

extern bool test(unsigned int a, int b);
@uut bool test(unsigned int a, int b);


/* ===============================
 * Abstract machine declaration.
 * ===============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:

        bool __rtt_return;
            unsigned int a;
            int b;
```

```
    @rttBeginTestStep; // ------------------------------------------------
    {
       b = 1073741824;
       a = 0;


       @rttCall(__rtt_return = test(a, b));
    }
    @rttEndTestStep; // --------------------------------------------------

    @rttBeginTestStep; // ------------------------------------------------
    {
       a = 2147483648;


       @rttCall(__rtt_return = test(a, b));
    }
    @rttEndTestStep; // --------------------------------------------------

}
```

14.3 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of two traces (corresponding to the two test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is listed.

At the bottom of the solution file statistics for the achieved coverage are reported. In this example the achieved coverage is 100%.

```
SOLUTION FOR FUNCTION test

TRACE 2
TRACE COMPLETED

TRACE
{ (foo_unsigned int_int_parametername_a = a;),
  (foo_unsigned int_int_parametername_b = b;),
  (<EMPTYSTATEMENT>;),
  (D_1725_foo_unsigned int_int = foo_unsigned int_int_parametername_b;),
  (foo_unsigned int_int_return = D_1725_foo_unsigned int_int;),
  (D_1719 = foo_unsigned int_int_return;),
  (retval_0 = (D_1719 > 0);),
  (D_1722 = 1;),
  (return = D_1722;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 8
  feasible: 1

CONSTRAINT:
((((foo_unsigned int_int_parametername_a@12 == 0) &&
 (foo_unsigned int_int_parametername_a@12 == ((unsigned int) a@10))) &&
 (a@10 == ((unsigned int) a@0))) &&
 (retval_0@16 != 0) &&
 (retval_0@16 == ((bool) (D_1719@15 > 0))) &&
 (D_1719@15 == ((int) foo_unsigned int_int_return@14)) &&
 (foo_unsigned int_int_return@14 == ((int) D_1725_foo_unsigned int_int@13)) &&
 (D_1725_foo_unsigned int_int@13 == ((int) foo_unsigned int_int_parametername_b@12)) &&
 (foo_unsigned int_int_parametername_b@12 == ((int) b@11)) &&
 (b@11 == ((int) b@0)))
SOLUTION:
   a@0 = (ConcreteLattice<unsigned int>, 0)
   b@0 = (ConcreteLattice<signed int>, 1073741824)
   a@10 = (ConcreteLattice<unsigned int>, 0)
   b@11 = (ConcreteLattice<signed int>, 1073741824)
   foo_unsigned int_int_parametername_a@12 = (ConcreteLattice<unsigned int>, 0)
   foo_unsigned int_int_parametername_b@12 = (ConcreteLattice<signed int>, 1073741824)
   D_1725_foo_unsigned int_int@13 = (ConcreteLattice<signed int>, 1073741824)
   foo_unsigned int_int_return@14 = (ConcreteLattice<signed int>, 1073741824)
   D_1719@15 = (ConcreteLattice<signed int>, 1073741824)
   retval_0@16 = (ConcreteLattice<bool>, 1)


TRACE 4
TRACE COMPLETED

TRACE
{ (foo_unsigned int_int_parametername_a = a;),
  (foo_unsigned int_int_parametername_b = b;),
  (<EMPTYSTATEMENT>;),
  (D_1725_foo_unsigned int_int = −1;),
  (foo_unsigned int_int_return = D_1725_foo_unsigned int_int;),
  (D_1719 = foo_unsigned int_int_return;),
  (retval_0 = (D_1719 > 0);),
  (D_1722 = 0;),
  (return = D_1722;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 8
  feasible: 1

CONSTRAINT:
((((((foo_unsigned int_int_parametername_a@12 != 0) &&
 (foo_unsigned int_int_parametername_a@12 != 1)) &&
 (foo_unsigned int_int_parametername_a@12 != 2)) &&
 (foo_unsigned int_int_parametername_a@12 == ((unsigned int) a@10))) &&
 (a@10 == ((unsigned int) a@0))) &&
 (retval_0@16 == 0) &&
 (retval_0@16 == ((bool) (D_1719@15 > 0))) &&
 (D_1719@15 == ((int) foo_unsigned int_int_return@14)) &&
 (foo_unsigned int_int_return@14 == ((int) D_1725_foo_unsigned int_int@13)) &&
 (D_1725_foo_unsigned int_int@13 == −1))
SOLUTION:
   a@0 = (ConcreteLattice<unsigned int>, 2147483648)
   a@10 = (ConcreteLattice<unsigned int>, 2147483648)
   foo_unsigned int_int_parametername_a@12 = (ConcreteLattice<unsigned int>, 2147483648)
   D_1725_foo_unsigned int_int@13 = (ConcreteLattice<signed int>, −1)
   foo_unsigned int_int_return@14 = (ConcreteLattice<signed int>, −1)
   D_1719@15 = (ConcreteLattice<signed int>, −1)
   retval_0@16 = (ConcreteLattice<bool>, 0)


Operation test() is covered.

Covered:
Total transitions:        100%
Transitions with guards: 100%
```

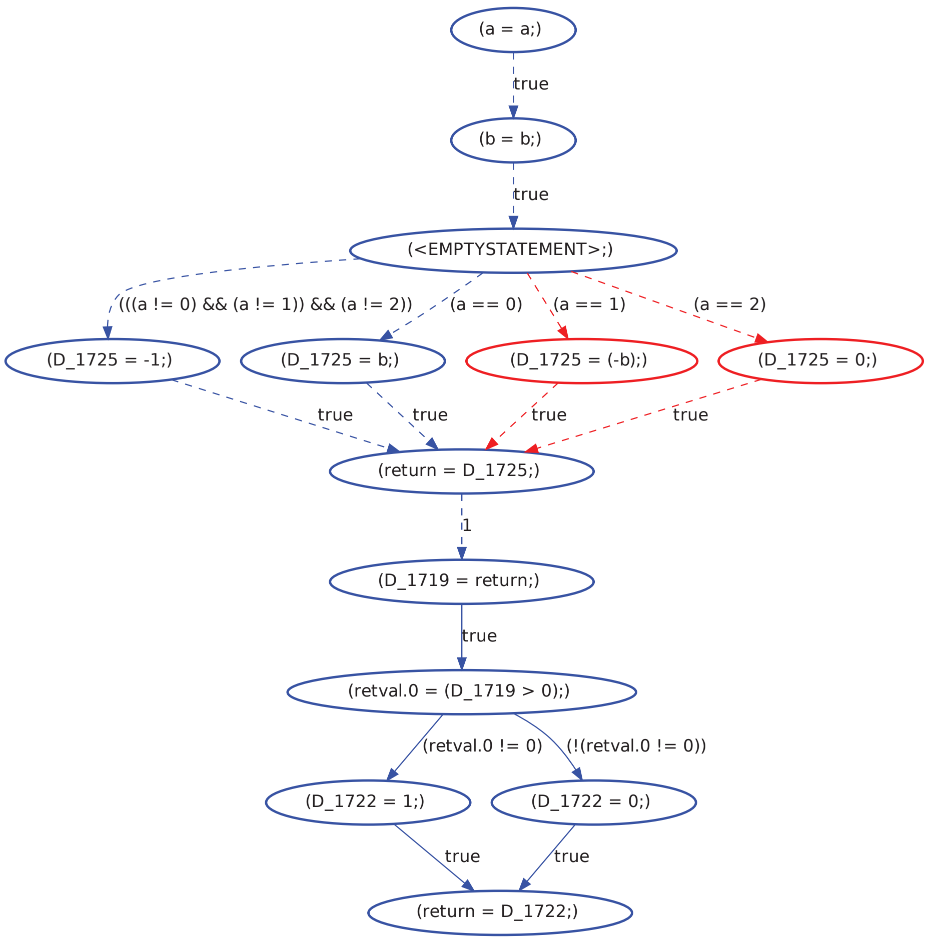


Figure 16: Graphical representation for the example *Defined Functions*.

14.4 Graphical Output

Figure 16 shows the graphical representation of the CFG corresponding to the module under test. The edges corresponding to the called function `foo()` are drawn dashed. All covered nodes and edges of this CFG are drawn blue, all uncovered ones are drawn red. Although it was reported, that the achieved coverage is 100% there are some red nodes and edges in the CFG. However, this is no contradiction since all uncovered edges and nodes belong to the called function `foo()` and not to the module under test `test()`. Moreover, the nodes of the called function are left uncovered deliberately, since their coverage does not contribute to the coverage of the module under test.

15 Processing undefined Functions

This example corresponds to the example discussed in Section 5.12.2 and demonstrates how CTGEN processes undefined functions and how it generates mock objects with the same signature to replace those functions.

15.1 Analyzed Code

The source code listed below contains the implementation of the module under test `test()`. The function receives two integer parameters `p1` and `p2` as input. The module under test contains two calls to the external function `func_ext()`. To reach the line with an "error", `func_ext()` must return a value that is greater than the value of the parameter `p2` on the first call. Furthermore, on the second call it must return a value that is equal to the value of the parameter `p1`.

```
#include "ctgen_annotation.h"
extern int func_ext(int a);
int globalVar;

void test(int p1, int p2)
{
    __rtt_modifies(globalVar);
    int error = 0;
    globalVar = -p2;
    if(func_ext(p1) > p2){

        if(func_ext(p2) == p1 && globalVar == p2){
            error = 1;
            return;
        }
    }
    return;
}
```

15.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve 100% branch coverage for the module under test `test()`. The generator produced four test cases, therefore the test driver contains four test steps. The function `test()` contains undefined function calls, which all refer to the same external function, therefore one stub function is generated in order to replace it. The source code of the stub function is listed in Section 15.3.

First, the module under test is declared. Next the auxiliary global variables required for stub manipulation and the variables required to pass the parameter values to the module under test are declared. In each test step the assignment of the parameters as well as of global stub variables is made according to the calculated values and the running number of the test case. After the settings are done, the module under test is invoked.

```
/* ==============================
 * Include section
 * ==============================*/


/* ==============================
 * Global or static C declarations and definitions
 * ==============================*/

extern void test(int p1, int p2);
@uut void test(int p1, int p2);


 /* external stubs vars for stub func_ext */
extern unsigned int func_ext_STUB_testCaseNr;
extern unsigned int func_ext_STUB_retID;
extern int func_ext_STUB_retVal[2];
extern int a_func_ext_PARAM_VALUE;

/* ==============================
 * Abstract machine declaration.
 * ==============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:

        int p1;
        int p2;


    @rttBeginTestStep; // ------------------------------------------------
    {
        /***** STUB func_ext *****/
        func_ext_STUB_testCaseNr = 0;
        func_ext_STUB_retID = 0;

        /* set values for return */
       func_ext_STUB_retVal[0] = 2147483647;
       func_ext_STUB_retVal[1] = 0;

        /* values for globals are set in stub */
        /* set values for output parameters */
       p2 = -1;
       p1 = 0;


       @rttCall(test(p1, p2));
    }
    @rttEndTestStep; // --------------------------------------------------

    @rttBeginTestStep; // ------------------------------------------------
    {
        /***** STUB func_ext *****/
        func_ext_STUB_testCaseNr = 1;
        func_ext_STUB_retID = 0;

        /* set values for return */
       func_ext_STUB_retVal[0] = -1;

        /* values for globals are set in stub */
        /* set values for output parameters */
       p2 = -1;
```

```
        @rttCall(test(p1, p2));
    }
    @rttEndTestStep; // ------------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------
    {
        /***** STUB func_ext *****/
        func_ext_STUB_testCaseNr = 2;
        func_ext_STUB_retID = 0;

        /* set values for return */
       func_ext_STUB_retVal[0] = 2147483647;
       func_ext_STUB_retVal[1] = -2;

        /* values for globals are set in stub */
        /* set values for output parameters */
       p2 = -1;
       p1 = -1;


        @rttCall(test(p1, p2));
    }
    @rttEndTestStep; // ------------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------
    {
        /***** STUB func_ext *****/
        func_ext_STUB_testCaseNr = 3;
        func_ext_STUB_retID = 0;

        /* set values for return */
       func_ext_STUB_retVal[0] = 2147483646;
       func_ext_STUB_retVal[1] = 0;

        /* values for globals are set in stub */
        /* set values for output parameters */
       p2 = -2;
       p1 = 0;


        @rttCall(test(p1, p2));
    }
    @rttEndTestStep; // ------------------------------------------------

}
```

15.3 Generated Stub

This section demonstrates the stub function generated by the test generator to replace the external function called by the module under test. The section `@GLOBAL` of the stub contains the declarations of the variables required to set the return values as well as to modify the global variables according to the values calculated by the generator. The variable `func_ext_STUB_testCaseNr` keeps track of the executed test steps, the variable `func_ext_STUB_retID` keeps track of the number of executions of the stub within the current test step. The array `func_ext_STUB_retVal[]` holds the data for the return values of the stub. This data is set by the test driver. The variable `a_func_ext_PARAM_VALUE` holds the value of the passed parameter. Furthermore the global variable `globalVar` modified by the stub is declared.

```
int func_ext(int a){
        @GLOBAL:
                unsigned int func_ext_STUB_testCaseNr;
                unsigned int func_ext_STUB_retID;
                int func_ext_STUB_retVal[2];
                int a_func_ext_PARAM_VALUE;
                extern int globalVar;
        @BODY:
                func_ext_RETURN = func_ext_STUB_retVal[func_ext_STUB_retID%2];
                a_func_ext_PARAM_VALUE = (int)a;
                if(func_ext_STUB_testCaseNr == 0){
                        if(func_ext_STUB_retID == 1){
                                globalVar = -1;
                        }
                }
                if(func_ext_STUB_testCaseNr == 3){
                        if(func_ext_STUB_retID == 1){
                                globalVar = -1;
                        }
                }

                func_ext_STUB_retID++;
};
```

15.4 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of four traces (corresponding to the four test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is shown.

At the bottom of the solution file statistics for the achieved coverage is reported. In this example the achieved coverage is 100%.

```
SOLUTION FOR FUNCTION test

TRACE 4
TRACE COMPLETED

TRACE
{ (__rtt_modifies__((&"globalVar"));),
  (error_0x40bf023c = 0;),
  (globalVar_0 = (-p2);),
  (globalVar = globalVar_0;),
  (D_1768 = func_ext(p1);),
  (D_1753 = D_1768;),
  (retval_1 = (D_1753 > p2);),
  (D_1769 = func_ext(p2);),
  (D_1760 = D_1769;),
  (globalVar_4 = globalVar;),
  (iftmp_3 = 1;),
  (retval_2 = iftmp_3;),
  (error_0x40bf023c = 1;),
  (return;) }
  traceState:     CONT_OF_ANOTHER_TRACE
  currentStepNr: 13
```

```
   feasible: 1

CONSTRAINT:
((((((((retval_1@21 != 0) &&
 (retval_1@21 == ((bool) (D_1753@20 > p2@20)))) &&
 (D_1753@20 == ((int) D_1768@19))) &&
 (p2@20 == ((int) p2@0))) &&
 (D_1768@19 == ((int) func_ext@RETURN@0))) &&
 (D_1760@24 == p1@24) &&
 (D_1760@24 == ((int) D_1769@23)) &&
 (p1@24 == ((int) p1@0)) &&
 (D_1769@23 == ((int) func_ext@RETURN@1))) &&
 (globalVar_4@25 == p2@25) &&
 (globalVar_4@25 == ((int) globalVar@24)) &&
 (p2@25 == ((int) p2@0)) &&
 (globalVar@24 == ((int) globalVar@func_ext@1))) &&
 (retval_2@27 != 0) &&
 (retval_2@27 == ((bool) iftmp_3@26)) &&
 (iftmp_3@26 == 1))
SOLUTION:
    func_ext@RETURN@0 = (ConcreteLattice<signed int>, 2147483647)
    p2@0 = (ConcreteLattice<signed int>, −1)
    p1@0 = (ConcreteLattice<signed int>, 0)
    func_ext@RETURN@1 = (ConcreteLattice<signed int>, 0)
    globalVar@func_ext@1 = (ConcreteLattice<signed int>, −1)
    D_1768@19 = (ConcreteLattice<signed int>, 2147483647)
    p2@20 = (ConcreteLattice<signed int>, −1)
    D_1753@20 = (ConcreteLattice<signed int>, 2147483647)
    retval_1@21 = (ConcreteLattice<bool>, 1)
    D_1769@23 = (ConcreteLattice<signed int>, 0)
    p1@24 = (ConcreteLattice<signed int>, 0)
    D_1760@24 = (ConcreteLattice<signed int>, 0)
    globalVar@24 = (ConcreteLattice<signed int>, −1)
    p2@25 = (ConcreteLattice<signed int>, −1)
    globalVar_4@25 = (ConcreteLattice<signed int>, −1)
    iftmp_3@26 = (ConcreteLattice<bool>, 1)
    retval_2@27 = (ConcreteLattice<bool>, 1)


TRACE 6
TRACE COMPLETED

TRACE
{ (__rtt_modifies__((&"globalVar"));),
  (error_0x40bf023c = 0;),
  (globalVar_0 = (−p2);),
  (globalVar = globalVar_0;),
  (D_1768 = func_ext(p1);),
  (D_1753 = D_1768;),
  (retval_1 = (D_1753 > p2);),
  (<EMPTYSTATEMENT>;),
  (return;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 8
  feasible: 1

CONSTRAINT:
(((((retval_1@21 == 0) &&
 (retval_1@21 == ((bool) (D_1753@20 > p2@20)))) &&
 (D_1753@20 == ((int) D_1768@19))) &&
 (p2@20 == ((int) p2@0))) &&
 (D_1768@19 == ((int) func_ext@RETURN@0)))
```

```
SOLUTION:
    func_ext@RETURN@0 = (ConcreteLattice<signed int>, −1)
    p2@0 = (ConcreteLattice<signed int>, −1)
    D_1768@19 = (ConcreteLattice<signed int>, −1)
    p2@20 = (ConcreteLattice<signed int>, −1)
    D_1753@20 = (ConcreteLattice<signed int>, −1)
    retval_1@21 = (ConcreteLattice<bool>, 0)


TRACE 9
TRACE COMPLETED

TRACE
{ (__rtt_modifies__((&"globalVar"));),
  (error_0x40bf023c = 0;),
  (globalVar_0 = (−p2);),
  (globalVar = globalVar_0;),
  (D_1768 = func_ext(p1);),
  (D_1753 = D_1768;),
  (retval_1 = (D_1753 > p2);),
  (D_1769 = func_ext(p2);),
  (D_1760 = D_1769;),
  (iftmp_3 = 0;),
  (retval_2 = iftmp_3;),
  (<EMPTYSTATEMENT>;),
  (return;) }
  traceState:     CONT_OF_ANOTHER_TRACE
  currentStepNr: 12
  feasible: 1

CONSTRAINT:
(((((((retval_1@21 != 0) &&
 (retval_1@21 == ((bool) (D_1753@20 > p2@20)))) &&
 (D_1753@20 == ((int) D_1768@19))) &&
 (p2@20 == ((int) p2@0))) &&
 (D_1768@19 == ((int) func_ext@RETURN@0))) &&
 (D_1760@24 != p1@24) &&
 (D_1760@24 == ((int) D_1769@23)) &&
 (p1@24 == ((int) p1@0)) &&
 (D_1769@23 == ((int) func_ext@RETURN@1))) &&
 (retval_2@26 == 0) &&
 (retval_2@26 == ((bool) iftmp_3@25)) &&
 (iftmp_3@25 == 0))
SOLUTION:
    func_ext@RETURN@0 = (ConcreteLattice<signed int>, 2147483647)
    p2@0 = (ConcreteLattice<signed int>, −1)
    p1@0 = (ConcreteLattice<signed int>, −1)
    func_ext@RETURN@1 = (ConcreteLattice<signed int>, −2)
    D_1768@19 = (ConcreteLattice<signed int>, 2147483647)
    p2@20 = (ConcreteLattice<signed int>, −1)
    D_1753@20 = (ConcreteLattice<signed int>, 2147483647)
    retval_1@21 = (ConcreteLattice<bool>, 1)
    D_1769@23 = (ConcreteLattice<signed int>, −2)
    p1@24 = (ConcreteLattice<signed int>, −1)
    D_1760@24 = (ConcreteLattice<signed int>, −2)
    iftmp_3@25 = (ConcreteLattice<bool>, 0)
    retval_2@26 = (ConcreteLattice<bool>, 0)


TRACE 10
TRACE COMPLETED
```

```
TRACE
{ (__rtt_modifies__((&"globalVar"));),
  (error_0x40bf023c = 0;),
  (globalVar_0 = (−p2);),
  (globalVar = globalVar_0;),
  (D_1768 = func_ext(p1);),
  (D_1753 = D_1768;),
  (retval_1 = (D_1753 > p2);),
  (D_1769 = func_ext(p2);),
  (D_1760 = D_1769;),
  (globalVar_4 = globalVar;),
  (iftmp_3 = 0;) }
  traceState:    CONT_AFTER_SOLVING
  currentStepNr: 10
  feasible: 1

CONSTRAINT:
(((((((retval_1@21 != 0) &&
 (retval_1@21 == ((bool) (D_1753@20 > p2@20)))) &&
 (D_1753@20 == ((int) D_1768@19))) &&
 (p2@20 == ((int) p2@0))) &&
 (D_1768@19 == ((int) func_ext@RETURN@0))) &&
 (D_1760@24 == p1@24) &&
 (D_1760@24 == ((int) D_1769@23)) &&
 (p1@24 == ((int) p1@0)) &&
 (D_1769@23 == ((int) func_ext@RETURN@1))) &&
 (globalVar_4@25 != p2@25) &&
 (globalVar_4@25 == ((int) globalVar@24)) &&
 (p2@25 == ((int) p2@0)) &&
 (globalVar@24 == ((int) globalVar@func_ext@1)))
SOLUTION:
   func_ext@RETURN@0 = (ConcreteLattice<signed int>, 2147483646)
   p2@0 = (ConcreteLattice<signed int>, −2)
   p1@0 = (ConcreteLattice<signed int>, 0)
   func_ext@RETURN@1 = (ConcreteLattice<signed int>, 0)
   globalVar@func_ext@1 = (ConcreteLattice<signed int>, −1)
   D_1768@19 = (ConcreteLattice<signed int>, 2147483646)
   p2@20 = (ConcreteLattice<signed int>, −2)
   D_1753@20 = (ConcreteLattice<signed int>, 2147483646)
   retval_1@21 = (ConcreteLattice<bool>, 1)
   D_1769@23 = (ConcreteLattice<signed int>, 0)
   p1@24 = (ConcreteLattice<signed int>, 0)
   D_1760@24 = (ConcreteLattice<signed int>, 0)
   globalVar@24 = (ConcreteLattice<signed int>, −1)
   p2@25 = (ConcreteLattice<signed int>, −2)
   globalVar_4@25 = (ConcreteLattice<signed int>, −1)


Operation test() is covered.

Covered:
Total transitions:        100%
Transitions with guards: 100%
```

15.5 Graphical Output

Figure 17 shows the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue, which indicates that all of them were successfully covered.

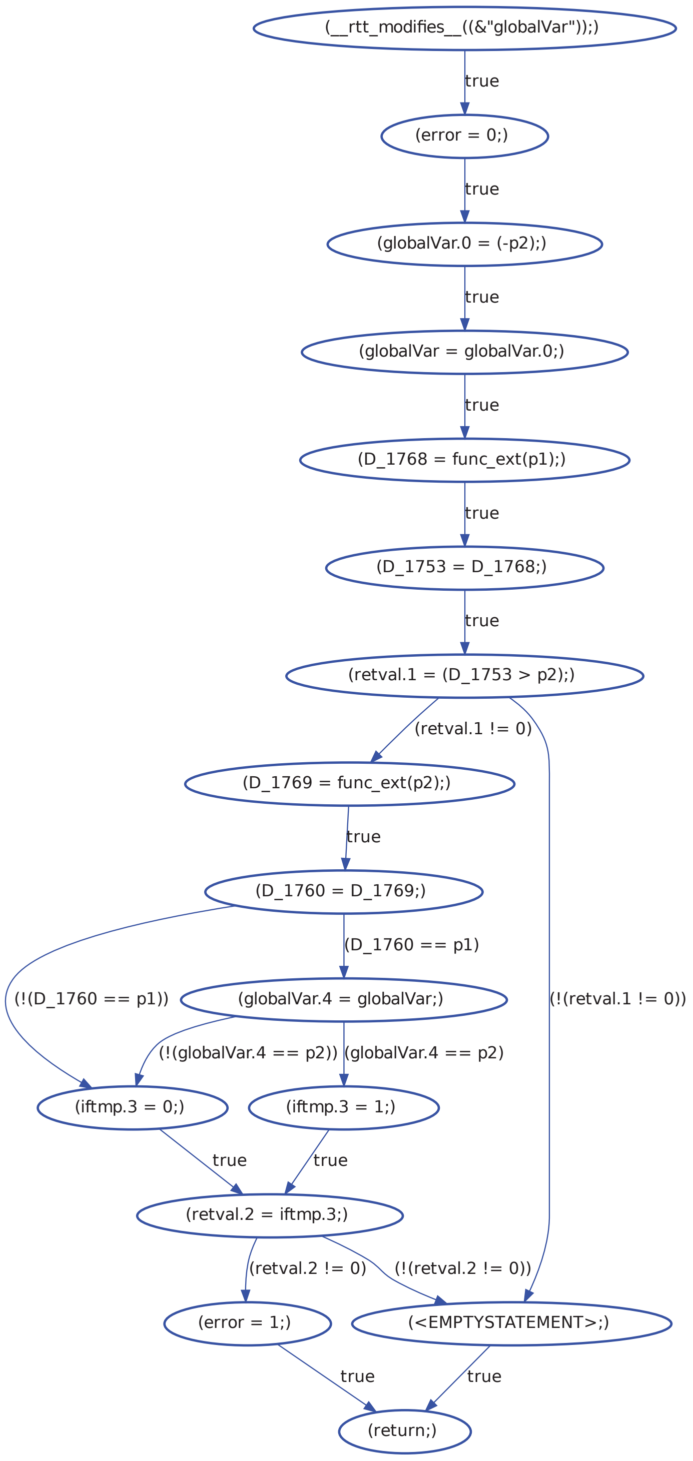


Figure 17: Graphical representation for the example *Undefined Functions*.

16 Processing undefined Functions with Stub Specification

This example corresponds to the example discussed in Section 5.12.3 and demonstrates how it is possible to specify an undefined function by means of the CTGEN annotation language.

16.1 Analyzed Code

The analyzed module under test is the same as in the previous example demonstrated in Appendix 15. The difference is, that the undefined function `func_ext()` received a specification. For this purpose a dummy function with the same signature was implemented. This function contains only the specification of the values of the parameter and of the return value as well as of the value of the global variable `globalVar`. They are restricted to the permitted value range by means of the pre- and postconditions.

```
#include "ctgen_annotation.h"
int globalVar;

int func_ext(int a)
{
    __rtt_extern();
    __rtt_precondition(a > 0 && -20 < globalVar && globalVar < 20);
    __rtt_postcondition(__rtt_return < 20 && globalVar > 17);

    return 0;
}

void test(int p1, int p2)
{
    __rtt_modifies(globalVar) ;
    int error = 0;
    globalVar = -p2;
    if(func_ext(p1) > p2){

        if(func_ext(p2) == p1 && globalVar == p2){
            error = 1;
            return;
        }
    }
    return;
}
```

16.2 Generated Test Driver

This section demonstrates the test driver generated by the test generator in order to achieve 100% branch coverage for the module under test `test()`. The generator produced four test cases, therefore the test driver contains four test steps. The function `test()` contains undefined function calls, which all refer to the same external function, therefore one stub function is generated in order to replace it. The source code of the stub function is shown in Section 16.3.

First, the module under test is declared. Next the auxiliary global variables required for stub manipulation and the variables required to pass the parameters to the module under test are declared. In each test step the assignment of the parameters as well as of global stub variables is made according to the calculated values and the running number of the test case. After the settings are done, the module under test is invoked.

Return values are all in the specified range, and the parameter of the module under test are set in such a way, that the precondition on the global variable `globalVar` holds.

```
/* ==============================
 * Include section
 * ==============================*/


/* ==============================
 * Global or static C declarations and definitions
 * ==============================*/

extern void test(int p1, int p2);
@uut void test(int p1, int p2);


 /* external stubs vars for stub func_ext */
extern unsigned int func_ext_STUB_testCaseNr;
extern unsigned int func_ext_STUB_retID;
extern int func_ext_STUB_retVal[2];
extern int a_func_ext_PARAM_VALUE;

/* ==============================
 * Abstract machine declaration.
 * ==============================*/

@abstract machine unit_test(){

 @INIT:
{
}
 @FINIT:

 @PROCESS:

        int p1;
        int p2;


    @rttBeginTestStep; // ----------------------------------------------------------------
    {
        /***** STUB func_ext *****/
        func_ext_STUB_testCaseNr = 0;
        func_ext_STUB_retID = 0;

        /* set values for return */
       func_ext_STUB_retVal[0] = -1;

        /* values for globals are set in stub */
        /* set values for output parameters */
       p2 = -1;
       p1 = 1073741824;


       @rttCall(test(p1, p2));
    }
    @rttEndTestStep; // ----------------------------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------------------------
    {
        /***** STUB func_ext *****/
        func_ext_STUB_testCaseNr = 1;
        func_ext_STUB_retID = 0;

        /* set values for return */
       func_ext_STUB_retVal[0] = 19;
       func_ext_STUB_retVal[1] = 16;

        /* values for globals are set in stub */
        /* set values for output parameters */
       p2 = 18;
       p1 = 16;


       @rttCall(test(p1, p2));
    }
    @rttEndTestStep; // ------------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------
    {
        /***** STUB func_ext *****/
        func_ext_STUB_testCaseNr = 2;
        func_ext_STUB_retID = 0;

        /* set values for return */
       func_ext_STUB_retVal[0] = 19;
       func_ext_STUB_retVal[1] = -1072693245;

        /* values for globals are set in stub */
        /* set values for output parameters */
       p2 = 15;
       p1 = 1073741824;


       @rttCall(test(p1, p2));
    }
    @rttEndTestStep; // ------------------------------------------------

    @rttBeginTestStep; // ----------------------------------------------
    {
        /***** STUB func_ext *****/
        func_ext_STUB_testCaseNr = 3;
        func_ext_STUB_retID = 0;

        /* set values for return */
       func_ext_STUB_retVal[0] = 19;
       func_ext_STUB_retVal[1] = 16;

        /* values for globals are set in stub */
        /* set values for output parameters */
       p2 = 3;
       p1 = 16;


       @rttCall(test(p1, p2));
    }
    @rttEndTestStep; // ------------------------------------------------

}
```

16.3 Generated Stub

This section demonstrates the stub function generated by the test generator to replace the external function called by the module under test. The section @GLOBAL of the stub contains declarations of the variables required to set the return values as well as to modify the global variables according to the values calculated by the generator. The variable func_ext_STUB_testCaseNr keeps track of the executed test steps and the variable func_ext_STUB_retID of the number of executions of the stub within the current test step. The array func_ext_STUB_retVal[] holds the data for the return values of the stub. This data is set by the test driver. The variable a_func_ext_PARAM_VALUE holds the value of the passed parameter. Furthermore the global variable globalVar modified by the stub is declared.

The global variable globalVar is set in such a way, that the postcondition holds.

```
/**
 * This file contains the local stub function definitions of the form
 *
* [<type-kind>] <return-type> <function-name> [<format>]
 *                              ( <parameter> (, <parameter>)* );
 */

int func_ext(int a){
        @GLOBAL:
                unsigned int func_ext_STUB_testCaseNr;
                unsigned int func_ext_STUB_retID;
                int func_ext_STUB_retVal[2];
                int a_func_ext_PARAM_VALUE;
                extern int globalVar;
        @BODY:
                func_ext_RETURN = func_ext_STUB_retVal[func_ext_STUB_retID%2];
                a_func_ext_PARAM_VALUE = (int)a;
                if(func_ext_STUB_testCaseNr == 0){
                        if(func_ext_STUB_retID == 0){
                                globalVar = 1073741840;
                        }
                }
                if(func_ext_STUB_testCaseNr == 1){
                        if(func_ext_STUB_retID == 0){
                                globalVar = 19;
                        }
                        if(func_ext_STUB_retID == 1){
                                globalVar = 18;
                        }
                }
                if(func_ext_STUB_testCaseNr == 2){
                        if(func_ext_STUB_retID == 0){
                                globalVar = 19;
                        }
                        if(func_ext_STUB_retID == 1){
                                globalVar = 1073741840;
                        }
                }
                if(func_ext_STUB_testCaseNr == 3){
                        if(func_ext_STUB_retID == 0){
                                globalVar = 19;
                        }
                        if(func_ext_STUB_retID == 1){
                                globalVar = 1073741825;
                        }
```

```
                    }

                    func_ext_STUB_retID++;
};
```

16.4 Solution File

This section demonstrates the solution file that results from the generation process. This file contains the description of four traces (corresponding to the four test steps in the test driver). Each trace is first specified by the list of its statements, then the path constraint calculated by the generator for this trace is given and finally the solution computed by the solver for this path constraint is shown.

At the bottom of the solution file statistics for the achieved coverage are reported. In this example the achieved coverage is 100%.

```
SOLUTION FOR FUNCTION test

TRACE 1
TRACE COMPLETED

TRACE
{ (error_0x40bf02d8 = 0;),
  (globalVar_0 = (-p2);),
  (globalVar = globalVar_0;),
  (func_ext_int_parametername_a = p1;),
  (__rtt_precondition_begin__();),
  (globalVar_5_func_ext_int = globalVar;),
  (globalVar_6_func_ext_int = globalVar;),
  (__rtt_precondition_end__();),
  (D_1755 = func_ext(p1);),
  (__rtt_postcondition_begin__();),
  (globalVar_7_func_ext_int = globalVar@func_ext;),
  (__rtt_postcondition_end__();),
  (D_1755 = func_ext@RETURN;),
  (retval_1 = (D_1755 > p2);),
  (<EMPTYSTATEMENT>;),
  (return;) }
  traceState:    CONT_OF_ANOTHER_TRACE
  currentStepNr: 15
  feasible: 1

CONSTRAINT:
(((((((func_ext_int_parametername_a@24 > 0) &&
 (func_ext_int_parametername_a@24 == ((int) p1@23))) &&
 (p1@23 == ((int) p1@0))) &&
 (globalVar_5_func_ext_int@25 >= -19) &&
 (globalVar_5_func_ext_int@25 == ((int) globalVar@24)) &&
 (globalVar@24 == ((int) globalVar_0@22)) &&
 (globalVar_0@22 == ((int) (-p2@21))) &&
 (p2@21 == ((int) p2@0))) &&
 (globalVar_6_func_ext_int@26 <= 19) &&
 (globalVar_6_func_ext_int@26 == ((int) globalVar@25)) &&
 (globalVar@25 == globalVar_0@22)) &&
 (func_ext@RETURN@0 <= 19)) &&
 (globalVar_7_func_ext_int@29 > 17) &&
 (globalVar_7_func_ext_int@29 == ((int) globalVar@func_ext@0))) &&
 (retval_1@31 == 0) &&
 (retval_1@31 == ((bool) (D_1755@30 > p2@30))) &&
 (D_1755@30 == func_ext@RETURN@0) &&
 (p2@30 == ((int) p2@0)))
SOLUTION:
    p1@0 = (ConcreteLattice<signed int>, 1073741824)
    p2@0 = (ConcreteLattice<signed int>, -1)
    func_ext@RETURN@0 = (ConcreteLattice<signed int>, -1)
    globalVar@func_ext@0 = (ConcreteLattice<signed int>, 1073741840)
    p2@21 = (ConcreteLattice<signed int>, -1)
    globalVar_0@22 = (ConcreteLattice<signed int>, 1)
    p1@23 = (ConcreteLattice<signed int>, 1073741824)
    func_ext_int_parametername_a@24 = (ConcreteLattice<signed int>, 1073741824)
    globalVar@24 = (ConcreteLattice<signed int>, 1)
    globalVar@25 = (ConcreteLattice<signed int>, 1)
    globalVar_5_func_ext_int@25 = (ConcreteLattice<signed int>, 1)
    globalVar_6_func_ext_int@26 = (ConcreteLattice<signed int>, 1)
    globalVar_7_func_ext_int@29 = (ConcreteLattice<signed int>, 1073741840)
    p2@30 = (ConcreteLattice<signed int>, -1)
    D_1755@30 = (ConcreteLattice<signed int>, -1)
    retval_1@31 = (ConcreteLattice<bool>, 0)


TRACE 11
TRACE COMPLETED

TRACE
{ (error_0x40bf02d8 = 0;),
   (globalVar_0 = (-p2);),
   (globalVar = globalVar_0;),
   (func_ext_int_parametername_a = p1;),
   (__rtt_precondition_begin__();),
   (globalVar_5_func_ext_int = globalVar;),
   (globalVar_6_func_ext_int = globalVar;),
   (__rtt_precondition_end__();),
   (D_1755 = func_ext(p1);),
   (__rtt_postcondition_begin__();),
   (globalVar_7_func_ext_int = globalVar@func_ext;),
   (__rtt_postcondition_end__();),
   (D_1755 = func_ext@RETURN;),
   (retval_1 = (D_1755 > p2);),
   (func_ext_int_parametername_a = p2;),
   (__rtt_precondition_begin__();),
   (globalVar_5_func_ext_int = globalVar;),
   (globalVar_6_func_ext_int = globalVar;),
   (__rtt_precondition_end__();),
   (D_1762 = func_ext(p2);),
   (__rtt_postcondition_begin__();),
   (globalVar_7_func_ext_int = globalVar@func_ext;),
   (__rtt_postcondition_end__();),
   (D_1762 = func_ext@RETURN;),
   (globalVar_4 = globalVar;),
   (iftmp_3 = 1;),
   (retval_2 = iftmp_3;),
   (error_0x40bf02d8 = 1;),
   (return;) }
   traceState:      CONT_OF_ANOTHER_TRACE
   currentStepNr: 28
   feasible: 1

CONSTRAINT:
(((((((((((((((func_ext_int_parametername_a@24 > 0) &&
 (func_ext_int_parametername_a@24 == ((int) p1@23))) &&
 (p1@23 == ((int) p1@0))) &&
 (globalVar_5_func_ext_int@25 >= -19) &&
 (globalVar_5_func_ext_int@25 == ((int) globalVar@24)) &&
 (globalVar@24 == ((int) globalVar_0@22)) &&
 (globalVar_0@22 == ((int) (-p2@21))) &&
 (p2@21 == ((int) p2@0))) &&
 (globalVar_6_func_ext_int@26 <= 19) &&
 (globalVar_6_func_ext_int@26 == ((int) globalVar@25)) &&
 (globalVar@25 == globalVar_0@22)) &&
 (func_ext@RETURN@0 <= 19)) &&
 (globalVar_7_func_ext_int@29 > 17) &&
 (globalVar_7_func_ext_int@29 == ((int) globalVar@func_ext@0))) &&
 (retval_1@31 != 0) &&
 (retval_1@31 == ((bool) (D_1755@30 > p2@30))) &&
 (D_1755@30 == func_ext@RETURN@0) &&
 (p2@30 == ((int) p2@0))) &&
 (func_ext_int_parametername_a@32 > 0) &&
 (func_ext_int_parametername_a@32 == ((int) p2@31)) &&
 (p2@31 == ((int) p2@0))) &&
 (globalVar_5_func_ext_int@33 >= -19) &&
 (globalVar_5_func_ext_int@33 == ((int) globalVar@32)) &&
 (globalVar@32 == globalVar@func_ext@0)) &&
 (globalVar_6_func_ext_int@34 <= 19) &&
 (globalVar_6_func_ext_int@34 == ((int) globalVar@33)) &&
 (globalVar@33 == globalVar@func_ext@0)) &&
 (func_ext@RETURN@1 <= 19)) &&
 (globalVar_7_func_ext_int@37 > 17) &&
 (globalVar_7_func_ext_int@37 == ((int) globalVar@func_ext@1))) &&
 (D_1762@38 == p1@38) &&
 (D_1762@38 == func_ext@RETURN@1) &&
 (p1@38 == ((int) p1@0))) &&
 (globalVar_4@39 == p2@39) &&
 (globalVar_4@39 == ((int) globalVar@38)) &&
 (p2@39 == ((int) p2@0)) &&
 (globalVar@38 == globalVar@func_ext@1)) &&
 (retval_2@41 != 0) &&
 (retval_2@41 == ((bool) iftmp_3@40)) &&
 (iftmp_3@40 == 1))
SOLUTION:
   p1@0 = (ConcreteLattice<signed int>, 16)
   p2@0 = (ConcreteLattice<signed int>, 18)
   func_ext@RETURN@0 = (ConcreteLattice<signed int>, 19)
   globalVar@func_ext@0 = (ConcreteLattice<signed int>, 19)
   func_ext@RETURN@1 = (ConcreteLattice<signed int>, 16)
   globalVar@func_ext@1 = (ConcreteLattice<signed int>, 18)
   p2@21 = (ConcreteLattice<signed int>, 18)
   globalVar_0@22 = (ConcreteLattice<signed int>, -18)
   p1@23 = (ConcreteLattice<signed int>, 16)
   func_ext_int_parametername_a@24 = (ConcreteLattice<signed int>, 16)
   globalVar@24 = (ConcreteLattice<signed int>, -18)
   globalVar@25 = (ConcreteLattice<signed int>, -18)
   globalVar_5_func_ext_int@25 = (ConcreteLattice<signed int>, -18)
   globalVar_6_func_ext_int@26 = (ConcreteLattice<signed int>, -18)
   globalVar_7_func_ext_int@29 = (ConcreteLattice<signed int>, 19)
   p2@30 = (ConcreteLattice<signed int>, 18)
   D_1755@30 = (ConcreteLattice<signed int>, 19)
   p2@31 = (ConcreteLattice<signed int>, 18)
   retval_1@31 = (ConcreteLattice<bool>, 1)
   func_ext_int_parametername_a@32 = (ConcreteLattice<signed int>, 18)
   globalVar@32 = (ConcreteLattice<signed int>, 19)
   globalVar@33 = (ConcreteLattice<signed int>, 19)
   globalVar_5_func_ext_int@33 = (ConcreteLattice<signed int>, 19)
   globalVar_6_func_ext_int@34 = (ConcreteLattice<signed int>, 19)
   globalVar_7_func_ext_int@37 = (ConcreteLattice<signed int>, 18)
```

```
    p1@38 = (ConcreteLattice<signed int>, 16)
    globalVar@38 = (ConcreteLattice<signed int>, 18)
    D_1762@38 = (ConcreteLattice<signed int>, 16)
    p2@39 = (ConcreteLattice<signed int>, 18)
    globalVar_4@39 = (ConcreteLattice<signed int>, 18)
    iftmp_3@40 = (ConcreteLattice<bool>, 1)
    retval_2@41 = (ConcreteLattice<bool>, 1)


TRACE 14
TRACE COMPLETED

TRACE
{ (error_0x40bf02d8 = 0;),
   (globalVar_0 = (-p2);),
   (globalVar = globalVar_0;),
   (func_ext_int_parametername_a = p1;),
   (__rtt_precondition_begin__();),
   (globalVar_5_func_ext_int = globalVar;),
   (globalVar_6_func_ext_int = globalVar;),
   (__rtt_precondition_end__();),
   (D_1755 = func_ext(p1);),
   (__rtt_postcondition_begin__();),
   (globalVar_7_func_ext_int = globalVar@func_ext;),
   (__rtt_postcondition_end__();),
   (D_1755 = func_ext@RETURN;),
   (retval_1 = (D_1755 > p2);),
   (func_ext_int_parametername_a = p2;),
   (__rtt_precondition_begin__();),
   (globalVar_5_func_ext_int = globalVar;),
   (globalVar_6_func_ext_int = globalVar;),
   (__rtt_precondition_end__();),
   (D_1762 = func_ext(p2);),
   (__rtt_postcondition_begin__();),
   (globalVar_7_func_ext_int = globalVar@func_ext;),
   (__rtt_postcondition_end__();),
   (D_1762 = func_ext@RETURN;),
   (iftmp_3 = 0;),
   (retval_2 = iftmp_3;),
   (<EMPTYSTATEMENT>;),
   (return;) }
   traceState:      CONT_OF_ANOTHER_TRACE
   currentStepNr:  27
   feasible: 1

CONSTRAINT:
((((((((((((((func_ext_int_parametername_a@24 > 0) &&
 (func_ext_int_parametername_a@24 == ((int) p1@23))) &&
 (p1@23 == ((int) p1@0))) &&
 (globalVar_5_func_ext_int@25 >= -19) &&
 (globalVar_5_func_ext_int@25 == ((int) globalVar@24)) &&
 (globalVar@24 == ((int) globalVar_0@22)) &&
 (globalVar_0@22 == ((int) (-p2@21))) &&
 (p2@21 == ((int) p2@0))) &&
 (globalVar_6_func_ext_int@26 <= 19) &&
 (globalVar_6_func_ext_int@26 == ((int) globalVar@25)) &&
 (globalVar@25 == globalVar_0@22)) &&
 (func_ext@RETURN@0 <= 19)) &&
 (globalVar_7_func_ext_int@29 > 17) &&
 (globalVar_7_func_ext_int@29 == ((int) globalVar@func_ext@0))) &&
 (retval_1@31 != 0) &&
 (retval_1@31 == ((bool) (D_1755@30 > p2@30))) &&
  (D_1755@30 == func_ext@RETURN@0) &&
  (p2@30 == ((int) p2@0))) &&
  (func_ext_int_parametername_a@32 > 0) &&
  (func_ext_int_parametername_a@32 == ((int) p2@31)) &&
  (p2@31 == ((int) p2@0))) &&
  (globalVar_5_func_ext_int@33 >= -19) &&
  (globalVar_5_func_ext_int@33 == ((int) globalVar@32)) &&
  (globalVar@32 == globalVar@func_ext@0)) &&
  (globalVar_6_func_ext_int@34 <= 19) &&
  (globalVar_6_func_ext_int@34 == ((int) globalVar@33)) &&
  (globalVar@33 == globalVar@func_ext@0)) &&
  (func_ext@RETURN@1 <= 19)) &&
  (globalVar_7_func_ext_int@37 > 17) &&
  (globalVar_7_func_ext_int@37 == ((int) globalVar@func_ext@1))) &&
  (D_1762@38 != p1@38) &&
  (D_1762@38 == func_ext@RETURN@1) &&
  (p1@38 == ((int) p1@0))) &&
  (retval_2@40 == 0) &&
  (retval_2@40 == ((bool) iftmp_3@39)) &&
  (iftmp_3@39 == 0))
SOLUTION:
    p1@0 = (ConcreteLattice<signed int>, 1073741824)
    p2@0 = (ConcreteLattice<signed int>, 15)
    func_ext@RETURN@0 = (ConcreteLattice<signed int>, 19)
    globalVar@func_ext@0 = (ConcreteLattice<signed int>, 19)
    func_ext@RETURN@1 = (ConcreteLattice<signed int>, -1072693245)
    globalVar@func_ext@1 = (ConcreteLattice<signed int>, 1073741840)
    p2@21 = (ConcreteLattice<signed int>, 15)
    globalVar_0@22 = (ConcreteLattice<signed int>, -15)
    p1@23 = (ConcreteLattice<signed int>, 1073741824)
    func_ext_int_parametername_a@24 = (ConcreteLattice<signed int>, 1073741824)
    globalVar@24 = (ConcreteLattice<signed int>, -15)
    globalVar@25 = (ConcreteLattice<signed int>, -15)
    globalVar_5_func_ext_int@25 = (ConcreteLattice<signed int>, -15)
    globalVar_6_func_ext_int@26 = (ConcreteLattice<signed int>, -15)
    globalVar_7_func_ext_int@29 = (ConcreteLattice<signed int>, 19)
    p2@30 = (ConcreteLattice<signed int>, 15)
    D_1755@30 = (ConcreteLattice<signed int>, 19)
    p2@31 = (ConcreteLattice<signed int>, 15)
    retval_1@31 = (ConcreteLattice<bool>, 1)
    func_ext_int_parametername_a@32 = (ConcreteLattice<signed int>, 15)
    globalVar@32 = (ConcreteLattice<signed int>, 19)
    globalVar@33 = (ConcreteLattice<signed int>, 19)
    globalVar_5_func_ext_int@33 = (ConcreteLattice<signed int>, 19)
    globalVar_6_func_ext_int@34 = (ConcreteLattice<signed int>, 19)
    globalVar_7_func_ext_int@37 = (ConcreteLattice<signed int>, 1073741840)
    p1@38 = (ConcreteLattice<signed int>, 1073741824)
    D_1762@38 = (ConcreteLattice<signed int>, -1072693245)
    iftmp_3@39 = (ConcreteLattice<bool>, 0)
    retval_2@40 = (ConcreteLattice<bool>, 0)


TRACE 15
TRACE COMPLETED

TRACE
{ (error_0x40bf02d8 = 0;),
   (globalVar_0 = (-p2);),
   (globalVar = globalVar_0;),
   (func_ext_int_parametername_a = p1;),
   (__rtt_precondition_begin__();),
   (globalVar_5_func_ext_int = globalVar;),
   (globalVar_6_func_ext_int = globalVar;),
   (__rtt_precondition_end__();),
   (D_1755 = func_ext(p1);),
   (__rtt_postcondition_begin__();),
   (globalVar_7_func_ext_int = globalVar@func_ext;),
   (__rtt_postcondition_end__();),
   (D_1755 = func_ext@RETURN;),
   (retval_1 = (D_1755 > p2);),
   (func_ext_int_parametername_a = p2;),
   (__rtt_precondition_begin__();),
   (globalVar_5_func_ext_int = globalVar;),
   (globalVar_6_func_ext_int = globalVar;),
   (__rtt_precondition_end__();),
   (D_1762 = func_ext(p2);),
   (__rtt_postcondition_begin__();),
   (globalVar_7_func_ext_int = globalVar@func_ext;),
   (__rtt_postcondition_end__();),
   (D_1762 = func_ext@RETURN;),
   (globalVar_4 = globalVar;),
   (iftmp_3 = 0;) }
   traceState:     CONT_AFTER_SOLVING
   currentStepNr: 25
   feasible: 1

CONSTRAINT:
(((((((((((((((func_ext_int_parametername_a@24 > 0) &&
  (func_ext_int_parametername_a@24 == ((int) p1@23))) &&
  (p1@23 == ((int) p1@0))) &&
  (globalVar_5_func_ext_int@25 >= -19) &&
  (globalVar_5_func_ext_int@25 == ((int) globalVar@24)) &&
  (globalVar@24 == ((int) globalVar_0@22)) &&
  (globalVar_0@22 == ((int) (-p2@21))) &&
  (p2@21 == ((int) p2@0))) &&
  (globalVar_6_func_ext_int@26 <= 19) &&
  (globalVar_6_func_ext_int@26 == ((int) globalVar@25)) &&
  (globalVar@25 == globalVar_0@22)) &&
  (func_ext@RETURN@0 <= 19)) &&
  (globalVar_7_func_ext_int@29 > 17) &&
  (globalVar_7_func_ext_int@29 == ((int) globalVar@func_ext@0))) &&
  (retval_1@31 != 0) &&
  (retval_1@31 == ((bool) (D_1755@30 > p2@30))) &&
  (D_1755@30 == func_ext@RETURN@0) &&
  (p2@30 == ((int) p2@0))) &&
  (func_ext_int_parametername_a@32 > 0) &&
  (func_ext_int_parametername_a@32 == ((int) p2@31)) &&
  (p2@31 == ((int) p2@0))) &&
  (globalVar_5_func_ext_int@33 >= -19) &&
  (globalVar_5_func_ext_int@33 == ((int) globalVar@32)) &&
  (globalVar@32 == globalVar@func_ext@0)) &&
  (globalVar_6_func_ext_int@34 <= 19) &&
  (globalVar_6_func_ext_int@34 == ((int) globalVar@33)) &&
  (globalVar@33 == globalVar@func_ext@0)) &&
  (func_ext@RETURN@1 <= 19)) &&
  (globalVar_7_func_ext_int@37 > 17) &&
  (globalVar_7_func_ext_int@37 == ((int) globalVar@func_ext@1))) &&
  (D_1762@38 == p1@38) &&
  (D_1762@38 == func_ext@RETURN@1) &&
  (p1@38 == ((int) p1@0))) &&
  (globalVar_4@39 != p2@39) &&
  (globalVar_4@39 == ((int) globalVar@38)) &&
  (p2@39 == ((int) p2@0)) &&
  (globalVar@38 == globalVar@func_ext@1))
```

```
SOLUTION:
    p1@0 = (ConcreteLattice<signed int>, 16)
    p2@0 = (ConcreteLattice<signed int>, 3)
    func_ext@RETURN@0 = (ConcreteLattice<signed int>, 19)
    globalVar@func_ext@0 = (ConcreteLattice<signed int>, 19)
    func_ext@RETURN@1 = (ConcreteLattice<signed int>, 16)
    globalVar@func_ext@1 = (ConcreteLattice<signed int>, 1073741825)
    p2@21 = (ConcreteLattice<signed int>, 3)
    globalVar_0@22 = (ConcreteLattice<signed int>, -3)
    p1@23 = (ConcreteLattice<signed int>, 16)
    func_ext_int_parametername_a@24 = (ConcreteLattice<signed int>, 16)
    globalVar@24 = (ConcreteLattice<signed int>, -3)
    globalVar@25 = (ConcreteLattice<signed int>, -3)
    globalVar_5_func_ext_int@25 = (ConcreteLattice<signed int>, -3)
    globalVar_6_func_ext_int@26 = (ConcreteLattice<signed int>, -3)
    globalVar_7_func_ext_int@29 = (ConcreteLattice<signed int>, 19)
    p2@30 = (ConcreteLattice<signed int>, 3)
    D_1755@30 = (ConcreteLattice<signed int>, 19)
    p2@31 = (ConcreteLattice<signed int>, 3)
    retval_1@31 = (ConcreteLattice<bool>, 1)
    func_ext_int_parametername_a@32 = (ConcreteLattice<signed int>, 3)
    globalVar@32 = (ConcreteLattice<signed int>, 19)
    globalVar@33 = (ConcreteLattice<signed int>, 19)
    globalVar_5_func_ext_int@33 = (ConcreteLattice<signed int>, 19)
    globalVar_6_func_ext_int@34 = (ConcreteLattice<signed int>, 19)
    globalVar_7_func_ext_int@37 = (ConcreteLattice<signed int>, 1073741825)
    p1@38 = (ConcreteLattice<signed int>, 16)
    globalVar@38 = (ConcreteLattice<signed int>, 1073741825)
    D_1762@38 = (ConcreteLattice<signed int>, 16)
    p2@39 = (ConcreteLattice<signed int>, 3)
    globalVar_4@39 = (ConcreteLattice<signed int>, 1073741825)


Operation test() is covered.

Covered:
Total transitions:        100%
Transitions with guards: 100%
```

16.5 Graphical Output

Figure 18 shows the graphical representation of the CFG corresponding to the module under test. All nodes and edges of this CFG are drawn blue, which indicates that all of them were successfully covered.

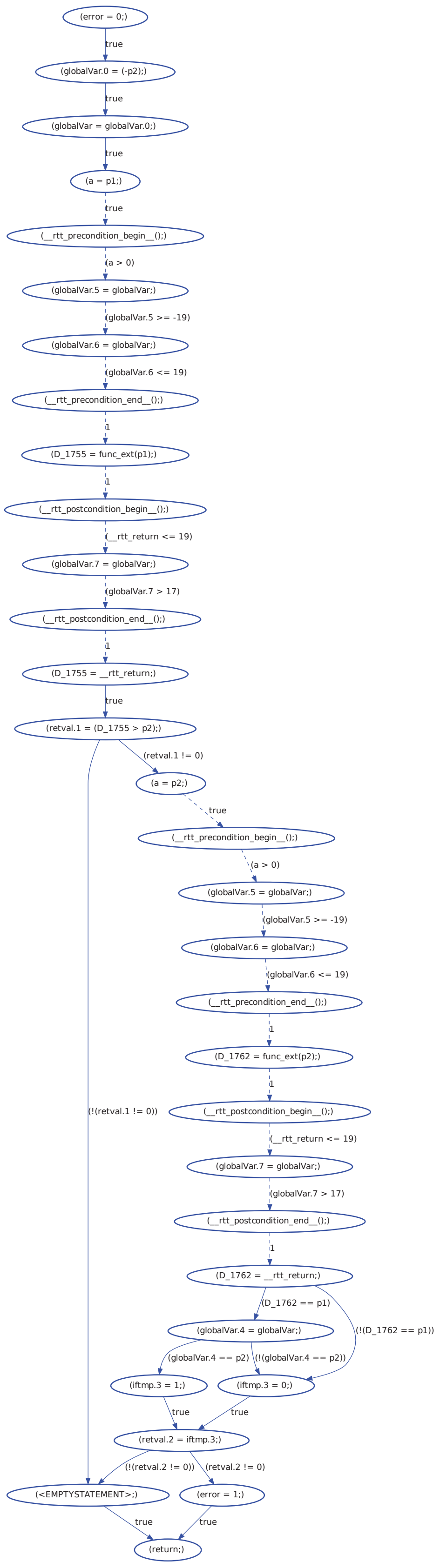


Figure 18: Graphical representation for the example *Undefined Functions with Stub Specification*.