AALTO UNIVERSITY
SCHOOL OF SCIENCE
DEGREE PROGRAMME OF COMPUTER SCIENCE AND ENGINEERING

Olli Saarikivi

Test-Guided Proofs for C Programs on LLVM

Master's Thesis
Espoo, May 16, 2013

Supervisor: Professor Keijo Heljanko
Instructor: M.Sc. (Tech.) Kari Kähkönen

Aalto University
School of Science

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Olli Saarikivi		
Title:	Test-Guided Proofs for C Programs on LLVM		
Date:	May 16, 2013	**Pages:**	61
Professorship:	Theoretical Computer Science	**Code:**	T-79
Supervisor:	Professor Keijo Heljanko		
Instructor:	M.Sc. (Tech.) Kari Kähkönen		

Software defects can be very expensive, especially when encountered in economically critical or safety critical systems. Many of these defects can be avoided if it can be ensured that a program meets its specification. When the specification is given formally, for example with assertions embedded in the source code, automated software verification methods can be applied to determine whether a program complies to its specification.

Recently there has been much interest in combining underapproximation and overapproximation based approaches to software verification. Such a technique is employed by the DASH algorithm originally developed at Microsoft, which generates tests to improve an underapproximation of the program under test. Simultaneously, an overapproximating abstraction of the program is refined with information gathered from test generation.

We present an open source implementation of the DASH algorithm for the verification of C programs compiled on the LLVM compiler framework. Our implementation is an extension of the dynamic symbolic execution tool LCT. We also present a detailed method for constructing the weakest precondition based refinement operator employed by DASH for instructions of the LLVM internal representation.

To maintain a mapping between concrete executions and the abstraction DASH needs to evaluate predicates on the concrete states visited during test executions. A straightforward implementation might store the complete concrete states of each executed test or might employ expensive re-executions to recover the concrete states. We present a technique which allows only the concrete values of pointer variables to be stored while still requiring no re-executions.

Finally we present a case study to show the viability of our tool. We also document a more powerful abstraction refinement method used in DASH and evaluate its effect.

Keywords: automated testing, verification, dynamic symbolic execution, abstraction refinement

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Olli Saarikivi		
Työn nimi:	Testiohjatut todistukset C-ohjelmille LLVM-järjestelmässä		
Päiväys:	16. toukokuuta 2013	**Sivumäärä:**	61
Professuuri:	Tietojenkäsittelyteoria	**Koodi:**	T-79
Valvoja:	Professori Keijo Heljanko		
Ohjaaja:	Diplomi-insinööri Kari Kähkönen		

Ohjelmistovirheet voivat olla hyvin kalliita, varsinkin turvallisuus- ja talouskriittisissä järjestelmissä. Monet näistä virheistä voidaan välttää jos voidaan varmistaa että ohjelmistot täyttävät nille annetut spesifikaatiot. Kun spesifikaatio on annettu formaalisti, esimerkiksi lähdekoodiin kirjoitetuilla assertioilla, voidaan automaattisilla ohjelmiston verifiointimenetelmillä tarkastaa että ohjelmisto täyttää sen spesifikaation.

Viime aikoina on ollut paljon kiinnostusta yhdistää yli- ja aliapproksimoivia lähestymistapoja ohjelmistojen verifiointiin. DASH algoritmi toteuttaa tällaisen menetelmän. Se generoi testejä parantaakseen aliapproksimaatiota verifioitavasta ohjelmasta. Samanaikaisesti DASH hienontaa ohjelmaa yliapproksimoivaa abstraktiota testien generoinnista kerätyllä informaatiolla.

Esittelemme avoimen lähdekoodin työkalun, joka toteuttaa DASH algoritmin LLVM kääntäjäkirjastolla käännettyjen C ohjelmien verifiointiin. Toteutuksemme perustuu dynaamisen symbolisen suorituksen työkaluun LCT:hen. Annamme myös yksityiskohtaisen menetelmän tuottaa DASH:n vaatima heikoimpiin esiehtoihin perustuva abstraktion hienonnusoperaattori LLVM:n välikielen käskyille.

DASH evaluoi predikaatteja testien aikana nähdyille konkreettisille tiloille ylläpitääkseen kuvausta testiajojen ja abstraktion välillä. Suoraviivainen toteutus saattaisi tallentaa jokaisen ajetun testin kokonaiset konkreettiset tilat tai ajaa testejä toistamiseen konkreettisten tilojen uudelleenkonstruoimiseksi. Esittelemme menetelmän jolla ainoastaan osoittimia sisältävien muuttujien arvot tallennetaan ilman että testejä pitää uudelleenajaa.

Lopuksi näytämme työkalumme käyttökelpoisuuden kokeiden avulla. Dokumentoimme myös DASH algoritmissa käytetyn tehokkaamman abstraktion hienonnusoperaattorin sekä esittelemme sen vaikutuksen kokeellisesti.

Asiasanat:	automaattinen testaus, verifiointi, dynaaminen symbolinen suoritus, abstraktion hienontaminen
Kieli:	Englanti

Contents

Chapter 1

Introduction

As our reliance on software grows, ensuring software correctness is an increasingly important task for the software industry. A 2002 report conducted by NIST estimates that the US economy loses $60 billion each year to problems associated with software errors [41]. The cost of a single Microsoft security bulletin has been estimated to be in the order of millions of dollars [15]. In safety critical systems software errors can also lead to loss of life. For example, the deaths and serious injuries caused by the radiation therapy machine Therac-25 are in part attributable to software errors [30].

One approach to preventing software errors is to design a specification, against which the software will be written. The problem of determining whether a program complies to its specification is called software verification. This process can be at least partly automated when the specification is formal enough. Automated verification methods involve exploration of the *state space* of the program to either find a violation of the specification or a proof of program correctness. However, the state space of a program can be very large as it often is exponential in the number of program locations and amount of memory used by the program. This problem, dubbed state space explosion [38], often makes enumerating all states of the program infeasible.

One approach to exploring the state space of a program is *testing*, where a program is executed with sets of input values that will drive the execution to different parts of the state space. Typically program specifications involve the reachability of specific program locations and therefore it is desirable for the set of test executions to provide a good coverage of the program's source code. Traditionally the test inputs are selected manually, e.g. by the programmer, which is often a slow process. One way to automate the process is to generate the test inputs randomly. While this technique, also known as fuzzing, can be effective in some cases [31], it can also generate many tests that do not increase coverage. A directed test generation method called dynamic symbolic execution (DSE) counters this problem by generating input values that will drive the tests to previously unexplored execution paths [7, 13, 36, 37]. It has also been called whitebox fuzzing in contrast to the "blackbox" approach of traditional fuzzing [14]. In DSE the execution of a test is monitored to collect a symbolic constraint over the program's input variables for the path taken by the execution. This constraint can then be supplied to a satisfiability modulo theories (SMT) solver to generate new inputs that will drive subsequent executions to follow previously unexplored paths.

Even with each new test exploring a previously unexplored path, DSE does not yet scale well to large real-world programs [14]. Gunter and Peled [18] propose a method for letting the user specify with an LTL formula which parts of the program under test should be explored, which could be useful e.g. for unit testing. In any case, except for some trivial programs, verification methods based on testing will underapproximate the state space, i.e. the set of states explored by the test executions will be a subset of the program's state space. This means that once the testing is done we can not say for sure whether the program satisfies its specification: there could still exist unexplored inputs to the program that would cause it to violate the specification.

While explicit state model checking of software is typically infeasible, it may be possible to use an abstracted version of the program to check the specification. With abstraction the behavior of the program is overapproximated by leaving out some details. For example, in predicate abstraction [16] an abstract state space is formed from the different valuations of a set of predicates on concrete variables of the program to be verified. The abstract state space can often have a much smaller representation than the state space of the original program, which may make it feasible to prove that the abstracted program satisfies the specification. Because the abstracted program is an overapproximation, any bad behavior of the original program is also present in the abstraction and therefore if the abstraction is verified then so is the original program. However, if the abstracted program does not satisfy the specification then we can not necessarily say anything about the original program. The bad behavior that violates the specification, called a *counterexample*, may be an actual behavior of the original program, in which case it should be reported as a bug. On the other hand, the counterexample might not be a feasible behavior of the original program, in which case it is called a *spurious* counterexample.

To get rid of spurious counterexamples various *abstraction refinement* methods have been developed. An abstracted program may contain a spurious counterexample when too many details have been left out in the abstraction process. This can be remedied by refining the abstraction, i.e., adding back some previously ignored facts about the program. The process of selecting how to refine the abstraction can be guided by analysis of the counterexample [9]. This approach, called counterexample guided abstraction refinement, has been successfully used in a number of tools [3, 8, 19, 27]. These tools work by iteratively refining the abstraction until it is sufficiently precise to prove that the specification is satisfied (assuming that it indeed is).

To explore the state space of an abstracted program the *abstract transition relation* must be constructed. This relation is induced by the original program when evaluated with respect to the abstract state space. The construction of the abstract transition relation may be costly. For example, with the technique described by Graf and Saidi [16] computing a successor state of a transition requires potentially a large number of theorem prover calls. One way in which this can be alleviated is by only refining the parts of the abstraction needed for eliminating the spurious counterexample. An algorithm following this approach, called lazy abstraction, has been implemented in the BLAST software verification tool [19]. With the lazy abstraction approach the work done to construct the abstract transition relation can be partly reused when the abstraction is refined.

In 2005 Godefroid and Klarlund [12] predicted that testing and abstraction based verification methods could be combined in useful ways. Yorsh, Ball and Sagiv [39] propose one such approach, which aims to alleviate the cost of constructing the abstraction by abstracting from a set of concrete test executions. In their method a theorem prover is used to check that the concrete tests cover all the reachable abstract states instead of directly computing the abstract transition relation. This can in the best case reduce the number of required theorem prover calls when compared to previous methods. Kroening, Groce and Clarke [27] describe the CRunner tool, which also combines information from concrete executions with counterexample guided abstraction refinement. Their approach combines concrete executions with partial SAT-based simulation of the program to make detecting spurious counterexamples more efficient.

Combining DSE-based testing with abstraction refinement is attractive because the two approaches can efficiently handle different types of programs [17]. Abstraction refinement works well for programs that require tracking a relatively small number of predicates to prove a property. Testing, on the other hand, can quickly find feasible paths through code which could otherwise require refining the abstraction many times to discover. The SYNERGY algorithm [17] implemented in Microsoft's YOGI tool [33] combines abstraction refinement with DSE in such a way that it retains advantages from both. It simultaneously constructs a concrete execution tree of the program and a partition of the program into abstract regions. From a testing point of view, SYNERGY works similarly to DSE with the addition that the branches to expand are selected along error traces found in the abstraction. This ensures that only the parts of the program that may have an error are explored by testing. From an abstraction refinement point of view, the test generation of SYNERGY is how spurious counterexamples are detected. If the counterexample is spurious, then at some point an SMT solver call to generate a test further along the error trace will fail. At this point the abstraction is refined to remove the error trace.

The DASH algorithm [4] (also implemented in the YOGI tool) presents several improvements over SYNERGY. The refinement method in DASH for maintaining the abstraction avoids extra solver calls: while SYNERGY uses a solver call to refine the abstraction, DASH uses information from the failed test generation attempt to do the refinement. DASH also presents a technique for efficiently handling programs with pointers by using concrete pointer aliasing information gathered from tests. Finally, while the SYNERGY algorithm only supports single-procedure programs, in DASH programs with multiple procedures are supported via a recursive call to the algorithm for each procedure call.

In this work we present an implementation of the DASH algorithm for C programs. The tool, called LCT-D, is a modification to the DSE tool LCT [21–23]. We have implemented our program transformations with the LLVM compiler framework [29]. The main contributions of this work are as follows:

1. We present LCT-D, an open source tool implementing the DASH algorithm for verifying C programs compiled to the LLVM intermediate representation.

2. We describe a strategy for constructing weakest preconditions for LLVM basic blocks in the presence of pointers.

3. We document and evaluate the effectiveness of a more powerful abstraction refinement method that can be applied when an abstract region's predicate becomes unsatisfiable.

4. We describe a method for mapping traces from test executions back to the abstraction that does not require directly evaluating predicates of abstract regions.

5. We present initial case studies showing the viability of our tool.

Due to time constraints we have placed several restrictions on the scope of our work. First, we have not implemented the interprocedural support for DASH described in [4]. While the feature itself is not conceptually very complex, we estimated that handling all of its implementation details would have been a significant undertaking. Second, we omit support for all pointer arithmetic, including array accesses. How to support arrays with symbolic indices in DASH has not been described and we believe the task is at least non-trivial. Finally, we have only implemented support for the LLVM instructions needed to run our case studies. Most notably this does not include the `getelementptr` instruction, which is used for C structure support.

The rest of this work is structured as follows. Chapter 2 goes over the basics of abstraction refinement and dynamic symbolic execution, and provides an overview of the LLVM intermediate representation. Chapter 3 describes the DASH algorithm in the context of a program model reminiscent of the LLVM intermediate representation. We also present our strategy for constructing weakest preconditions and the alternate abstraction refinement method for unsatisfiable predicates in regions. Chapter 4 describes our implementation, including our method for mapping execution traces back to the abstraction. In Chapter 5 we present preliminary case studies and discuss their results. Finally, Chapter 6 provides some concluding remarks and discussion on directions for future research.

Chapter 2

Background

In this chapter we go over some background theory needed for understanding the DASH algorithm. First we go over the basics of dynamic symbolic execution, on which the test generation method in DASH is based. Next we provide a short explanation of abstraction refinement and step through a running example to familiarize the reader with the type of abstraction employed in DASH. Finally we provide an overview of the LLVM intermediate representation to help in understanding our program model and some special handling presented in Chapter 3.

2.1 Dynamic Symbolic Execution

Dynamic symbolic execution [13, 36, 37] (DSE) is a technique for systematically exploring all execution paths of a program. In DSE symbolic constraints for program inputs are gathered during execution to form a *path constraint* for the execution. This path constraint is then used to solve new sets of inputs that will drive subsequent executions to follow previously unexplored paths.

We now apply DSE to the example program in Figure 2.1. The example program marks its inputs by calls to `input()`. The program also has two lines, 5 and 9, which have calls to `error()`. We would like to find a set of inputs that will cause the program to execute one of these calls.

Initially the program will be executed with random inputs. Let us assume the first random input given to the program is 5, which means that `x == 13` will not hold and the next statement executed will be the `return 0`, terminating the program. During this execution symbolic counterparts for the statements executed are recorded in the *execution tree* shown in Figure 2.2(a). For the statement on line 2 we get the constraint $x_1 = \mathit{input}_1$ indicating that the first input gets assigned to x's first symbolic value x_1. For the if statement two child nodes corresponding to the two branches are added to the tree. The edges to the children are labeled with constraints that must be true for the program to follow the path to the child in question. In our example the constraint for the branch taken in the first execution is $\neg(x_1 = 13)$ and the constraint for the unexplored branch is the negation of the first one, $(x_1 = 13)$.

To generate the inputs for the next execution we select the unexplored branch

```
int main() {                                  1
    int x = input();                          2
    if (x == 13) {                            3
        if (x < 10)                           4
            error();                          5
        int y = input();                      6
        x = x + y;                            7
        if (x < 0)                            8
            error();                          9
    }                                        10
    return 0;                                11
}                                            12
```

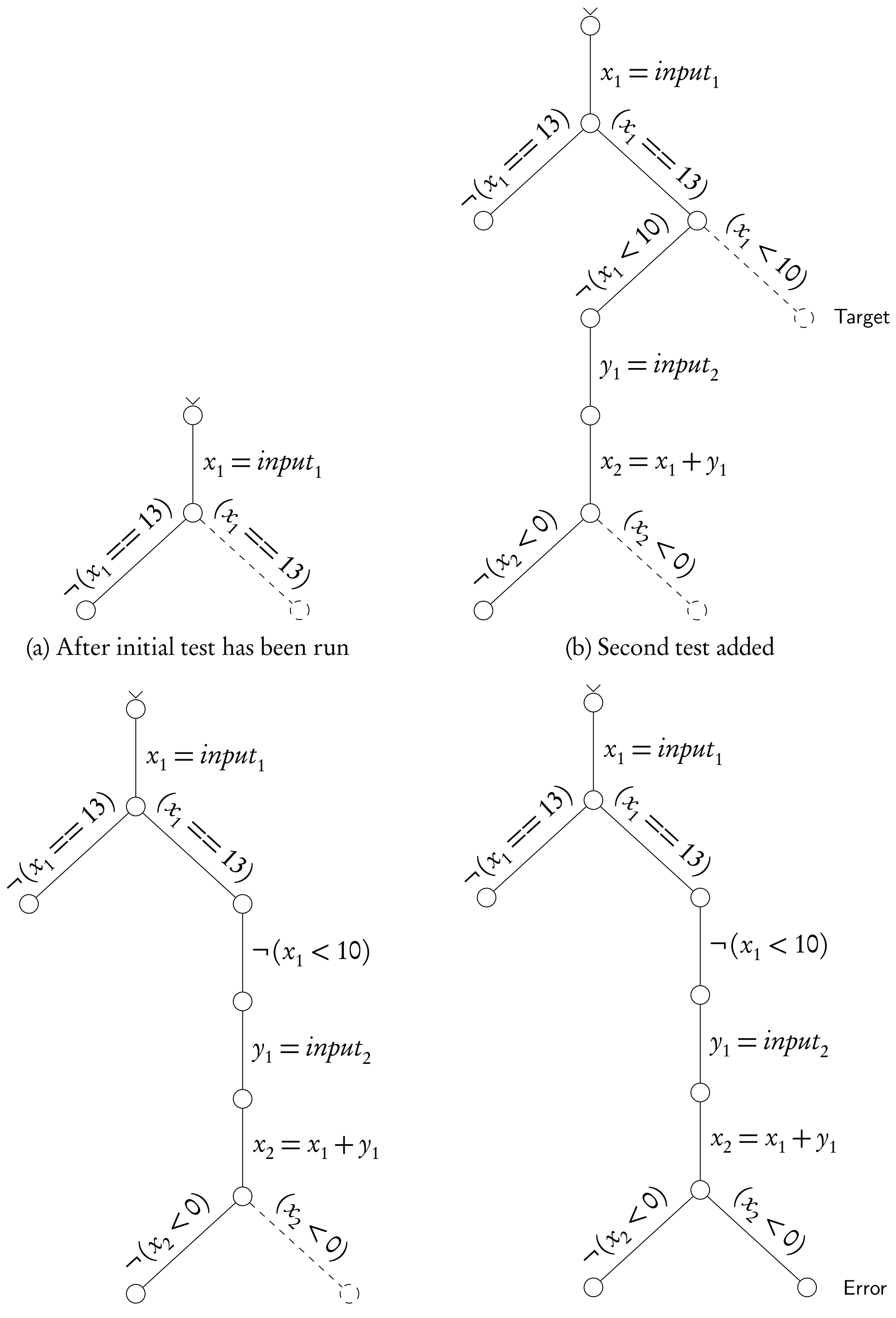
Figure 2.1.: A running example program for DSE

which is shown in Figure 2.2(a) drawn with a dashed line. The path constraint for reaching this branch is $(x_1 = \mathit{input}_1) \wedge (x_1 = 13)$. This constraint is passed to an SMT solver, which will return a satisfying assignment where input_1 gets the value 13. The program is then executed again with the new input assignment. This time the program will go to the true branch of the first if statement and the false branch of the second one. On line 6 a second call to `input()` is encountered. Because this value was not set in the input assignment we instead use a random value. For this example let us assume the value 20 is used, which results in the execution taking the false branch also for the last if statement on line 8.

The path followed by the second execution is added to the execution tree. The updated tree can be seen in Figure 2.2(b). For the next execution let us assume the path selected is the one labeled "Target" in Figure 2.2(b), for which the path constraint is $(x_1 = \mathit{input}_1) \wedge (x_1 = 13) \wedge (x_1 < 10)$. When querying an SMT solver for an assignment for this constraint we will find that the constraint is unsatisfiable (evident from x_1 being required to be both 13 and less than 10). From this we know that there exist no inputs with which the program would reach the node at the end of the target path. Therefore, we remove that node from the execution tree resulting in the tree shown in Figure 2.2(c).

The next iteration begins by selecting the lone unexplored path in Figure 2.2(c), and passing its path constraint, $(x_1 = \mathit{input}_1) \wedge (x_1 = 13) \wedge \neg(x_1 < 10) \wedge (y_1 = \mathit{input}_2) \wedge (x_2 = x_1 + y_1) \wedge (x_2 < 0)$, to an SMT solver. We get a satisfying input assignment of, for example, $\mathit{input}_1 = 13$ and $\mathit{input}_2 = -15$. Executing the program with these inputs will cause it to follow the desired path after which it executes line 9 and thus reaches an error. The final execution tree with the path to the error is shown in Figure 2.2(d).

For a more in depth explanation of DSE see for example the paper by Godefroid, Klarlund and Sen [13].



(a) After initial test has been run

(b) Second test added

(c) Infeasible path removed

(d) Final execution tree

Figure 2.2.: Various stages of an execution tree constructed with dynamic symbolic execution

```
int main() {                    1
    int x = 0;                  2
    int y = x;                  3
    while (x == y) {            4
        x++;                    5
        if (x == y)             6
            error();            7
        y = input();            8
    }                           9
    return 0;                   10
}                               11
```

Figure 2.3.: A running example program for abstraction refinement

2.2 Abstraction Refinement

Abstraction refinement is a technique for model checking large or infinite state systems by trying to verify the property of interest in a simpler abstraction of the system. The abstraction may exhibit *spurious counterexamples* (ones that do not correspond to an execution of the concrete system) and because of this an *abstract-check-refine loop* is used [3, 9, 19]:

abstract Choose a set of predicates and build an abstraction of the program. The states in the abstraction, which are also called regions, correspond to truth assignments of the chosen predicates.

check Check whether the abstraction conforms to the property we wish to verify. If it does then we can stop with a "pass" result. Otherwise, we get an abstract counterexample. If this counterexample is also an execution of the program then we can stop with a "fail" result. If not, then we have a spurious counterexample and we proceed to the next step.

refine Infer a new predicate, which will remove this spurious counterexample. Add this new predicate to the predicates used to build the abstraction and return to the first step.

Consider the running example program in Figure 2.3, for which we wish to verify that the error on line 7 is not reachable. Dynamic symbolic execution (see Section 2.1) can not be used to verify this, because due to the loop on line 4 the program has an infinite number of execution paths. We will show how abstraction refinement can be applied to this verification task.

On the first iteration of the abstract-check-refine loop we have no predicates selected for the abstraction. Therefore, the initial abstraction is simply the program's control flow graph (CFG) shown in Figure 2.4. From this we can then find a counterexample path of $(2,3,4,5,6,7)$. To check whether this counterexample is spurious we can

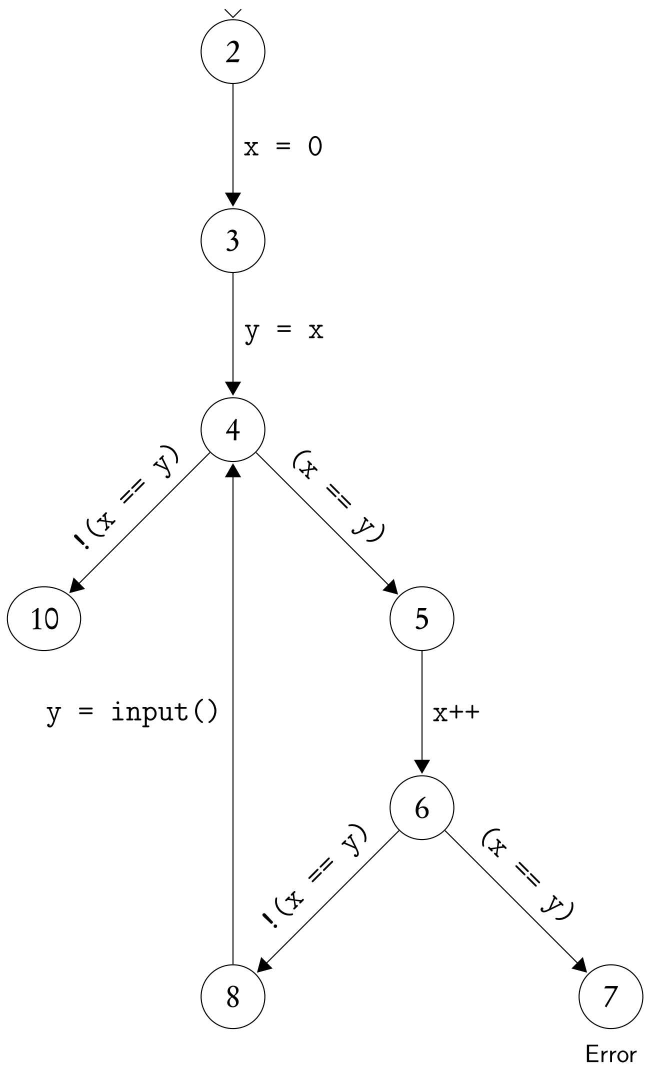


Figure 2.4.: CFG for the example program

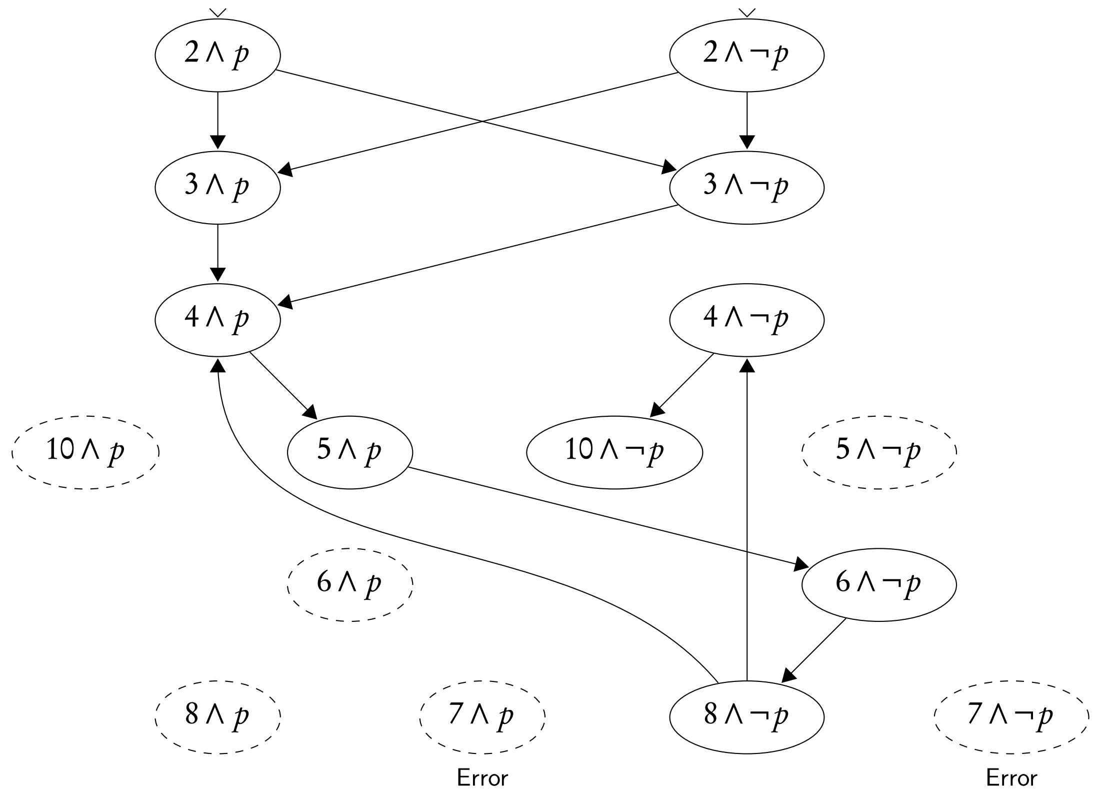


Figure 2.5.: Regions of the abstraction constructed with the predicate $p := (x = y)$

gather the constraints for the path projected to the original program. For our example the path constraint is $(x_1 = 0) \wedge (y_1 = x_1) \wedge (x_1 = y_1) \wedge (x_2 = x_1 + 1) \wedge (x_2 = y_1)$. By passing this constraint to an SMT solver we would find out that the constraint is unsatisfiable and, therefore, the counterexample is indeed spurious.

Now we must select a predicate which eliminates this spurious counterexample. It turns out that adding the predicate $p := (x = y)$ will work. The details of how the predicates to refine with are inferred can be quite involved and vary between different approaches to abstraction refinement. We omit these details here for brevity.

On the second iteration we construct the abstraction while observing the predicate p. The state space of the abstraction is shown in Figure 2.5. For each region in the original abstraction we now have two versions: one in which p is true and another in which $\neg p$ is true. A state of the concrete program belongs to a region if it is in the corresponding program location and predicate for the region is true in that state. The transition relation in the abstraction is such that there is a transition between two regions if there is also a transition between two concrete states belonging to the regions. Regions that are not reachable from an initial region (i.e. a region containing an initial state) are drawn with dashed lines and transitions from unreachable regions are omitted.

To get a sense of how the transitions between regions work look at the transition from $5 \wedge p$ to $6 \wedge \neg p$. This transition exists because if p is true and the statement `x++` is executed then in the next state p will necessarily be false as y can not be x and $x+1$ at the same time. For the same reason a transition from $5 \wedge p$ to $6 \wedge p$ does not exist.

Regions can also have multiple outgoing transitions. For example from $8 \wedge \neg p$ there is a transition both to $4 \wedge p$ and $4 \wedge \neg p$. This is because whether executing the statement `y = input()` makes p true or false depends on the nondeterministic input.

If we check the new abstraction for a counterexample we will find that the error regions are now unreachable. Because any concrete execution of the program can be simulated by the abstraction we have now verified that the example program can not execute the error statement.

In this example we presented the state space of the abstraction directly. In actual implementations of the technique the representation of the abstract model may be more compact. For example boolean programs have been used [3] or the abstraction may be represented implicitly in a search tree [19].

2.3 The LLVM Intermediate Representation

Our tool LCT-D targets verification of C programs compiled to the intermediate representation of the LLVM compiler framework, which is a flexible compiler backend and a collection of related technologies. The program model we use in Chapter 3 to describe the DASH algorithm is reminiscent of the LLVM intermediate representation (LLVM IR). This section will give an overview of the basic concepts of the LLVM IR.

The LLVM IR [1, 28] is a type-safe, low-level, static single assignment (SSA) [2, 34] based representation used in LLVM for most program transformations. The LLVM IR uses a register-based representation, where an unlimited number of named registers are available. Due to the SSA form, for each used register there exists exactly one instruction that assigns to it. Situations where alternate values need to be assigned to the same register are encoded with the `phi` instruction, which selects a value based on along which control flow edge the instruction is reached. The `phi` instruction is explained in more detail in Section 3.3, where its handling in the DASH algorithm is presented.

LLVM IR exists in three equivalent representations: as C++ data structures, in a binary representation called LLVM bitcode or in a human readable assembly language. For our purposes the assembly language representation is the most useful to understand. An example of the assembly language can be seen in Figure 2.7, which is the result of compiling the C program in Figure 2.6.

Control flow inside functions is represented as jumps between *basic blocks*, which are lists of non-terminator instructions followed by a single *terminator instruction*. Terminator instructions encountered in LLVM IR generated from C programs include branching, goto and return instructions. Additionally an `unreachable` instruction may be used to indicate that the end of that basic block is never reached. Non-terminator instructions include for example all arithmetic, conversion and comparison instructions.

```
int a = 20;

int main() {
    int x = 0;
    int y;
    if (x > a) {
        x = x - 10;
        y = x;
    } else {
        y = 0;
    }
    return y;
}
```

Figure 2.6.: An example program to be compiled to LLVM IR

```
@a = global i32 20, align 4

define i32 @main() nounwind uwtable {
entry:
  %x = alloca i32, align 4
  store i32 0, i32* %x, align 4
  %0 = load i32* %x, align 4
  %1 = load i32* @a, align 4
  %cmp = icmp sgt i32 %0, %1
  br i1 %cmp, label %if.then, label %if.end

if.then:
  %2 = load i32* %x, align 4
  %sub = sub nsw i32 %2, 10
  store i32 %sub, i32* %x, align 4
  br label %if.end

if.end:
  %y = phi i32 [ %sub, %if.then ], [ 0, %entry ]
  ret i32 %y
}
```

Figure 2.7.: The example program in Figure 2.6 compiled to LLVM IR

Memory in LLVM IR is accessed through the `load` and `store` instructions. While in LLVM IR the memory is addressed on the byte level, our tool only supports programs in which all memory accesses either do not overlap or access exactly the same bytes. In the rest of this work we will only consider such programs. Memory is allocated in three ways: statically by declaring a global variable, dynamically on the stack through the `alloca` instruction or dynamically on the heap by calling a runtime library function (e.g. `malloc()` in C).

Chapter 3

DASH on LLVM

In this chapter we explain the DASH algorithm [4], which combines dynamic symbolic execution with counterexample guided abstraction refinement. DASH attempts to generate tests based on counterexamples found in the abstraction. When DASH fails to generate a test it refines the abstraction to remove the counterexample. The tests can be seen as an underapproximation of the reachable states of the program under test, which DASH tries to expand to include an error. The abstraction on the other hand is an overapproximation which, if error free, also proves the program under test to be so.

Determining the reachability of error states for arbitrary programs is in general undecidable and as such running DASH on a program has three potential outcomes:

- An error is found and inputs that lead the program to the error are returned.
- The verification succeeds if the abstraction is refined to remove all error traces.
- The algorithm may not terminate.

The following description of the DASH algorithm has been adapted from Beckman et al. [4]. In our model a program P is a tuple (N, E, n_0, λ, B), where:

- N is a finite set of nodes corresponding to program locations.
- $E \subseteq N \times N$ is a set of control flow edges.
- n_0 is the program's entry point.
- $\lambda : E \rightarrow \mathit{Stmts}$ labels each control flow edge with a statement.
- B is the set of variables in the program. We assume that all variables are of either integer or pointer type.

In the following explanation we use the notion of a *map*, which is a partial function with some additional notation for updating it. Given a map M we adopt the following notations:

- $M(x)$ is the value of x in the map M.

- $M[x \mapsto y]$ is a map otherwise identical to M, except that now x maps to y.
- $M[x \not\mapsto]$ is a map otherwise identical to M, except that the mapping for x has been removed.

The set *Stmts* in the definition above is the set of statements present in the program P. Table 3.1 lists the types of statements used in our program model. Programs written with these statements correspond to single procedure programs in a subset of the LLVM IR. For better readability we have chosen to write the binary and the comparison expressions in the style of the C programming language.

The *state* of a program is a tuple (n, Y, W, U), where n is the current program location, Y and W map the program's variables and memory addresses, respectively, to their current values and U is a set of addresses that have already been used for memory allocation. The state space Σ of the program is the product of all possible values of n, Y, W and U. Now we define the transition relation $\rightarrow \subseteq \Sigma \times \Sigma$ so that for all $s, s' \in \Sigma$ it holds that $s \rightarrow s'$ if and only if there is a statement $op \in Stmts$ that is enabled in s such that executing op in s according to the semantics laid out in Table 3.1 would take the program to the state s'. Note that the semantics do include undefined operations. For example we do not define what happens on division by zero. However, from now on we assume that programs have no concrete executions which would result in undefined behavior.

Let $s_0 \in \Sigma$ be the initial state of the program. A property $\varphi \subseteq \Sigma$ is a set of states that we want to verify to not be reachable from s_0. The problem instance is now a pair (P, φ) of the program and the property. Let $\rightarrow^+$ be the transitive closure of the relation $\rightarrow$. Now the answer to (P, φ) is "fail" if there exists a state $s \in \varphi$ such that $s_0 \rightarrow^+ s$ and "pass" otherwise.

As a set of states, the property φ can be any property expressible as a predicate over the program's variables and memory. This includes any property expressible with the C language's `assert` macro.

3.1 The Algorithm

For a given problem instance (P, φ) to detect "fail" instances the DASH algorithm will attempt to find a sequence of states $(s_0, s_1, \ldots, s_n)$ such that $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$ and $s_n \in \varphi$. We call such a sequence of states an *error trace*.

To detect "pass" instances the DASH algorithm maintains an abstraction $\Sigma_\simeq$ which partitions the state space Σ into a finite number of equivalence classes. We refer to the equivalence classes of the abstraction $\Sigma_\simeq$ as *regions*. Let $\rightarrow_\simeq \subseteq \Sigma_\simeq \times \Sigma_\simeq$ be a transition relation such that for all regions $R, R' \in \Sigma_\simeq$ it holds that $R \not\rightarrow_\simeq R'$ if and only if there exist no states $s \in R$ and $s' \in R'$ such that $s \rightarrow s'$. In other words, the transition relation $\rightarrow_\simeq$ may be an overapproximation. Let $R_0 \in \Sigma_\simeq$ be the region that contains the initial state s_0 and $\varphi_\simeq$ be the regions that contain a state from φ. An *abstract error trace* is a sequence of regions $(R_0, R_1, \ldots, R_n)$ such that $R_0 \rightarrow_\simeq R_1 \rightarrow_\simeq \cdots \rightarrow_\simeq R_n$ and $R_n \in \varphi_\simeq$.

Given the current state (n, Y, W, U), a statement *op* is enabled if there exists a control flow edge $(n, n') \in E$ such that $op = \lambda(n, n')$. After a statement is executed the new state is (n', Y', W', U'). How Y', W' and U' change depends on the statement.

Statement	Semantics
r `=` v_1 `<op>` v_2	The program moves to the state (n', Y', W, U), where $Y' = Y[r \mapsto \text{Eval}(v_1$ `<op>` $v_2)]$. The function Eval(e) computes the result of an expression e. If the operands v_1 and v_2 are variables they are determined by the current states Y of the program's variables. Alternatively, the operands may also be constants. The operation `<op>` is one of the binary operators available in the C language. All operands are 32 bit integers. Booleans are represented with zero as false and other values as true. We do not consider pointers to be integers, i.e., all pointer arithmetic is forbidden. We assume that no undefined operations (e.g. divide by zero) are encountered.
r `=` v	The program moves to the state (n', Y', W, U), where $Y' = Y[r \mapsto Y(v)]$, i.e., the value of v is assigned to r. The operand v may alternatively be a constant, in which case the destination state is one with $Y' = Y[r \mapsto v]$.
r `= input`	The program moves to a state where the value of the variable r in Y is replaced with an arbitrary integer. Because this statement induces multiple transitions the result of executing it is nondeterministic.
r `= load` a	The program moves to the state (n', Y', W, U), where $Y' = Y[r \mapsto W(a)]$.
`store` v a	The program moves to the state (n', Y, W', U), where $W' = W[a \mapsto Y(v)]$. If the operand v is a constant then the destination state is instead one with $W' = W[a \mapsto v]$.
a `= allocate`	A memory address $m \in \mathbb{N}$ such that $m \notin U$ is selected and the program moves to the state (n', Y', W, U'), where $Y' = Y[a \mapsto m]$ and $U' = U \cup \{m\}$. Because no pointer arithmetic is allowed, the way m is selected is not important.

`(`v_1 `<comp>` v_2`)`
 `!(`v_1 `<comp>` v_2`)`
	In addition to the requirement that the program location is correct these statements are only enabled if also the condition represented is true. Otherwise these guard statements have no effect. The comparison operator `<comp>` is one of the comparison operators available in the C language.

Table 3.1.: The statements of the program model and their semantics

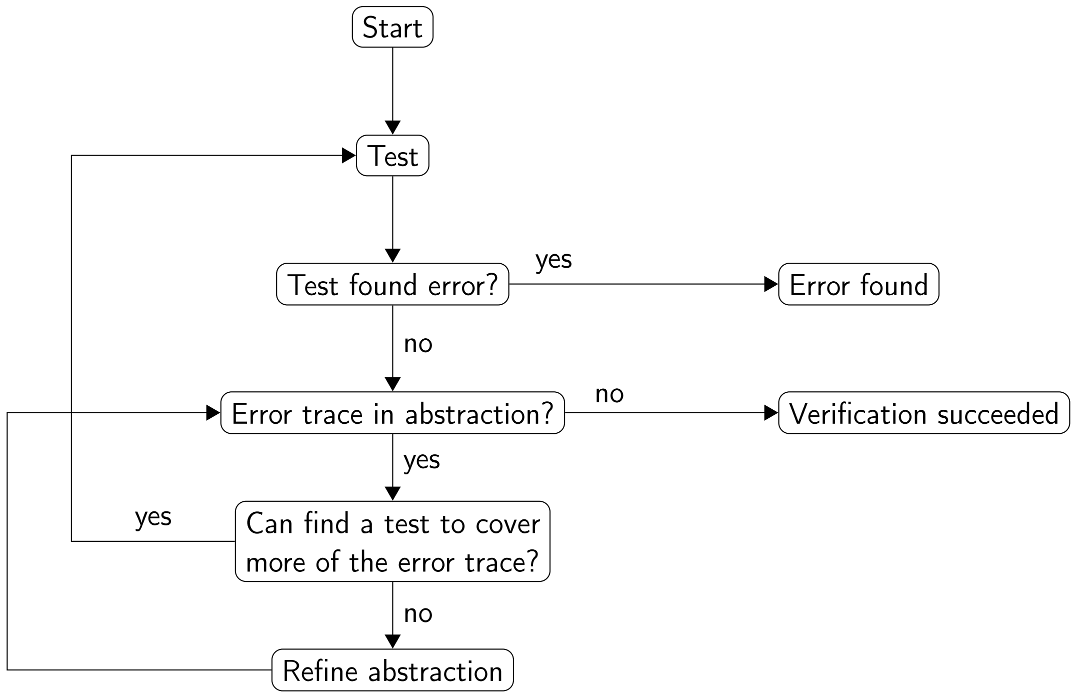


Figure 3.2.: Flowchart for the DASH algorithm

Theorem 1. *Given a problem instance (P, φ), if there exists an abstraction which does not have an abstract error trace then the answer to the instance is "pass".*

Proof. Assume (P, φ) is a "fail" instance and $\Sigma_\simeq$ is an abstraction with no abstract error trace. There exists an error trace $(s_0, s_1, \ldots, s_n)$. Let Region : $\Sigma \to \Sigma_\simeq$ be a function that maps each state to the region that contains it. Now for the sequence of regions $(T_0, T_1, \ldots, T_n)$ such that $T_i = \text{Region}(s_i)$ for $i \in [0 \ldots n]$ it holds that: (1) because $s_0 \to s_1 \to \cdots \to s_n$ then from the definition of $\to_\simeq$ it follows that $T_0 \to_\simeq T_1 \to_\simeq \cdots \to_\simeq T_n$ and (2) $T_n \in \varphi_\simeq$ because $s_n \in \varphi$ and $\text{Region}(s_n) = T_n$. Therefore $\Sigma_\simeq$ does have an abstract error trace and the original assumption must be false. $\square$

In addition to the abstraction, the DASH algorithm maintains a set C of test runs, which are pairs consisting of an *execution trace*, represented by a sequence of states, and a sequence of input values.

The flowchart in Figure 3.2 presents a rough sketch of the DASH algorithm. Pseudocode for its main procedure is presented in Figure 3.3. DASH follows a modified version of the abstract-check-refine loop introduced in Section 2.2. Whenever an abstract error trace is found, DASH will attempt to generate a test that extends the set of regions that are known to be reachable along the abstract error trace. If the test generation fails then the counterexample was spurious and DASH will refine the abstraction to eliminate the abstract error trace. The main procedure uses the auxiliary procedures `GetAbstractTrace`, `GetCombinedConstraint`, `Solve`, `RunTest`, `RefinePred` and `SplitFrontier`. The purpose of these procedures will be explained in the following explanation of the algorithm.

```
Input   : $P$, $\varphi$
Output : ("fail", $t$), where t is an error trace of $P$, or
         ("pass", $\Sigma_{\simeq}$), where $\Sigma_{\simeq}$ does not have an abstract error trace
$\Sigma_{\simeq} := \bigcup_{n \in N} \{\{s \in \Sigma \mid \text{the program location of } s \text{ is } n\}\}$
$\rightarrow_{\simeq} := \{(R,R') \in \Sigma_{\simeq} \times \Sigma_{\simeq} \mid \text{exist } s \in R \text{ and } s' \in R' \text{ such that } (s,s') \in E\}$
$C := \{\texttt{RunTest}(\epsilon, P)\}$
loop
    if $\Sigma_{\simeq}$ has an abstract error trace then
        $\tau :=$ GetAbstractTrace($\Sigma_{\simeq}$, $\rightarrow_{\simeq}$, $\varphi$)          // Find counterexample
        $(\phi, A) :=$ GetCombinedConstraint($\tau$, $P$, $C$)
        $(isSat, I_{test}) :=$ Solve($\phi$)                              // Check if spurious
        if $isSat$ then
            $(t, I_{all}) :=$ RunTest($I_{test}$, $P$)              // If not, extend frontier
            $C := C \cup \{(t, I_{all})\}$
            if $t$ has a state from $\varphi$ then
                return ("fail" , $t$)
            end if
        else
            $p :=$ RefinePred($\tau$, $A$)      // If spurious, refine abstraction
            $(\Sigma_{\simeq}, \rightarrow_{\simeq}) :=$ SplitFrontier($\Sigma_{\simeq}$, $\rightarrow_{\simeq}$, $\tau$, $p$)
        end if
    else
        return ("pass" , $\Sigma_{\simeq}$)
    end if
end loop
```

Figure 3.3.: The DASH algorithm

To explain the algorithm we will step through a running example of applying DASH to verify the program in Figure 3.4. The example program takes one input, which is marked by the call to `input()`. We wish to verify that no matter what this input value is the program can not execute the error statement on line 6.

At startup DASH creates the initial abstraction $\Sigma_{\simeq}$ and related transition function $\rightarrow_{\simeq}$ from the program's control flow graph: states from Σ are partitioned into regions such that there is one region for each separate program location. There is a transition from a region R to another region R' if the program location of R has an edge to the program location of R'. The initial abstraction of our example program is shown in Figure 3.5. Regions are represented by a program location that the contained states must be in together with an optional constraint written as a predicate over the program's variables and memory. None of the regions in the initial abstraction have any constraints yet, which can be seen by the regions in Figure 3.5 being labeled only by line numbers from Figure 3.4.

We have also labeled edges in Figure 3.5 with their corresponding statements.

```
int main() {
    int x = input();
    if (x>20) {
        x = x-20;
        if (x==0)
            error();
    }
    return 0;
}
```

Figure 3.4.: Example program to verify with DASH

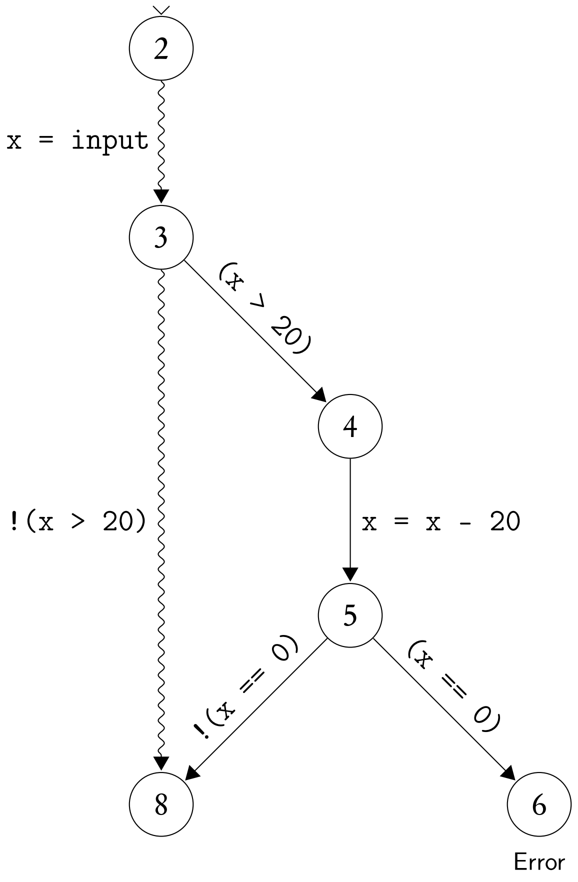


Figure 3.5.: The initial abstraction for the example program in Figure 3.4

Notice how the edges $(3,4)$ and $(3,8)$ are labeled with guard statements. These are used to represent the "then" and "else" branches of the if statement on line 3 of the program in Figure 3.4.

Before entering the main loop DASH will run one test on the program with random inputs, which is accomplished by calling the procedure `RunTest` with an empty sequence of inputs. This results in a random test because once values in the provided input sequence run out, `RunTest` will supply random values to any r `= input` statements executed. The procedure `RunTest` returns a pair containing the sequence of states executed together with the input values used (including random ones).

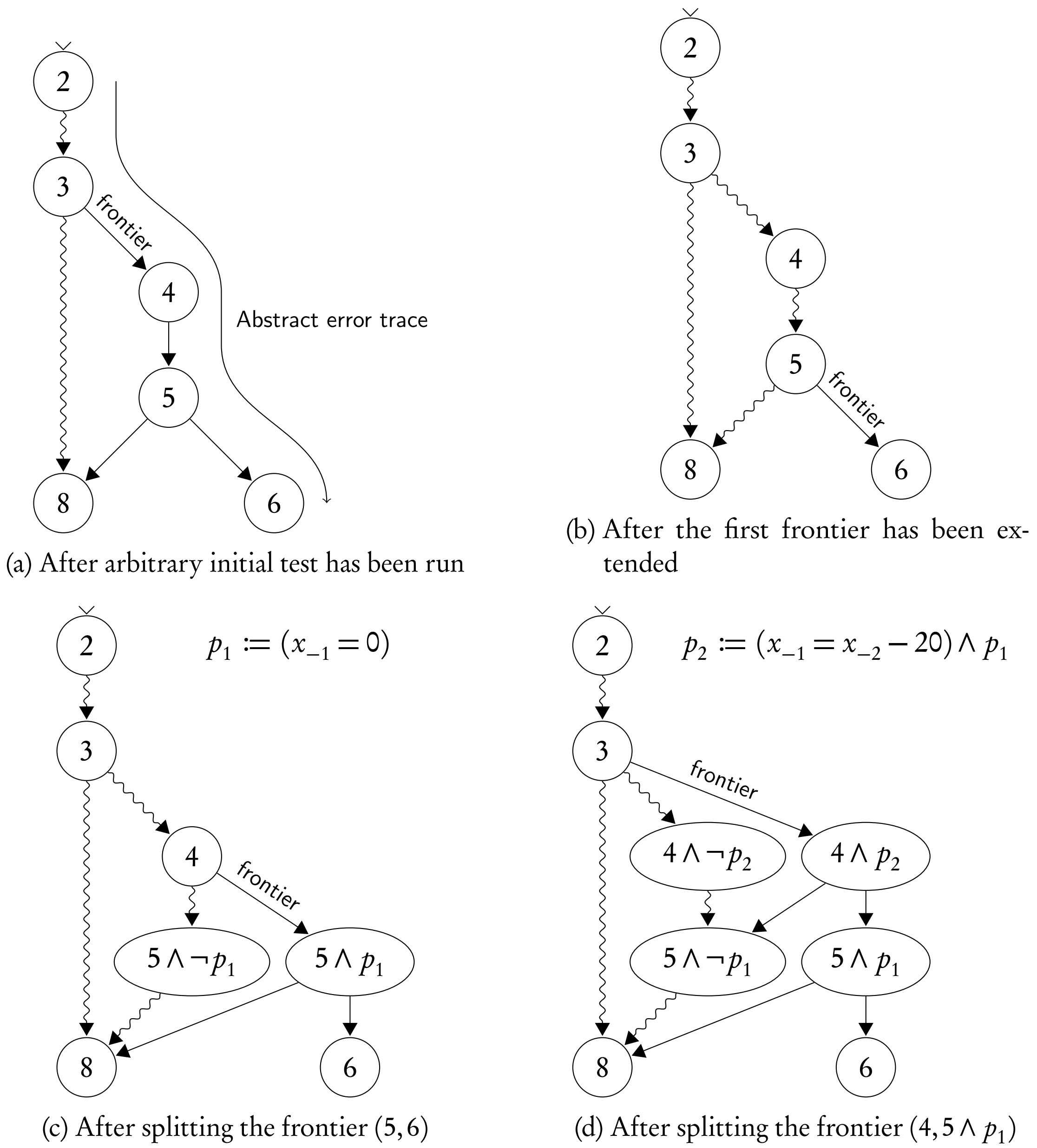
For our example let us assume that on the line 2 in Figure 3.4 the variable `x` is initialized to a value of 15. The set C is initialized to contain this random test, which is a pair $((2,3,8),(15))$. This is visualized in Figure 3.5 with the edges that are covered by the trace being wavy.

After initialization, the first iteration of DASH's main loop starts by finding the abstract error trace $\tau = (2,3,4,5,6)$, also shown in Figure 3.6(a). The call to the procedure `GetAbstractTrace` in Figure 3.3 always returns an abstract error trace such that after the first region *not visited* by a test from C no subsequent region has been visited either. If any abstract error trace exists, we can be sure that there is one that satisfies this requirement because any abstract error trace can be converted to fulfill it. Given an arbitrary abstract error trace $\tau' = (R_0, R_1, \ldots, R_n)$ let R_{k-1} be the last region that contains a state $s_{i-1} \in C$ and let $(s_0, s_1, \ldots, s_{i-1})$ be the execution trace leading up to s_{i-1}. Now, the desired abstract error trace is $\tau'' = (R'_0, R'_1, \ldots, R'_{i-1}, R_i, \ldots, R_m)$, such that $(R_i, \ldots, R_m) = (R_k, \ldots, R_n)$ and for each $h \in [0 \ldots i-1]$ it holds that $s_h \in R'_h$.

After finding the abstract error trace, DASH will try to extend the *frontier* of the abstract error trace τ. Frontier(τ, C) is the pair of adjacent regions (R_{k-1}, R_k) such that R_{k-1} has been visited by a test in C and R_k has not. On the first iteration of our example the frontier is $(3,4)$.

First the procedure `GetCombinedConstraint` in Figure 3.7 is called to construct a constraint ϕ that is satisfiable if and only if there exists a sequence of inputs that will cause the program to follow the abstract error trace up to the frontier and across it. To construct ϕ, `GetCombinedConstraint` selects from C a sequence of inputs $I_{frontier}$ that are known to take the program to the frontier. In our example the only test that has a state in the frontier region R_{k-1} is the one with the input sequence (15), which is therefore selected.

To create the constraint we call the procedure `GatherConstraints`, which uses techniques similar to those used in dynamic symbolic execution (see Section 2.1) to gather a *path constraint* of the program's execution to and over the frontier. In our example the statements executed are `x = input` and `x > 20`. The path constraint ϕ_{path} for these is $(x_1 = input_1) \wedge (x_1 > 20)$. Note that x_1 is a variable that represents the first *value* assigned to the program's variable `x`. In general a path constraint is the weakest predicate over the inputs of the program such that the execution will still follow the given path. The combined constraint ϕ is a conjunction of the path constraint ϕ_{path} and the constraint from the region R_k. In our example R_k has no additional constraint and, therefore, we have $\phi := \phi_{path}$. In addition to the path



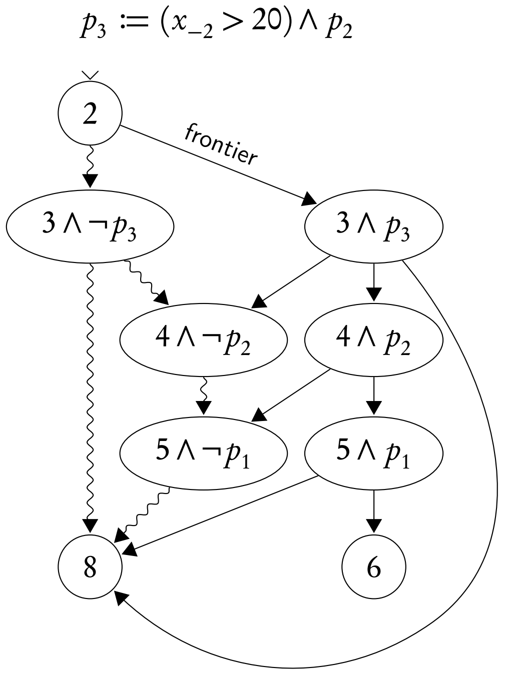
(a) After arbitrary initial test has been run

(b) After the first frontier has been extended

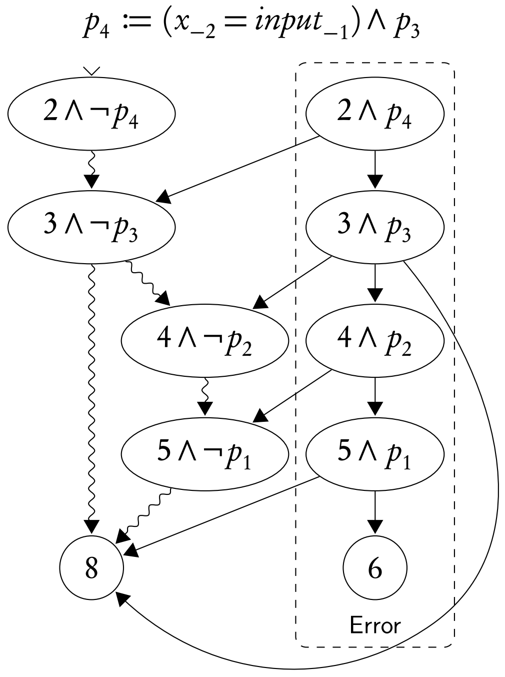
(c) After splitting the frontier (5, 6)

(d) After splitting the frontier $(4, 5 \wedge p_1)$

Figure 3.6.: Various stages of the abstraction for the example program in Figure 3.4



(e) After splitting the frontier $(3, 4 \wedge p_2)$



(f) Final abstraction for the example program

Figure 3.6.: Various stages of the abstraction for the example program in Figure 3.4 (cont.)

```
Output : $(\phi, A)$, where $\phi$ is the combined constraint to the target region and $A$ is the
         assignment of pointers at the frontier
procedure GetCombinedConstraint($\tau$, $P$, $C$)
    $(R_{k-1}, R_k) :=$ Frontier$(\tau, C)$
    $I_{frontier} :=$ inputs $I$ such that $\exists (t, I) \in C : t \cap R_{k-1} \neq \emptyset$
    $(\phi, A) :=$ GatherConstraints($I_{frontier}$, $\tau$, $P$)
    return $(\phi, A)$
end
```

Figure 3.7.: The procedure for constructing the combined constraint to the target region

constrain ϕ, the procedure `GatherConstraints` records pointer aliasing information, which is returned in A. We can ignore this for now as the example program does not use any pointers.

Be aware that when the part of the path constraint for crossing the frontier is generated, the condition value for the branching operator may be concrete (i.e. not dependent on input values). In DSE such a situation is never encountered as only symbolic branches can be subject to test generation. In DASH this situation must be handled by evaluating the concrete constraint at the frontier. If the constraint is true then it can be omitted. If it is false then the whole path constraint will be unsatisfiable and the solver call can be skipped. Alternatively, the concrete constraint can just be added to the path constraint, letting the the SMT solver handle the situation.

Once `GetCombinedConstraint` has returned, the constraint ϕ is passed to the procedure `Solve` (see Figure 3.3), which will indicate whether ϕ is satisfiable or not. `Solve` will also return the assignment as a sequence of input values (in this case only a value for $input_1$) if it is satisfiable. On the first iteration of our example ϕ is satisfiable and we get a satisfying sequence of inputs. Let us say the sequence is (38). This input sequence is then used to run a test with a call to the procedure `RunTest`, which executes the program with the given inputs. `RunTest` returns an *execution sequence* as a sequence t of states the execution visited and the sequence I_{all} of all the inputs given to the program. The sequence I_{all} has as a prefix the inputs given to `RunTest` followed by any that the program may have retrieved after the supplied sequence ran out. For our example program the call `RunTest((38),` P`)` returns a pair $((2,3,4,5,8),(38))$. The new execution trace and inputs are then added to the set C, which is illustrated in Figure 3.6(b).

Note that here we identify the states in the execution trace by the regions they belong to. For how we have implemented this mapping from states in execution traces to regions see Section 4.5.

We are now finished with the first iteration of DASH on our example program. The next iteration starts like the first one. We still find the same abstract error trace $\tau = (2,3,4,5,6)$, which is passed to `GetCombinedConstraint`. However, now the frontier has moved forward to $(5,6)$ and the resulting constraint is $\phi := (x_1 = input_1) \wedge (x_1 > 20) \wedge (x_2 = x_1 - 20) \wedge (x_2 = 0)$. This constraint is unsatisfiable and return value from `Solve(`ϕ`)` will indicate this.

The constraint ϕ being unsatisfiable means that there can not exist a sequence of input values that would cause the program to follow τ over the frontier. DASH will exploit this information to refine the abstraction by *splitting* the frontier region R_{k-1} in the manner presented in Figure 3.8. Notice how after the split, the region $R_{k-1} \wedge \neg p$ is the one visited by the execution trace and the edge from it to the region R_k has been eliminated. To perform this split DASH needs a predicate p such that no state $s \in R_{k-1}$ reachable by executions following τ for which p does not hold has a transition $s \rightarrow s'$ such that $s' \in R_k$. One example of such a predicate is the *weakest precondition* [10], which is defined as the weakest predicate over the program's variables such that after executing a statement some postcondition holds. In this context the postcondition is that specified by the region R_k. While we will use weakest preconditions while going through the current example, the actual algorithm uses a slightly different type of

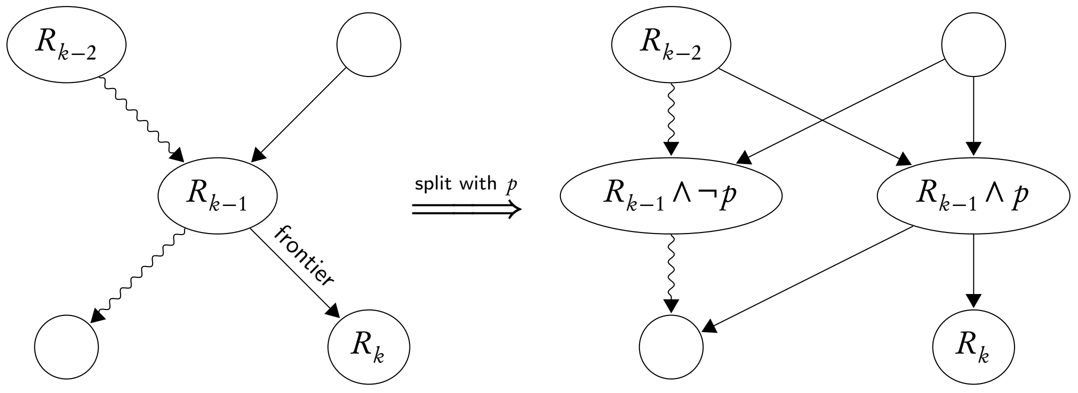


Figure 3.8.: The refinement performed by DASH

```
Output : $(\Sigma_{\simeq}, \rightarrow_{\simeq})$, such that the split has been performed
procedure SplitFrontier($\Sigma_{\simeq}$, $\rightarrow_{\simeq}$, $\tau$, $p$)
    $(R_{k-1}, R_k) := \text{Frontier}(\tau)$
    $\Sigma_{\simeq} := (\Sigma_{\simeq} \setminus \{R_{k-1}\}) \cup \{R_{k-1} \wedge p, R_{k-1} \wedge \neg p\}$
    $\rightarrow_{\simeq} := \rightarrow_{\simeq} \setminus \{(R, R') \in \rightarrow_{\simeq} \mid R = R_{k-1} \vee R' = R_{k-1}\}$
    $\rightarrow_{\simeq} := \rightarrow_{\simeq} \cup \{(R, R_{k-1} \wedge p) \mid R \in \text{Parents}(R_{k-1})\}$
                  $\cup \{(R, R_{k-1} \wedge \neg p) \mid R \in \text{Parents}(R_{k-1})\}$
                  $\cup \{(R_{k-1} \wedge p, R) \mid R \in \text{Children}(R_{k-1})\}$
                  $\cup \{(R_{k-1} \wedge \neg p, R) \mid R \in (\text{Children}(R_{k-1}) \setminus \{R_k\})\}$
    return $(\Sigma_{\simeq}, \rightarrow_{\simeq})$
end
```

Figure 3.9.: The procedure for splitting the frontier with a predicate

predicate which we will describe in Section 3.2. Also note that due to the slightly stronger predicate that would actually be used, the region $(R_{k-1} \wedge p)$ still has both outgoing edges present in Figure 3.8.

To produce the splitting predicate DASH calls the procedure `RefinePred`. For our example the statement associated with the frontier is `x == 0` and because the region across the frontier has no additional predicate, the weakest precondition p is simply $(x_{-1} = 0)$. The index is -1 because negative indices are used for symbolic values in the splitting predicates to differentiate them from ones in path constraints.

The predicate p is passed along to the procedure `SplitFrontier` shown in Figure 3.9. This procedure performs the splitting operation in Figure 3.8. The region R_{k-1} is removed from the abstraction and replaced by two new regions, $(R_{k-1} \wedge p)$ and $(R_{k-1} \wedge \neg p)$. All transitions for R_{k-1} that were in $\rightarrow_{\simeq}$ are replicated for the new regions except the transition $(R_{k-1} \wedge \neg p) \rightarrow_{\simeq} R_k$, which is the one we want to eliminate. The new abstraction and its transition relation, shown in Figure 3.6(c), are returned and the main DASH algorithm overwrites the previous versions with the refined ones. In the figure the predicate has been renamed to p_1. This concludes the second iteration of DASH.

On the third iteration the abstract error trace found in the abstraction (see Figure 3.6(c)) is $(2,3,4,5 \wedge p_1,6)$ and the frontier examined is $(4,5 \wedge p_1)$. The only input sequence found in C that will take the program to the frontier region 4 is still (38).

For this frontier the call to `GatherConstraints` in the `GetCombinedConstraint` procedure will also have to consider the predicate p_1 in the region over the frontier. The constraint returned will be $\phi := (x_1 = input_1) \wedge (x_1 > 20) \wedge (x_2 = x_1 - 20) \wedge (x_{-1} = x_2) \wedge (x_{-1} = 0)$, where the path constraint part is $(x_1 = input_1) \wedge (x_1 > 20) \wedge (x_2 = x_1 - 20)$ and the constraint from the region $5 \wedge p_1$ is $(x_{-1} = 0)$. The two parts have been "glued" together with the constraint $(x_{-1} = x_2)$, which links the symbolic value of `x` in p_1 with its last symbolic value in the path constraint part.

The constraint is effectively the same as on the previous iteration and a call to `Solve(`ϕ`)` will again return a value indicating it is unsatisfiable. The procedure `RefinePred` now has to create a weakest precondition for the statement `x = x - 20` and the postcondition $(x_{-1} = 0)$. The resulting predicate is $p_2 := (x_{-1} = x_{-2} - 20) \wedge (x_{-1} = 0)$. This predicate is used to split the frontier region 4 in the abstraction, the result of which is shown in Figure 3.6(d).

On the fourth iteration the abstract error trace in the current abstraction (see Figure 3.6(d)) is $(2,3,4 \wedge p_2,5 \wedge p_1,6)$ and its frontier is $(3,4 \wedge p_2)$. This time there are two input sequences in C that will take the program to the frontier, (38) and, from the initial test, (15). Assume we select the input sequence (15). The resulting constraint is $\phi := (x_1 = input_1) \wedge (x_1 > 20) \wedge (x_{-2} = x_1) \wedge (x_{-1} = x_{-2} - 20) \wedge (x_{-1} = 0)$, which is again unsatisfiable.

The procedure `RefinePred` will then be called to create a weakest precondition for the statement `x > 20` and the postcondition p_2. For these we get the predicate $p_3 := (x_{-2} > 20) \wedge p_2$, which is then used to split region 3. The resulting abstraction can be seen in Figure 3.6(e). Note that we have written predicate p_3 as a conjunction with the postcondition it was constructed for.

The final iteration proceeds largely as before. For the frontier $(2,3 \wedge p_3)$ the constraint $(x_1 = input_1) \wedge (x_{-2} = x_1) \wedge (x_{-2} > 20) \wedge (x_{-1} = x_{-2} - 20) \wedge (x_{-1} = 0)$ is created. It is again unsatisfiable. From the statement `x = input` and the postcondition p_3 we get the weakest precondition $p_4 := (x_{-2} = input_{-1}) \wedge p_3$.

The final version of the abstraction, which has been refined with p_4 is shown in Figure 3.6(f). The regions that have a path to the error region 6 have been boxed. Notice that the region containing the initial state (marked with the wedge above it) no longer has such a path. Because of this, at the beginning of the next iteration DASH will not find an abstract error trace. This proves that there is no set of inputs that would cause the program to encounter the error and DASH will return a pair of ("pass", $\Sigma_{\simeq}$).

3.2 Suitable Predicates

To do the refinement outlined in Figure 3.8 we need to have a predicate p such that having no edge from $(R_{k-1} \wedge \neg p)$ to R_k is sound, which means that the predicate must not be too strong. As a trivial example if we use $p = \bot$ as the predicate then all possible concrete traces would belong to $R_{k-1} \wedge \neg p$ and removing the edge to R_k would not be sound. On the other hand the predicate p must not be too weak for DASH to make progress. For example, using $p = \top$ as the predicate would cause all concrete traces to belong to $R_{k-1} \wedge p$ and the frontier for the next iteration to be effectively the same. To capture these requirements for predicates we use the following definition of a *suitable predicate* [4].

Definition 1 (Suitable predicate)**.** Let τ be an abstract error trace and let (R_{k-1}, R_k) be its frontier. A predicate p is said to be suitable with respect to τ only if all possible concrete states obtained by executing τ up to the frontier belong to the region $R_{k-1} \wedge \neg p$, and if there is no transition from any state in $R_{k-1} \wedge \neg p$ to a state in R_k.

Theorem 2. *A suitable predicate ensures that refinement performed by the* DASH *algorithm is sound.*

Proof. Let there be an abstract error trace $\tau = (R_0, R_1, \dots, R_n)$ and let (R_{k-1}, R_k) be its frontier. The refinement in Figure 3.8 splits the region R_{k-1} into two regions, $(R_{k-1} \wedge p)$ and $(R_{k-1} \wedge \neg p)$. All concrete states from R_{k-1} belong to one of these new regions: if a state that is in R_{k-1} does not belong to $R_{k-1} \wedge p$ then p is not true for that state and therefore the state belongs to $R_{k-1} \wedge \neg p$. Because no other regions are modified we can be sure that the refined abstraction contains all the concrete states from the original.

The only edge eliminated from these new regions is $(R_{k-1} \wedge \neg p, R_k)$. However, by Definition 1 there are no transitions from any concrete state in $R_{k-1} \wedge \neg p$ to a state in R_k and, therefore, all concrete traces from the original abstraction are still contained in the refined one. Because the refined abstraction contains all concrete states and traces from the original, the refinement is sound. □

To define progress we will first define a strict partial order $\sqsubset$ on abstract error traces. Given two abstract error traces $\tau = (R_0, R_1, \dots, R_n)$ and $\tau' = (T_0, T_1, \dots, T_n)$ of length n, we say that $\tau \sqsubset \tau'$ if one of the following conditions holds.

1. For all $i \in [0 \dots n]$ it holds that $R_i \supseteq T_i$ and there exists a $k \in [0 \dots n]$ such that $R_k \supset T_k$.
2. For the frontiers $(x-1, x) = \text{Frontier}(\tau)$ and $(y-1, y) = \text{Frontier}(\tau')$ it holds that $x < y$ and for all $i \in [0 \dots n]$ it holds that $R_i = T_i$.

Now $\tau \sqsubset \tau'$ if in τ' the frontier has moved forward or if states have been removed from at least one region. Note that two abstract error traces must at least visit the same program locations to be comparable under $\sqsubset$. With this relation we can define *progress*.

Definition 2 (Progress). Let $(\tau_0, \tau_1, \dots)$ be a sequence of abstract error traces examined by DASH. We say that DASH makes progress if there do not exist i and j such that $i < j$ and $\tau_j \sqsubset \tau_i$.

Theorem 3. *If suitable predicates are used to perform refinement, then the* DASH *algorithm makes progress.*

Proof. Assume that in the sequence of abstract error traces $(\tau_0, \tau_1, \dots)$ examined by DASH there exists τ_i and τ_j such that $i < j$ and $\tau_j \sqsubset \tau_i$. Because DASH never removes visited states from the forest F, the frontier can not move backwards and thus the second condition for $\tau_j \sqsubset \tau_i$ can not hold. According to the first condition there exists at equal indices a pair of regions, $R_k \in \tau_j$ and $T_k \in \tau_i$, such that $R_k \supset T_k$. However, all regions in τ_j are either the same as corresponding regions in τ_i or they are products of refinement with a suitable predicate. Because the new regions produced by refinement can never hold more states than the original, there can be no regions R_k and T_k. Therefore the original assumption is false. □

3.2.1 Suitable Predicates from Weakest Preconditions

Next we outline a method with which the procedure `RefinePred` can construct predicates that satisfy Definition 1. The construction is based on *weakest preconditions* [10]. We will explain this method with example statements from our program model. For a list of statement types and their semantics refer to Table 3.1.

Given a statement $op \in Stmts$ and a predicate ϕ (also called the postcondition), the weakest precondition $\mathrm{WP}(op, \phi)$ is the weakest predicate whose truth before *op*'s execution implies ϕ after *op* is executed. For example, consider the statement $op =$ "`x = x + 1`" and postcondition $\phi = (x_{-1} < 7)$. The weakest precondition for these is $\mathrm{WP}(op, \phi) = (x_{-1} = x_{-2} + 1) \wedge (x_{-1} < 7)$.

A weakest precondition p constructed for an abstract error trace τ and its frontier (R_{k-1}, R_k) is also a suitable predicate. Because DASH only splits the frontier when no inputs exist to force the execution across the frontier then no concrete state reachable by executing τ to the frontier belongs to $R_{k-1} \wedge p$. If there was such a state then the weakest precondition being true would imply that the execution would cross the frontier. Also, if there was a transition from any state in $R_{k-1} \wedge \neg p$ to a state in R_k then p would not be the weakest precondition because a precondition that also includes the concrete state in question would be weaker.

In the presence of memory and pointers constructing weakest preconditions may not be as straightforward as in the example above. For example, if we have a statement $o_1 =$ "`x = load p`" together with some postcondition ϕ we could have $\mathrm{WP}(o_1, \phi) = (x_{-1} = m) \wedge \phi$, where m is used to represent the memory pointed at by `p`. Now consider using this weakest precondition as the postcondition for another statement $o_2 =$ "`store 5 q`". Because the result of the store depends on whether `p` and `q` are equal the weakest precondition must handle both cases:

$$\begin{aligned}\mathrm{WP}(o_2, \mathrm{WP}(o_1, \phi)) =& ((p_{-1} = q_{-1}) \wedge ((m = 5) \wedge (x_{-1} = m) \wedge \phi)) \vee \\ =& ((p_{-1} \neq q_{-1}) \wedge ((x_{-1} = m) \wedge \phi))\end{aligned}$$

The number of disjuncts in the predicate scales exponentially with the number of memory locations referenced by the postcondition. In the general case if we have k memory locations then we must handle 2^k different aliasings. A solution to this problem is to use aliasing information gathered from a concrete test run. For this purpose we define the projection of a weakest precondition down to a specific aliasing constraint α as follows [4]:

$$\mathrm{WP}{\downarrow_\alpha}(op, \phi) = \alpha \wedge \mathrm{WP}(op, \phi)$$

This projected weakest precondition can be constructed to only consider one aliasing situation thus avoiding the exponential number of disjuncts. In the example above if the concrete aliasing situation was that q and p are equal, then we would have:

$$\mathrm{WP}{\downarrow_\alpha}(o_2, \mathrm{WP}{\downarrow_\alpha}(o_1, \phi)) = (p_{-1} = q_{-1}) \wedge ((m = 5) \wedge (x_{-1} = m) \wedge \phi)$$

Using this projected weakest precondition and adding a term to handle the remaining aliasings we get a predicate that can be used for refinement as the predicate returned by the procedure `RefinePred`:

$$\mathrm{WP}_\alpha(op, \phi) = \neg\alpha \vee \mathrm{WP}{\downarrow_\alpha}(op, \phi)$$

Consider a predicate p constructed with WP_α in the context of the splitting operation in Figure 3.8. The region $R_{k-1} \wedge \neg p$ contains only states where the aliasing constraint α is true. Other aliasing situations belong to the region $R_{k-1} \wedge p$ and the predicate p does not constrain their subsequent control flow.

Theorem 4. *The predicate* $\mathrm{WP}_\alpha(op, \phi)$ *is a suitable predicate when it is returned from the procedure* `RefinePred`.

Proof. To prove that a predicate $\mathrm{WP}_\alpha(op, R_k)$ that is returned by `RefinePred` is a suitable predicate (see Definition 1) we must show (1) that all possible concrete states obtained by executing the abstract error trace τ up to the frontier belong to the region $R_{k-1} \wedge \neg\mathrm{WP}_\alpha(op, R_k)$ and (2) that there is no transition from any state in $R_{k-1} \wedge \neg\mathrm{WP}_\alpha(op, R_k)$ to a state in R_k:

1. Assume that there is a concrete state c reachable by executing τ up to the frontier such that $c \notin (R_{k-1} \wedge \neg\mathrm{WP}_\alpha(op, R_k))$. Because the abstraction is sound the concrete state c must satisfy R_{k-1}. Because our program model prohibits pointer arithmetic then c must also satisfy α. From this it follows that c satisfies $\mathrm{WP}{\downarrow_\alpha}(op, R_k)$, because otherwise $c \in (R_{k-1} \wedge \neg\mathrm{WP}_\alpha(op, R_k))$ would hold. However, $\mathrm{WP}{\downarrow_\alpha}(op, R_k)$ ensures that after op is executed if α holds then also R_k holds. This is a contradiction because for `RefinePred` to be called the previous solver call must have found no inputs that would cause the program to cross the frontier. Therefore there can be no concrete state c such that $c \notin (R_{k-1} \wedge \neg\mathrm{WP}_\alpha(op, R_k))$.

2. Assume that there is a concrete state $c \in (R_{k-1} \wedge \neg \mathrm{WP}_\alpha(op, R_k))$ that has a transition to R_k. From $\neg \mathrm{WP}_\alpha(op, R_k)$ we get

$$\begin{aligned}
&\neg \mathrm{WP}_\alpha(op, R_k) \\
\Leftrightarrow\ &\alpha \wedge \neg \mathrm{WP}{\downarrow_\alpha}(op, R_k) \\
\Leftrightarrow\ &\alpha \wedge \neg(\alpha \wedge \mathrm{WP}(op, R_k)) \\
\Leftrightarrow\ &\alpha \wedge \neg \mathrm{WP}(op, R_k)
\end{aligned}$$

Now $\neg \mathrm{WP}(op, R_k)$ ensures that there can be no transition to R_k and the original assumption is false. Therefore there is no state with a transition from $(R_{k-1} \wedge \neg \mathrm{WP}_\alpha(op, R_k))$ to R_k

□

3.2.2 Suitable Predicate Construction Rules

We will now provide a detailed set of rules for constructing suitable predicates for the statements in our program model (see Table 3.1). The suitable predicates are constructed using temporary variables (instead of substitution) to represent assignments. For this purpose we associate each predicate with a variable version map V which maps each variable to an integer representing its current version. For example, if we have a variable x then its current version is $x_{V(x)}$. When variables are assigned to, the predicate is written to assign to the current version of the variable, after which the version number of the variable is decremented. This ensures that any further suitable predicates constructed with this predicate as their postcondition will refer to the version of the variable that exists before the assignment. Initially all variables are mapped to the version number -1. This way these negative variable version numbers do not get mixed up with the positive variable version numbers created during dynamic symbolic execution.

Consider the situation where we are constructing a suitable predicate for the statement `store 7 p` and a target predicate $(y_{-1} = m_1) \wedge (x_{-1} = m_2) \wedge (x_{-1} \neq y_{-1})$, where m_1 and m_2 represent memory locations. To construct the predicate we need to know which memory locations the `store` statement assigns to. Variables for memory locations are added when constructing suitable predicates for statements that read from memory. To construct suitable predicates for statements that write to memory the following information will suffice:

1. the pointer each memory location m_i appearing in the target predicate was read through, and
2. an equivalence relation for pointers at the point before the statement for which we are constructing the suitable predicate.

The first item can be addressed by associating with each suitable predicate a *mentions map* M, which is a mapping from versioned variables to variables representing memory locations. When we construct a new suitable predicate on top of an existing one the mentions map is inherited with the appropriate modifications. For `load` statements a new mapping from the current version of the pointer to a new variable for the memory location is introduced (or an existing mapping may be reused). Conversely, `store` statements remove mappings to the variables representing memory locations that the `store` assigns to.

The equivalence relation for pointers is obtained from a concrete execution that has visited the frontier region. During the test run we record the values of all assignments to pointers. Using the recorded information we then construct a map A from all pointers to their current values at the frontier. We can check if two pointers are equivalent by checking whether they map to the same value.

Now each suitable predicate constructed is associated with a variable version map V and a mentions map M. Given a statement op and the assignment A of pointers at the frontier, the suitable predicate $p' = \mathrm{WP}_\alpha(op, p)$ can be constructed by following the rules in Table 3.10. Each rule describes how a new suitable predicate p' is constructed. The rules also produce new versions of the variable version and the mentions maps as V' and M'. These are stored with p' for constructing further suitable predicates where p' is the postcondition. For updating the maps the rules employ the notation introduced in Section 3.1. In particular note the previously unused notation $M[x \not\mapsto]$ for removing the entry for x from the map M.

The rule for statements of the form `x = y <op> z` is straightforward: a new constraint encoding the assignment is added to p' and the version number of x is decremented in V'. No special handling for pointers is needed because we assume our programs do not contain pointer arithmetic.

Statements of the form `x = input` also encode the assignment in p' and decrement the version number of x. The input is implemented here as a variable *input*, the current version of which is assigned to x. Because each input statement should use a new input we also decrement the version number of *input* in V'.

For simple assignments the rule is similar to the one for assignment of binary expressions in how p' and V' are updated. However, now the assignment may be from a pointer to another, which must be handled by the rule. To explain this handling we will first describe how the statements of the form `x = load a` are handled. A `load` statement crates a new symbolic variable m to represent the memory location from which the read happens. This variable m is stored in the mentions map M' so that the current version of the pointer $a_{V(a)}$ maps to m. If M already contains an entry for $a_{V(a)}$ then the variable representing the memory is reused as m and the mentions map is not modified. The mapping from $a_{V(a)}$ to m is added so that we can resolve for any subsequent `store` statements which memory locations the added constraints should refer to. Now the handling of a simple assignment of the form `x = y`, where x and y are pointers, can be understood. If a `load` has mentioned the current version of the pointer x, then the assigned value $y_{V(y)}$ becomes the pointer value through which the memory location is mentioned. To indicate this the mapping for $x_{V(x)}$ is removed from M' and a new mapping from $y_{V(y)}$ to $M(x_{V(x)})$ is added.

As explained above, for statements of the form `x = load a` a symbolic variable m is selected to represent the memory location. In the new constraint p' this variable m

Statement	Actions
`x = y <op> z`	$p' = p \land (x_{V(x)} = y_{V(y)}$ `<op>` $z_{V(z)})$ $V' = V[x \mapsto V(x) - 1]$
`x = input`	$p' = p \land (x_{V(x)} = input_{V(input)})$ $V' = V[x \mapsto V(x) - 1][input \mapsto V(input) - 1]$
`x = y`	$p' = p \land (x_{V(x)} = y_{V(y)})$ $V' = V[x \mapsto V(x) - 1]$ If x is a pointer and M contains $x_{V(x)}$ then also do: $M' = M[x_{V(x)} \not\mapsto][y_{V(y)} \mapsto M(x_{V(x)})]$
`x = load a`	Let $m = \begin{cases} M(a_{V(a)}), \text{ if } M \text{ contains } a_{V(a)} \\ \text{a new unique variable otherwise} \end{cases}$ $p' = p \land (x_{V(x)} = m)$ $M' = M[a_{V(a)} \mapsto m]$ $V' = V[x \mapsto V(x) - 1]$
`store x a`	$\alpha = \bigwedge\{a_{V(a)} = b \mid b \in \text{keys}(M), A(a_{V(a)}) = A(b)\} \land$ $\bigwedge\{a_{V(a)} \neq b \mid b \in \text{keys}(M), A(a_{V(a)}) \neq A(b)\}$ $p' = \Big(p \land \bigwedge\{m = x_{V(x)} \mid m = M(b) \text{ where } b \in \text{keys}(M)$ $\text{and } A(a_{V(a)}) = A(b)\}\Big) \lor \neg\alpha$ $M' = M$ except all keys b such that $A(b) = A(a_{V(a)})$ are removed.
`a = allocate`	Let $s =$ an integer not used in any previous `allocate` $p' = p \land (a_{V(a)} = s)$ $M' = M[a_{V(a)} \not\mapsto]$ $V' = V[a \mapsto V(a) - 1]$
`(x <comp> y)`	$p' = p \land (x_{V(x)}$ `<comp>` $y_{V(y)})$
`!(x <comp> y)`	$p' = p \land \neg(x_{V(x)}$ `<comp>` $y_{V(y)})$

Table 3.10.: Rules for suitable predicate construction

is used for the assignment to $x_{V(x)}$ and the version number of x is decremented in V' due to the modification from the assignment. The mentions map M' is also modified to record that m is mentioned through $a_{V(a)}$.

The rule for statements of the form `store` x a combines information from the mentions map M and the pointer assignments A at the point before the statement. First an aliasing constraint α is constructed by examining all pointer values currently in M. For each mentioning pointer value b we create a constraint of the form $(a_{V(a)} = b)$ if the pointers have the same value, i.e., $A(a_{V(a)})$ is equal to $A(b)$. If the pointers are not equal we instead create the constraint $(a_{V(a)} \neq b)$. The aliasing constraint α is a conjunction of all the created equalities and inequalities. Using α the constraint p' is then created. It is in this constraint that the form of the suitable predicate WP_α described in Section 3.2.1 can be seen. The created constraint is true if either α does not hold or if the weakest precondition from the `store` statement in the current aliasing situation holds. The weakest precondition part of the constraint is constructed by examining all mentioned memory locations m in M and creating a constraint of the form $(m = x_{V(x)})$ if the pointer value b through which m is mentioned is equal to $a_{V(a)}$, i.e., $A(a_{V(a)})$ is equal to $A(b)$. Now, the weakest precondition part is a conjunction of p and any new constraints created for mentioned memory locations. Finally, all mentioned memory locations that were assigned to are removed from the memory map, because any `store` statement executed before this one should not have any effect on these.

The rule for statements of the form a `= allocate` selects a new integer s that has not been used by any previous `allocate` statement. The modifications to p' and V' from assigning s to a are the same as in the other assignment statements described above. However, an allocation always assigns to a pointer and as such the mentions map might also be modified. If $a_{V(a)}$ is in the mentions map, then the mapping can be removed because this `allocate` statement is the first point at which the value s is seen and as such can not appear in any `store` statement that might be handled later (i.e. earlier in the execution order). Note that if $a_{V(a)}$ is actually found in the mentions map then the program under test reads from uninitialized memory. This could be handled either as an error or as an additional input to the program.

Finally, the rules for the guard statements simply add the appropriate guard constraint to p'. These statements do not modify variable versions or the mentions map as they represent branching conditions and have no effect on the state of variables or memory.

A situation not handled by the rules is when pointers are stored to or loaded from memory. Proper handling of these situations would involve also having the mentions map support entries of the form $[m_1 \mapsto m_2]$, where a memory location is mentioned through another memory location. When memory locations can appear as keys in the mentions map the assignment A of pointers also needs to track memory locations which contain a pointer. We have left support for loading and storing pointers for further work.

In the DASH algorithm the constructed suitable predicates are combined with path constraints generated by dynamic symbolic execution. As was already mentioned in the running example in Section 3.1, if we have a path constraint ϕ_{path} and a predicate p for a target state then p can be added to the end of the path constraint with the help of some additional "glue" constraints. The additional constraints must ensure the following:

1. The earliest versions of variables in p must be equivalent to their latest versions found in the path constraint. If they are not found in the path constraint (i.e. they are not symbolic) then they must be assigned concrete values obtained from the test execution at the frontier.

2. Variables for memory locations that are still in the mentions map M must be equivalent to their appropriate symbolic or concrete values.

Because concrete values from the execution are needed and storing the whole history of concrete values for all test executions could be memory intensive we have in our implementation chosen to re-execute the program to the frontier to obtain the necessary values. In the pseudocode of the DASH algorithm this re-execution is part of the `GatherConstraints` procedure.

3.2.3 Block Level Suitable Predicates

As already mentioned in Section 2.3, programs in the LLVM IR are structured into basic blocks and control flow is transferred always to the beginning of a basic block. Non-terminator instructions do not transfer control flow (i.e. the next instruction listed in the block is the next to be executed), with the exception of the `call` instruction. In this work we omit support for programs with procedures and with this restriction we know that when a basic block is executed the instructions listed in it will be executed one by one until the terminator instruction is reached, after which the control flow may jump to another basic block. Therefore, we can treat reachability in the program on the level of basic blocks. Nodes in DASH's abstraction will represent basic blocks instead of single instructions. All suitable predicate construction and splitting operations will be done on basic blocks.

To construct a suitable predicate for a basic block we can recursively apply the construction from the previous section. Given a basic block $b = (op_1, op_2, \ldots, op_n)$ and a postcondition ϕ the suitable predicate for ϕ when b is executed is:

$$\mathrm{WP}_\alpha(b,\phi) = \mathrm{WP}_\alpha(op_1, \mathrm{WP}_\alpha(op_2, \mathrm{WP}_\alpha(\ldots \mathrm{WP}_\alpha(op_n, \phi)\ldots)))$$

A similar approach to combining sequences of operations in the control flow graph has been presented by Beyer et al. [5], who also present a method for combining alternate paths in a control flow graph.

3.3 Handling Phi Instructions

Programs in the LLVM IR (see Section 2.3) may contain `phi` instructions, which select a value to assign based on which control flow edge the basic block containing the phi instruction is entered. In LLVM phi instructions are always before any other types of instructions in a basic block (there may be multiple phi instructions). Each phi instruction in a basic block contains a list of *source-value pairs*, which specify a corresponding value (i.e. a variable or a constant) for every predecessor basic block. The following is a phi instruction presented similarly to statements in our program model:

$$r \texttt{ = phi [}b_1 \mapsto v_1\texttt{], [}b_2 \mapsto v_2\texttt{], ..., [}b_n \mapsto v_n\texttt{]}$$

The semantics are such that if the previous basic block was b_i where $i \in [1 \ldots n]$ then r gets the value v_i.

In dynamic symbolic execution whenever a phi is encountered the source basic block is always known and therefore phi instructions simply reduce to simple assignments. For suitable predicate construction this is not the case. As a suitable predicate can be used in multiple contexts we can not simplify phi instructions to simple assignments at construction time. However, the simplification can be done if we defer the construction as follows.

Assume we are constructing a suitable predicate $\mathrm{WP}_\alpha(op_{pred}, \mathrm{WP}_\alpha(op_{phi}, \phi))$, where op_{pred} is an instruction belonging to the previous basic block and op_{phi} is a phi. Because op_{pred} gives us the previous basic block we can now simplify the phi instruction to a simple assignment. This observation can be extended to situations with multiple sequential phi instructions. Now we can construct suitable predicates whenever the first (in terms of the program's execution order) instruction is not a phi. However, if we have a suitable predicate of the form $\mathrm{WP}_\alpha(op_{phi}, \phi))$ we are still stuck. To handle these we can observe that in DASH suitable predicates are always solved in the context of a specific execution path. Therefore, when we have a suitable predicate of the form $\mathrm{WP}_\alpha(op_{phi}, \phi))$ we can use this path information to finish the simplification for the solver call in question.

In conclusion, we can now always simplify phi instructions before we need to solve them due to the following: (1) phi instructions used to construct suitable predicates that are used as postconditions for other suitable predicates can be immediately simplified, and (2) the phi instruction in the outermost WP_α application can be simplified when combining with the path constraint.

There is one additional complication to constructing suitable predicates for phi instructions: in LLVM all phi instructions at the beginning of a basic block are executed atomically [40]. In other words a value assigned by a phi instruction is visible only after all phi instructions in a basic block have been executed. These semantics must be reflected in the suitable predicate construction. Consider a phi instruction that has a variable x in one of its source-value pairs. Assuming x_{-k} is the current version of x before the constraints for the phi instructions have been created, then the correct version to use in the constraint is as follows:

- x_{-k}, if x is not assigned to by another phi in the same basic block.
- x_{-k-1}, if x is assigned to by another phi in the same basic block. For the phi instruction that assigns to x the constraint should be written so that it assigns to x_{-k}.

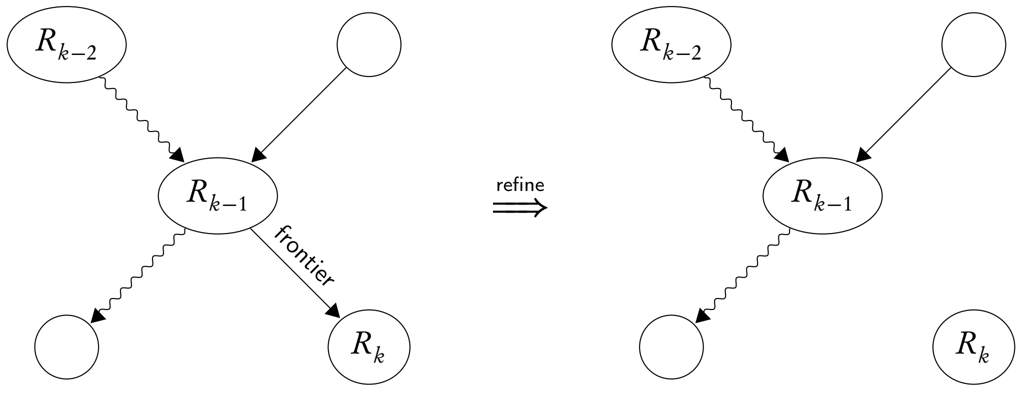


Figure 3.11.: Refinement when R_k is unsatisfiable

3.4 Exploiting Unsatisfiable Regions

In the algorithm presented so far the only way to eliminate all paths to an error region is to continue refining the abstraction until the initial region is split. However, propagating the splitting all the way to the initial region can be expensive as it can involve many steps during which the constraints grow as more suitable predicates are constructed. Moreover, diamond structures in the CFG on the path from the initial region to the current frontier will during the splitting duplicate the frontier even if the code in the diamond structure is irrelevant to the verification task at hand. The duplicated frontiers would all have to be propagated to the initial region to eliminate them. The aggregate effect of this phenomenon is a combinatorial explosion as all reverse paths back from the frontier are explored through the splitting process. For an example of this see Chapter 5.

To alleviate this problem we have modified how DASH attempts to extend and refine frontiers. In Figure 3.3 the constraint passed to the solver is of the form $\phi_{path} \wedge \phi_{glue} \wedge R_k$. Observe that if the constraint R_k of the target region is unsatisfiable by itself, then no matter what the path constraint ϕ_{path} is the conjunction will still be unsatisfiable. Therefore, before attempting to generate a test to cross the frontier we make an additional solver call `Solve(`R_k`).` If the constraint R_k is unsatisfiable, then we can apply the stronger refinement presented in Figure 3.11 where we simply remove the frontier edge (R_{k-1}, R_k). Note that because of the way `phi` instructions are handled (see Section 3.3) the suitable predicates created for regions may be dependent on the edge a region is entered. Due to this we can only remove the frontier edge and not the whole target region.

Chapter 4

Implementation

In this chapter we will describe how we have implemented the DASH algorithm as a modification to the Lime Concolic Tester (LCT), which is an open source dynamic symbolic execution tool for C and Java programs. Our tool LCT-D extends the LLVM based C support in LCT.

4.1 Lime Concolic Tester

Dynamic symbolic execution is described in Section 2.1 and this section will concentrate on the details of LCT necessary to understand our implementation. For a more in-depth description of LCT see [21–23].

LCT uses a client-server model to distribute test execution and constraint solving work. A *testing server* keeps track of the execution tree and selects which paths are to be explored next. When a client connects to the server a new path to explore is selected and the path constraint for the selected path is sent to the client. The client calls an SMT solver with the path constraint and if the constraint was satisfiable the client executes the program with the obtained inputs. During execution the client sends details of each instruction it executes to allow the server to construct path constraints for further tests. The clients lose all state after each execution and all persistent state is stored on the testing server.

In LCT the client is implemented as an instrumented version of the program under test. For C programs the instrumentation is implemented as an LLVM transform pass, which adds the following calls to functions in LCT's runtime library: (1) at the program's entry point a call to an initialization function and (2) for each instruction in the program a call to a symbolic counterpart function in LCT's runtime library. LCT also assumes that inputs to the program have been marked with calls to functions of the form `lct_get_*()`, where the wildcard is the type of the input. The initialization function connects to the testing server and retrieves a path constraint. Next it calls an SMT solver (currently Boolector [6]) to find values for the program's inputs that would satisfy the constraint. If the constraint was unsatisfiable the client reports to the server and receives a new path constraint. When a satisfiable constraint is found the execution is allowed to continue with the solved inputs. The symbolic counterpart functions send the details of the operations (e.g. what variables they operate on and where the results are stored) to the testing server. As the client reports the operations it executes, the server keeps the execution tree up to date by storing the corresponding symbolic constraints for each operation as well as adding new branches as necessary.

The server is implemented as a Java application. The server is largely language agnostic as language specific issues are handled in the client implementations.

4.2 Adapting Dash to the Distributed Model

The distributed model of LCT poses two restrictions on implementing DASH: (1) the client loses all state between executions and therefore all persistent state must reside on the server and (2) concrete values encountered during execution are only available on the client unless explicitly sent to the server. The abstraction, solved input values and traces of test execution are needed across iterations of DASH and are therefore stored on the server. Because the server has the abstraction it also runs the main loop of the DASH algorithm. On the other hand test execution, constraint gathering and SMT solver calls are performed by the client. Figure 4.1 provides an overview of how the functionality is divided between the server and the client.

Comparing the flowchart of the client-server version of DASH to the original flowchart in Figure 3.2 we can see that when the server needs to execute a test it sends the inputs to a client, which in turn executes the instrumented program with the given inputs. During the execution the client sends back the input values used, which include the new random inputs from the previously unexplored part of the path tested. Concrete values of pointers are also sent to the server to allow the aliasings for the test execution to be resolved. The client also informs the server which LLVM basic blocks it executes. Using this path information the server checks that the client follows the expected path (for the explored part) and maps the trace of the execution back to the regions in the abstraction (for the unexplored part). The process of mapping execution traces back to regions in the abstraction is discussed in detail in Section 4.5.

The server also needs the client to solve the constraints for checking whether frontiers can be extended. This corresponds to calling the `GatherConstraints` and `Solve` procedures in Figures 3.7 and 3.3, respectively. In LCT's original implementation of dynamic symbolic execution the server knows the path constraints for each past test execution, which in DASH would allow the `GatherConstraints` procedure to be implemented on the server. However, in Section 3.2.2 we noted that the suitable predicates may have free variables that are not symbolic values in the path constraint and therefore must be constrained to the concrete value at the frontier. Instead of also recording all concrete values during test executions we have chosen to re-execute the program to the frontier to recover them. This change also removes the need to store the path constraints on the server as the path constraint may also be recovered during the re-execution.

When attempting to extend a frontier the server sends to the client the inputs that will take it to the frontier, the branching decision that will take it to the target region and the predicate at the target region. The server also sends a list of free

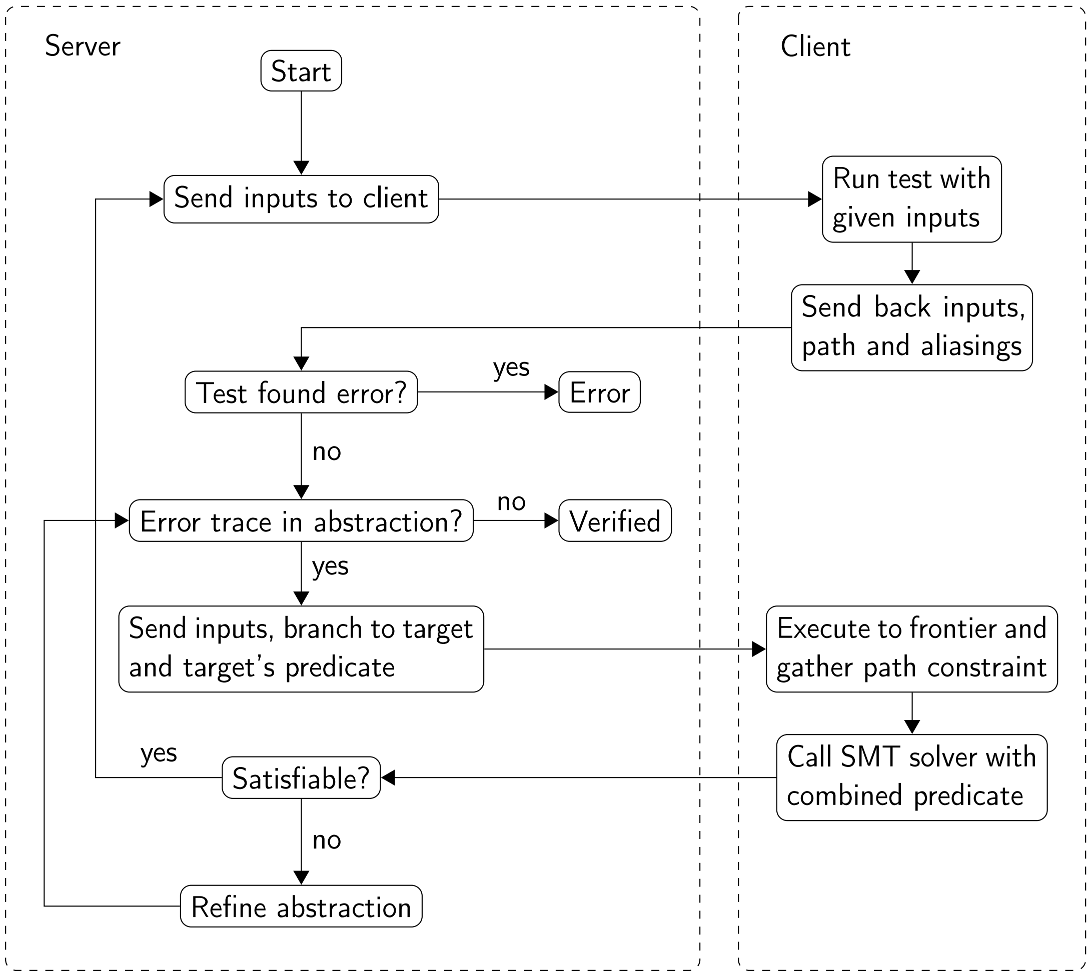

Server
Start
Send inputs to client
Test found error?
yes
Error
no
Error trace in abstraction?
no
Verified
yes
Send inputs, branch to target and target's predicate
yes
Satisfiable?
no
Refine abstraction
Client
Run test with given inputs
Send back inputs, path and aliasings
Execute to frontier and gather path constraint
Call SMT solver with combined predicate

Figure 4.1.: Flowchart of DASH adapted to the client-server model of LCT

variables (including mentioned memory locations), which is used by the client to combine the path constraint with the target region's predicate. The client then executes the program under test up to the frontier during which it constructs the path constraint and records the concrete or symbolic values for the free variables in the target region's predicate. Once the frontier is reached the client combines the path constraint (including constraints to take the execution over the frontier) with the target region's predicate. Then the client calls an SMT solver with the combined constraint. The client informs the server whether the constraint is satisfiable and also sends the solved input values, if any, to the server to be used in a subsequent test execution. See Section 4.4 for more details on how the combined constraint is constructed.

LCT allows for constraint solving to be distributed among multiple concurrent clients. However, our DASH implementation can not handle concurrent clients and such an extension to the algorithm is left for future work.

4.3 Representing the Abstraction

To initialize the abstraction the server needs the control flow graph of the program under test. Additionally details of the program's instructions are needed on the server for the suitable predicate construction. We record the required information during the instrumentation of the program under test. When the instrumenter, which in LCT is implemented as an LLVM transform pass, processes the program it also outputs a *block graph dump* (BGD) file. An example of the format used for the BGD file can be seen in Figure 4.3, which has been generated from the program in Figure 4.2. The BGD file includes an entry for each basic block, each of which list the instructions contained in it. For the supported arithmetic and memory operations the instructions written to the BGD file closely match those in the LLVM assembly language. Although programs with arbitrary calls are not currently supported, some `call` instructions have specialized handling:

- Calls to the input functions of the form `lct_get_*()` in LCT's runtime library are written to the BGD file with the keyword `input`.
- Calls to the `__assert_fail` function, which is used to implement the `assert` macro, are written with the keyword `assertfail`.

While the format is somewhat human readable it is designed to be so as a debugging aid and is mainly intended for machine consumption. As such we will not devote space to explaining its details. A grammar for the format is available in Appendix A.

Upon server startup the BGD file is used to initialize the abstraction. Each block corresponds to one region in the abstraction. In the BGD file the blocks are associated with numeric identifiers which are used in the terminators of the blocks. If a block's terminator mentions another block then the initial abstraction will contain a control flow edge between the two blocks.

Initially there is a one-to-one correspondence between basic blocks and regions and the abstraction is simply a graph connected by control flow edges. However, when

```c
#include <lct.h>
#include <assert.h>
int x;
int main() {
    x = lct_get_int();
    char y = x;
    while (y != 0) {
        assert(x != y);
        ++y;
    }
    return 0;
}
```

Figure 4.2.: An example program for the block graph dump format

DASH begins performing refinement the regions will be split. To keep track of the regions and the suitable predicates they have been split with we store for each basic block a tree of regions. Whenever a region is split the predicate used for the refinement is stored in the tree's node that represents the region to be split and two new child nodes are added to represent the region where the predicate is true and the one where it is false. The control flow edges are moved from the split region to the new child regions (see Figure 3.8). In other words, the structure of our abstraction is a forest of regions, in which each tree corresponds to one basic block and the leaves of the trees are connected by control flow edges to form the current version of the abstraction. The tree structure used allows us to easily determine how any two regions seen by DASH at any point in the algorithm are related to each other, which will be used in Section 4.5.

4.4 Solving Constraints on the Client

In this section we will go over some of the details involved in solving the combined constraints on the client. As was already explained in Section 4.2, when trying to extend a frontier the server will send to the client:

- A set of inputs from a previous test execution that will take the execution to the frontier.
- The index of the basic block at the frontier.
- A branching decision at the frontier that takes the client to the basic block of the target region.
- The predicate of the target region, i.e., a conjunction of zero or more suitable predicates.

```
block %39782752 {
  eq i32 %39873888 input
  store %39873888 @39819272
  eq i32 %39785192 load @39819272
  eq i8 %39785320 trunc %39785192 8
  goto %39782848
}

block %39782848 {
  eq i8 %39812448 phi %39785320 %39782752 %39787728 %39810656
  eq i32 %39812744 sext %39812448 32
  eq i1 %39786256 ne %39812744 0
  br %39786256 %39810272 %39810752
}

block %39810272 {
  eq i32 %39786552 load @39819272
  eq i32 %39786680 sext %39812448 32
  eq i1 %39786832 ne %39786552 %39786680
  br %39786832 %39810656 %39810464
}

block %39810464 {
  assertfail
  unreachable
}

block %39810656 {
  eq i8 %39787728 add %39812448 1
  goto %39782848
}

block %39810752 {
  ret 0
}
```

Figure 4.3.: The block graph dump for the example program in Figure 4.2

To generate the path constraint part of the combined constraint the program under test is executed with the inputs from the server. During execution a constraint for the path followed is collected by the instrumentation added to the program. When the execution has reached the end of the basic block at the frontier the branching decision received from the server is used to add a constraint that corresponds to the execution entering the basic block of the target region. If the terminator at the frontier is a `goto` instruction then no additional constraint needs to be added.

Now that the client has a path constraint for the path to the target region it needs to combine this constraint with the constraint for the target region. To do this the free variables in the target region's constraint need to be constrained to the values at the end of the frontier block. Two kinds of free variables may be present in the constraint: (1) variables for current versions of the program's variables and (2) variables for memory locations that are still in one of the mentions sets of the suitable predicates comprising the target region's constraint.

To handle free variables of the first type, while the program under test is being executed to the frontier a map from the variables in the program to their current symbolic or concrete values is maintained (and also used in gathering the path constraint). When the end of the basic block at the frontier is reached this map will contain the necessary values. For each free variable we add a new constraint of the form $(x_{-k} = v)$, where x_{-k} is the free variable from the target region's constraint and v is the appropriate symbolic value (if available) or concrete value (otherwise).

To create constraints for the free variables of memory locations we use a memory map, which maps memory addresses to symbolic or concrete values. However, the addresses of the memory locations are not known by the server and memory locations are instead identified by what pointers are used in the `load` instruction that would read the memory location, i.e., through what pointer they are mentioned. See Section 3.2.2 for details on how these mentions are maintained. To resolve the appropriate values for the free variables the client also tracks the addresses that are assigned to these mentioning pointers. Now when the end of the basic block at the frontier is reached the values can be resolved by using the recorded addresses of the mentioning pointers to look up the concrete or symbolic values stored in the memory map. With these values we add new constraints of the form $(m_i = v)$ just as we did for the free variables of the first type.

Now the combined constraint is a conjunction of the path constraint, the constraints generated for the free variables and the predicate of the target region. The client may now solve the combined constraint and send the results back to the server.

4.5 Mapping Traces to Regions

In the pseudocode of the `GetCombinedConstraint` procedure in Figure 3.7 the process of selecting a test that visits the frontier is presented as simply choosing a trace that intersects the frontier region. However, in an actual implementation of the algorithm it is not immediately clear what is the best way to decide to which regions the states of a concrete trace belong to. When a test execution enters a program location which

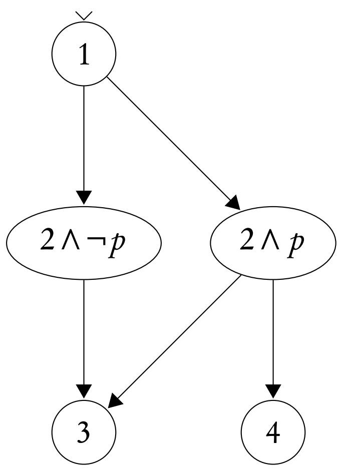


Figure 4.4.: Example abstraction for mapping traces

due to splitting corresponds to multiple regions, we have to have a way to evaluate the predicates to decide which region the concrete state belongs to. A trivial way of doing this is to evaluate the relevant suitable predicates at each step of a test execution. In our implementation this would have meant the client would request predicates from the server during execution. However, it turns out that as long as (1) we know the region of the last state of an concrete trace and (2) we can evaluate the pointer constraints in each suitable predicate, then we can also resolve the regions of the previous states. For example, consider the abstraction shown in Figure 4.4 and assume we wish to determine which regions a test with the trace $(1, 2, 3)$ belongs to. While regions 1 and 3 are clear, a selection must be made between $(2 \wedge p)$ and $(2 \wedge \neg p)$. Recall that p is a weakest precondition for the execution proceeding to region 4 projected down to a set of pointer constraints. If the test satisfies the pointer constraints then p must be false because otherwise the execution would have proceeded to region 4. With this we can evaluate p and make the selection between $(2 \wedge p)$ and $(2 \wedge \neg p)$. We will now present this selection process more formally. Assume we have the following:

- t_1 and t_2 as adjacent states in a concrete execution trace,
- R' as the abstract region t_2 belongs to and
- A as the assignment of pointer values at t_1.

We wish to determine the region that t_1 belongs to. If the region that was originally created for the program location of t_1 has not been split, then we know that t_1 belongs to that. Therefore we focus on the case that the region has been split. Now we have a set of candidate regions $R_0, R_1, \ldots, R_n$. Each region is associated with a conjunction of suitable predicates or their negations. Each predicate is of the form:

$$p = \neg\alpha \vee \mathrm{WP}{\downarrow_\alpha}(op, R_{target})$$

We can evaluate α using the pointer assignments in A. If α was false then we know that p is true and we are done. Otherwise, now that we know α is true then for $\mathrm{WP}\!\downarrow_{\alpha}(op, R_{target})$ to be true the state t_2 must belong to the region R_{target}. More succinctly p is true if and only if the following is true:

$$\neg\alpha(A) \vee (t_2 \in R_{target})$$

To evaluate $t_2 \in R_{target}$ we recognize that the region R' which t_2 belongs to and the region R_{target} are related in one the following ways:

- The regions correspond to different program locations and thus $R' \cap R_{target} = \emptyset$.
- $R' = R_{target}$
- Region R' is a result of splitting (or multiple splits of) R_{target} and thus $R' \subseteq R_{target}$.
- The regions are on different sides of a split and thus $R' \cap R_{target} = \emptyset$.

Note that R_{target} can not be a result of splitting R' because R' is a region currently in the abstraction and therefore can not have been split. So now it holds that either $R' \subseteq R_{target}$ or $R' \cap R_{target} = \emptyset$. Both can not be true because we know that R' is not empty. If $R' \subseteq R_{target}$ then because $t_2 \in R'$ we also have $t_2 \in R_{target}$. Alternatively if $R' \cap R_{target} = \emptyset$ then we know that $t_2 \notin R_{target}$.

Using the procedure outlined above we can evaluate any suitable predicate associated with the candidate regions $R_0, R_1, \ldots, R_n$ in the state t_1 and, therefore, we can decide which region t_1 belongs to. If we have the whole concrete trace $T = (s_0, \ldots, s_{k-1}, s_k)$, know the pointer assignments for all states in the trace and know the region for s_k, we can now repeat this procedure to resolve s_{k-1} through s_0.

To see how the method described above works in practice consider the example abstraction in Figure 4.4. Assume we have some execution trace that visits the program location 2 and that we wish to know whether the state belongs to $(2 \wedge p)$ or $(2 \wedge \neg p)$. First, we evaluate the aliasing constraint α from p using the pointer assignments at the program location 2. If α is false then we know that p is true and the correct region must be $(2 \wedge p)$. However, if α is true we must examine the subsequent control flow of the execution trace. For p to be true when α is true, the next program location must be 4. Since one of our assumptions was that the exact region of the next state in the execution trace is known, we have enough information to evaluate p.

In our implementation the regions in the abstraction are stored as a forest, where each tree represents the regions for a single program location. The leaves in the trees are the regions currently in the abstraction while internal nodes represent regions which have been split. In the forest $R' \subseteq R_{target}$ holds if and only if the node for R' is in the subtree rooted at R_{target}. Therefore, with the regions stored as a forest we can check how any two regions are related by checking whether one is an ancestor of the other.

Our approach differs from the one taken in the YOGI tool [32], which stores the whole concrete state at each program location. We only store the concrete values for pointer variables and avoid storing other concrete state by exploiting our knowledge of the execution's subsequent control flow. Similarly to the YOGI tool, our tool LCT-D only stores the changed pointers from one concrete state to another.

Chapter 5

Evaluation

In this chapter we present results from verifying a set of programs with our tool LCT-D, which implements the DASH algorithm. We selected the programs used for the experiments from the set of benchmarks for the 2013 Competition on Software Verification (SV-COMP). We excluded programs that allocate memory from the heap (e.g. with `malloc()`). Although adding support for this in LCT-D would be easy, heap allocated memory is often used together with pointer arithmetic (e.g. array accesses), which our tool does not support. We also excluded programs that use structs due to struct member accesses compiling down to the `getelementptr` instruction, which we do not support. Finally, as our tool does not support procedure calls, we had to exclude some programs where all procedures could not be inlined.

One further limitation we encountered was that some of the benchmarks were such that all executions were infinite. These could be handled by implementing a bound on the depth of the test executions. However, the method by which we map test executions back to the abstraction (described in Section 4.5) requires that the region of the last state in the execution is known. This is not necessarily the case when an execution is stopped before termination. Instead of extending LCT-D to support bounding the execution depth we chose to modify the benchmarks. For programs that consist of some initialization followed by an infinite loop with no code after, it is safe (i.e. reachability of error states is not altered) to replace the loop with one where a non-deterministic choice to continue is made. So in programs that have a top-level `while(1)` loop we would replace the loop with `while(lct_get_bool())`.

The programs and the version of LCT-D used in the following experiments can be downloaded from: `http://users.ics.aalto.fi/osaariki/lctd-msc/`

5.1 Experiment Setup

We ran all experiments on an Intel Core i5-2500 CPU @ 3.30 GHz with 8 GB of memory. Both the client and server ran on the same computer and as such their communication was over the TCP loopback interface.

The SV-COMP benchmarks have varying conventions for indicating inputs and errors. We took the following steps to prepare the benchmark programs for verification:

- We identified the inputs variables and modified the program to initialize them with the `lct_get_*()` functions from our tool's runtime library. Some inputs were already marked with a call to `__VERIFIER_nondet_int()` or similar, while others were simply uninitialized variables.
- We replaced errors marked in the program with calls to the `assert` macro from the C standard library.
- To handle programs with multiple functions we added the `inline` keyword and `always_inline` attribute to all functions apart from `main`.

We then compiled the prepared programs into LLVM IR, producing `.bc` files. Next we ran an optimization pass with the switches `-always-inline`, `-mem2reg` and `-lowerswitch`. Then we instrumented the programs with our instrumentation pass, which is implemented as a transformation pass that can be loaded into the LLVM `opt` tool. This transformation produces a `.bc` file containing the instrumented program as well as the `.bgd` file for initializing the abstraction on the server. Finally we compiled the instrumented `.bc` files into executables.

To run each test we started the server with the `.bgd` file of the program to verify. Once the server had started up we repeatedly ran the instrumented executable until the server reported the program to be safe or unsafe. The server then reported the following statistics:

- **Result**: Whether the program was SAFE or UNSAFE.
- **Time**: The time from the first client connection to either an error being reported or refinement of the abstraction removing all counterexamples.
- **Test runs**: The number of tests run including the initial arbitrary execution.
- **Solve runs**: The number of times DASH tried to extend a frontier.
- **SMT solver calls**: The number of calls made to the SMT solver. Each solve run makes one call if the target region's constraint is unsatisfiable and if not a second call is made for extending the frontier.
- **Unsat targets**: The number of unsatisfiable constraints encountered in target regions.

We report the results of one verification run for each program as the time used did not vary significantly. However, in our tests we used a fixed random seed for generating the new inputs encountered during test runs. The random inputs generated affect which parts of the program are explored first and therefore have an effect on the time and iterations required for the verification task. In a comparative benchmark between verification tools this effect should be explored, but as we only wish to show the viability of our tool we chose not to. Thus the results we report only represent one data point from a distribution of possible results.

5.2 Results

The results for the programs can be seen in Table 5.1. We will now go over some points of note in the results.

Looking at the ratio of test runs to solve runs, we can see that most programs require many more solve runs. For example, let us look at the "Byte add safe" programs number 1 and 2, which emulate a byte-wise carry adder and the specification for which is that the calculated addition matches C semantics. For these the only test run is the initial execution. After this no tests are generated and instead the abstraction is refined until the programs have been verified.

We can see two examples where a buggy version of a program is very fast to detect. In "Byte add unsafe" DASH needs only one solve run to find two values for which the emulated byte adder overflows and calculates a wrong result. A more extreme example is "Mutex lock int unsafe" where the faulty behavior does not depend on input and is found on the first arbitrary execution. On the other hand, not all of our unsafe programs were fast to falsify: in the "Stateful check" program the bug is found on one specific path and the program has many input dependent branches. As such DASH requires multiple iterations to discover the correct path.

Comparing the "Modulus" program to the "Jain" programs illustrates how different kinds of operations used can affect the verification time. The "Modulus" program implements an algorithm for calculating the remainder of a division without using the C remainder operator and uses bitwise ands and shifts to do so. The "Jain" programs, on the other hand, use additions and constant multiplications. The suitable predicates constructed for the bitwise operators are more difficult for an SMT solver to solve, which can be seen in that the verification uses equally many solver calls for both programs but takes approximately four times as long for the "Modulus" program.

The "Locks N" programs consist of an infinite loop, in which a random subset of N locks are first acquired and then the same locks are released in reverse order. As each lock is released, it is checked that the lock was actually previously acquired. Because the branches related to handling each lock are interleaved with each other, the verification time could potentially be multiplied whenever a new lock is introduced. However, looking at the running times from 5 to 15 locks we can see that a combinatorial explosion is avoided.

We also illustrate the effectiveness of the refinement for unsatisfiable constraints of target regions described in Section 3.4. For this we used the program in Figure 5.2. The assertion on line 28 is never violated because the variable `lock` is initialized to 1 and never changed. The assertion is preceded by five input dependent if-else constructs, which are irrelevant to the verification task at hand. We compared verifying this program with DSE, plain DASH and a version of DASH with support for the additional refinement method. The results are shown in Table 5.3. We can see that DSE and plain DASH achieve similar running times. However, the version with the check for unsatisfiable target regions is some 35 times faster than plain DASH. Comparing the iterations used we can see that plain DASH has encountered a combinatorial explosion due to the diamond structures in the program, while the modified version has not.

Name	Result	Time (s)	Test runs	Solve runs	SMT solver calls	Unsat targets
Alias of return 1	SAFE	0.039	1	1	2	0
Alias of return 1.1	SAFE	0.039	1	1	2	0
Alias of return 2	SAFE	0.107	2	4	6	2
Alias of return 2.1	SAFE	0.036	1	1	2	0
Byte add safe 1	SAFE	5.906	1	78	126	30
Byte add safe 2	SAFE	5.863	1	79	127	31
Byte add unsafe	UNSAFE	0.051	2	1	2	0
GCD 1	SAFE	1.039	3	20	34	6
GCD 2	SAFE	0.522	1	11	18	4
GCD 3	SAFE	1.699	2	18	30	6
GCD 4	SAFE	2.286	1	32	55	9
Jain 1	SAFE	0.421	1	9	15	3
Jain 2	SAFE	0.448	1	9	15	3
Jain 4	SAFE	0.656	1	9	15	3
Jain 5	SAFE	0.417	1	9	15	3
Jain 6	SAFE	0.541	1	9	15	3
Jain 7	SAFE	0.525	1	9	15	3
Modulus	SAFE	2.024	2	9	15	3
Mutex lock int safe	SAFE	0.102	1	3	5	1
Mutex lock int unsafe	UNSAFE	0.009	1	0	0	0
Num conversion	SAFE	0.611	1	13	21	5
oomInt	SAFE	0.148	1	4	7	1
Parity	SAFE	1.162	1	18	29	7
Size of parameters	SAFE	0.037	1	1	2	0
Stateful check	UNSAFE	0.856	4	17	38	2
Locks 5	SAFE	3.874	6	87	148	26
Locks 6	SAFE	4.838	7	107	182	32
Locks 7	SAFE	8.629	9	174	287	61
Locks 8	SAFE	6.746	9	147	250	44
Locks 9	SAFE	12.452	11	226	373	79
Locks 10	SAFE	8.831	11	187	318	56
Locks 11	SAFE	17.475	13	278	459	97
Locks 12	SAFE	20.002	14	304	502	106
Locks 13	SAFE	12.270	14	247	420	74
Locks 14	SAFE	13.328	15	267	454	80
Locks 15	SAFE	14.666	16	287	488	86

Table 5.1.: DASH test results

```
#include <lct.h>
#include <assert.h>
int main() {
    int lock = 1;
    int x = 0;
    int y = 0;
    int *p = &y;
    if (lct_get_bool())
        x = x + 1;
    else
        *p = *p + 1;
    if (lct_get_bool())
        x = x + 1;
    else
        *p = *p + 1;
    if (lct_get_bool())
        x = x + 1;
    else
        *p = *p + 1;
    if (lct_get_bool())
        x = x + 1;
    else
        *p = *p + 1;
    if (lct_get_bool())
        x = x + 1;
    else
        *p = *p + 1;
    assert(lock == 1);
}
```

Figure 5.2.: A program with diamond structures

Algorithm	Time (s)	Iterations	Solver calls	Unsat targets
DSE	6.883	32	31	-
DASH, plain	5.044	132	128	-
DASH, target unsat check	0.144	6	6	2

Table 5.3.: Results for verifying the program in Figure 5.2 with different algorithms

Chapter 6

Conclusions

Combining automated test generation with counterexample guided abstraction refinement is a promising approach to software verification. In this work we have presented LCT-D, which implements the DASH algorithm [4] for C programs on LLVM. As all of our instrumentation operates on the LLVM internal representation, with minor modifications LCT-D can be used for any programming language that compiles down to our supported set of instructions. Our case study, while limited, shows the viability of our implementation. LCT-D is open source and is available for download at: `http://users.ics.aalto.fi/osaariki/lctd-msc/`

As our tool targets the LLVM internal representation, we have in Section 3.2.2 presented a strategy for constructing splitting predicates for LLVM instructions in the presence of pointers. One limitation in LCT-D is that currently storing pointers to memory is not handled. We have left support for this for further work.

Section 4.5 describes how our tool evaluates splitting predicates for concrete states using only concrete values of pointers and the execution path taken. In contrast, the YOGI tool from Microsoft stores the complete concrete state along the execution path. Due to the storage requirements of storing concrete states YOGI stores sets of values, called *delta states*, that contain only the values that change between states [32]. LCT-D also stores delta states, but because only concrete values of pointers are stored our tool potentially uses less storage. One drawback with our approach is that the region of the final state of an execution must be known. Currently this is ensured in LCT-D as the test executions are not bounded and therefore can not terminate in a region that has been split.

In the future we plan to extend our tool to support a wider range of programs. This includes adding support for C structures, extending the algorithm to handle pointer arithmetic for array support and implementing an interprocedural version of DASH.

Once LCT-D has sufficient support to allow verification of real-world programs, exploring optimizations for the implementation and algorithm will become of interest. Some potential optimizations have already been evaluated in the YOGI tool. We are also interested in extending our tool to support multiple concurrent clients, which would allow test execution and constraint solving to be distributed to multiple machines. For dynamic symbolic execution LCT scales well to at least 20 clients [25], but whether we can achieve similar results for DASH remains to be seen.

An issue we encountered in DASH during implementation was that the predicates in regions could grow very large from continual splitting along the same abstract error trace. This problem disappeared once we implemented the more powerful refinement operation described in Section 3.4. However, as we attempt to verify larger programs we may encounter this problem again. One approach to alleviate this problem may be to employ interpolation, which can produce smaller predicates than weakest preconditions [20, 32].

Another direction for future work is extending our tool to support verification of multithreaded programs. We have previously extended LCT to support testing multithreaded programs by combining dynamic symbolic execution with the dynamic partial order reduction algorithm [35]. Our research group also has made advancements in testing multithreaded software by applying Petri net unfoldings to achieve better partial order reductions [24]. We would be interested to see if some of these techniques could be combined with the counterexample guided abstraction refinement approach of DASH.

Bibliography

[1] V. S. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A low-level virtual instruction set architecture. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, pages 205–216. ACM/IEEE, 2003. ISBN 0-7695-2043-X.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In Ferrante and Mager [11], pages 1–11. ISBN 0-89791-252-7.

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN)*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2001. ISBN 3-540-42124-6.

[4] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In B. G. Ryder and A. Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 3–14. ACM, 2008. ISBN 978-1-60558-050-0.

[5] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 25–32. IEEE, 2009. ISBN 978-1-4244-4966-8.

[6] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In Kowalewski and Philippou [26], pages 174–177. ISBN 978-3-642-00767-5.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2), 2008.

[8] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6): 388–402, 2004.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000. ISBN 3-540-67770-4.

[10] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[11] J. Ferrante and P. Mager, editors. *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1988. ACM Press. ISBN 0-89791-252-7.

[12] P. Godefroid and N. Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In J. Romijn, G. Smith, and J. van de Pol, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods (IFM)*, volume 3771 of *Lecture Notes in Computer Science*, pages 20–32. Springer, 2005. ISBN 3-540-30492-4.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[14] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2008.

[15] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: Whitebox fuzzing for security testing. *ACM Queue*, 10(1):20, 2012.

[16] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997. ISBN 3-540-63166-6.

[17] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In M. Young and P. T. Devanbu, editors, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 117–127. ACM, 2006. ISBN 1-59593-468-5.

[18] E. L. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Aspects of Computing*, 17(2):201–221, 2005.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In J. Launchbury and J. C. Mitchell, editors, *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 58–70. ACM, 2002. ISBN 1-58113-450-9.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 232–244. ACM, 2004. ISBN 1-58113-729-X.

[21] K. Kähkönen. Automated dynamic test generation for sequential Java programs. Master's thesis, Helsinki University of Technology, Department of Information and Computer Science, 2008.

[22] K. Kähkönen. Automated test generation for software components. Technical Report TKK-ICS-R26, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland, December 2009.

[23] K. Kähkönen, T. Launiainen, O. Saarikivi, J. Kauttio, K. Heljanko, and I. Niemelä. LCT: An open source concolic testing tool for Java programs. In *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, pages 75–80, 2011.

[24] K. Kähkönen, O. Saarikivi, and K. Heljanko. Using unfoldings in automated testing of multithreaded programs. In M. Goedicke, T. Menzies, and M. Saeki, editors, *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 150–159. ACM, 2012. ISBN 978-1-4503-1204-2.

[25] K. Kähkönen, O. Saarikivi, and K. Heljanko. LCT: A parallel distributed testing tool for multithreaded Java programs. In *Proceedings of the 11th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*, to appear.

[26] S. Kowalewski and A. Philippou, editors. *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, 2009. Springer. ISBN 978-3-642-00767-5.

[27] D. Kroening, A. Groce, and E. M. Clarke. Counterexample guided abstraction refinement via program execution. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods (ICFEM), Proceedings*, volume 3308 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2004. ISBN 3-540-23841-7.

[28] C. Lattner and V. Adve. LLVM assembly language reference manual, Apr. 2013. Available: `http://llvm.org/docs/LangRef.html`.

[29] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*, pages 75–88. IEEE Computer Society, 2004. ISBN 0-7695-2102-9.

[30] N. G. Leveson and C. S. Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[31] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[32] A. V. Nori and S. K. Rajamani. An empirical study of optimizations in YOGI. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE)*, pages 355–364. ACM, 2010. ISBN 978-1-60558-719-6.

[33] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The YOGI project: Software property checking via static analysis and testing. In Kowalewski and Philippou [26], pages 178–181. ISBN 978-3-642-00767-5.

[34] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In Ferrante and Mager [11], pages 12–27. ISBN 0-89791-252-7.

[35] O. Saarikivi, K. Kähkönen, and K. Heljanko. Improving dynamic partial order reductions for concolic testing. In J. Brandt and K. Heljanko, editors, *Proceedings of the 12th International Conference on Application of Concurrency to System Design (ACSD)*, pages 132–141. IEEE, 2012.

[36] K. Sen. *Scalable automated methods for dynamic program analysis*. Doctoral thesis, University of Illinois, 2006. URL `http://osl.cs.uiuc.edu/~ksen/paper/sen-phd.pdf`.

[37] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

[38] A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1996. ISBN 3-540-65306-6.

[39] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In L. L. Pollock and M. Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 145–156. ACM, 2006. ISBN 1-59593-263-1.

[40] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 427–440. ACM, 2012. ISBN 978-1-4503-1083-3.

[41] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

Appendix A

BGD Grammar

⟨*file*⟩	::=	⟨*global-list*⟩ ⟨*block-list*⟩
⟨*global-list*⟩	::=	⟨*global*⟩ ⟨*global-list*⟩ | ϵ
⟨*global*⟩	::=	‘`global`’ ⟨*type*⟩ `GLOBAL_IDENTIFIER`
⟨*block-list*⟩	::=	⟨*block*⟩ | ⟨*block*⟩ ⟨*block-list*⟩
⟨*block*⟩	::=	‘`block`’ `LOCAL_IDENTIFIER` ‘`{`’ ⟨*statement-list*⟩ ⟨*terminator*⟩ ‘`}`’
⟨*statement-list*⟩	::=	⟨*statement*⟩ ⟨*statement-list*⟩ | ϵ
⟨*statement*⟩	::=	== ⟨*type*⟩ ⟨*variable*⟩ ⟨*rvalue*⟩
	|	‘`store`’ ⟨*value*⟩ ⟨*variable*⟩
	|	‘`assertfail`’
	|	‘`alloca`’ ⟨*variable*⟩
⟨*variable*⟩	::=	`LOCAL_IDENTIFIER` | `GLOBAL_IDENTIFIER`
⟨*type*⟩	::=	`INT_TYPE`
	|	‘`ptr`’
⟨*rvalue*⟩	::=	⟨*value*⟩
	|	⟨*bin-op*⟩ ⟨*value*⟩ ⟨*value*⟩
	|	‘`load`’ ⟨*variable*⟩
	|	‘`phi`’ ⟨*source-list*⟩
	|	‘`select`’ ⟨*value*⟩ ⟨*value*⟩ ⟨*value*⟩
	|	‘`input`’
⟨*value*⟩	::=	`INT_LITERAL`
	|	⟨*variable*⟩
⟨*source-list*⟩	::=	⟨*source*⟩ | ⟨*source*⟩ ⟨*source-list*⟩

$\langle source \rangle$ ::= $\langle value \rangle$ `LOCAL_IDENTIFIER`

$\langle argument\text{-}list \rangle$ ::= $\langle value \rangle$ $\langle argument\text{-}list \rangle$ | ϵ

$\langle terminator \rangle$ ::= ‘`ret`’ $\langle value \rangle$
| ‘`retvoid`’
| ‘`br`’ $\langle variable \rangle$ `LOCAL_IDENTIFIER` `LOCAL_IDENTIFIER`
| ‘`goto`’ `LOCAL_IDENTIFIER`
| ‘`unreachable`’

$\langle bin\text{-}op \rangle$ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`%`’ | ‘`<<`’ | ‘`>>`’ | ‘`|`’ | ‘`&`’ | ‘`^`’ |
‘`==`’ | ‘`!=`’ | ‘`>`’ | ‘`>=`’ | ‘`<`’ | ‘`<=`’

All terminals must be separated by whitespace. The following rules we define by regular expressions:

Rule	Regular expression
`GLOBAL_IDENTIFIER`	`@[0-9]+`
`LOCAL_IDENTIFIER`	`%[0-9]+`
`INT_TYPE`	`i[1-9][0-9]*`
`INT_LITERAL`	`[0123456789abcdef]+`