A study of the DASH algorithm for software property checking

Kasper Føns, 20083881

Jacob Hougaard, 20083206

Master's Thesis, Computer Science

May 2014

Advisor: Anders Møller

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Developing error-free programs is a challenging and difficult task. Researchers at Microsoft Research have developed the DASH algorithm to verify that Windows NT device drivers are free of errors and have presented the algorithm in the DASH article [A2]. DASH is able to identify errors, by concretely executing a given program and to prove it as correct by refining a model of the program. The DASH article is compact, and the goal of this thesis is to uncover undescribed details by implementing a subset of DASH and then present the algorithm in detail. We also investigate whether the DASH algorithm has unknown weaknesses.

We have implemented two subsets of DASH. The first can analyze single-procedure integer programs and the second is an extension with interprocedural analysis. The implementation process was guided by test programs of increasing complexity, for which the implementation must determine whether the test programs contain errors or not.

We have found numerous details and special cases that were not disclosed in the DASH article. We have presented solutions to these cases and incorporated them into our presentation of DASH. Furthermore, we have also found example programs where DASH keeps refining the model infinitely. Our implementation is able to analyze programs written in a limited subset of Java. The limited subset includes static methods with integer arithmetic. Our implementation is released to the general public, unlike DASH, based on the assessment that it would be helpful for others that wish to understand DASH[1].

[1]Public repository: `https://github.com/foens/dash`

Resumé

Udvikling af fejlfrie programmer er både en udfordrende og vanskelig opgave. Forskere hos Microsoft Research har udviklet DASH algoritmen for at kunne verificere at Windows NT drivere er fejlfrie og har beskrevet algoritmen i DASH artiklen [A2]. DASH kan identificere fejl ved at afvikle et program konkret og kan bevise at det er korrekt ved at forfine en model over programmet. DASH artiklen er kompakt og formålet i dette speciale er at finde ubeskrevne detaljer ved at implementere en delmængde af DASH og derefter præsentere algoritmen i detaljer. Vi undersøger også om DASH algoritmen har ukendte svagheder.

Vi har implementeret to delmængder af DASH. Den første kan analysere heltals programmer med kun én procedure, og den anden er en udvidelse med interprocedural analyse. Implementeringsprocessen blev styret af testprogrammer med stigende kompleksitet, hvor implementeringen skulle afgøre om testprogrammerne indeholdte fejl eller ej.

Vi har fundet adskillige detaljer og særtilfælde, som ikke fremgik af DASH artiklen. Vi har præsenteret løsninger til disse tilfælde og tilføjet dem i vores præsentation af DASH. Derudover har vi også fundet eksempler på programmer hvor DASH forfiner modellen i en uendelighed. Vores implementering er i stand til at analysere programmer skrevet i en begrænset delmængde af Java. Den begrænsede delmængde inkluderer statiske metoder med heltalsaritmetik. Vores implementering er frit tilgængelig for offentligheden, i modsætning til DASH, ud fra en vurdering af at det kan være behjælpeligt for andre der vil forstå DASH[2].

[2]Offentligt repository: `https://github.com/foens/dash`

Acknowledgements

Thanks to our thesis supervisor Anders Møller for valuable feedback on this thesis and for the lengthy discussions about the DASH algorithm. The Microsoft employee Aditya V. Nori requires special thanks for the numerous emails he has answered for us, which have cleared up many details about the DASH implementation. Thanks to Mathias Schwarz for kick-starting us on how to use the Soot Java library. Thanks to Kent Grigo for comments on various parts of the thesis.

Kasper Føns and Jacob Hougaard,
Aarhus, May 1, 2014.

Contents

Chapter 1

Introduction

Software writing is a daunting task. The computer is mindless in the sense that it can only do what it is told, therefore all software programmers must go into absolute detail in how the program behaves. When software misbehaves, it can crash the program or corrupt data, which will leave the end user in disarray. This reflects badly on the software programmer or company that produced the program. When drivers crash, they do not only crash the program, they crash the complete operating system, which may require a reboot. Since such crashes are highly problematic, drivers often undergo rigorous reviews. Such reviews can be done manually by having a person check the code for errors or, more recently, software tools have emerged that can automatically check certain properties of the code.

DASH is such an algorithm that can automatically check safety properties by both concretely executing and abstractly analyzing a given C-program [A2]. It uses an under-approximation for finding errors and an over-approximation to prove the absence of errors. DASH executes the code both concretely and symbolically to maintain its abstractions. Concrete executions expand the under-approximation while refinements of the over-approximation make it more precise. An error is reported whenever an error state is reached by concrete execution, and the program has been validated whenever the over-approximation cannot reach the error state. DASH is developed by Microsoft to check safety properties in Windows NT device drivers. An example of a safety property could be that locks are acquired and released correctly. If safety properties are defined thoroughly and algorithms such as DASH are used, it is highly unlikely that device drivers crash the operating system.

The authors of DASH have described their implementation in the DASH journal article [A2]. Articles can be compact, simplistic, and straight to the point, with the result that some details may be missing. When reading such an article, the reader can often convince himself that implementing such an algorithm is straightforward. However, it can be the case that non-disclosed problems arise when implementing an algorithm since all details must be handled. Often articles only point out how superior their algorithms are whereas shortcomings are often not described. Such shortcomings are hard to find unless one knows the intricate details of the algorithm being described.

In this thesis, we seek to describe the DASH algorithm in detail and investigate whatever corner cases we might strike. To do so, we implement the DASH algorithm in Java such that it can analyze a limited subset of Java.

1.1 Thesis statement

> *What challenges are there in implementing the DASH algorithm for a restricted subset of Java?*

Expanding the thesis statement raises some questions:

- First of all, how does DASH work? The paper where DASH is described is compact and the authors leave out a lot of details [A2]. What are these details and how can they be addressed? We believe that implementing DASH is an effective strategy for finding these details. Conversely, by writing pseudocode that incorporates the missing details, we believe it may be possible to obtain a cleaner and more complete algorithm.
- When reading about DASH, it seems like a wonderful algorithm. The DASH article does not describe any weaknesses, so is it possible to find interesting examples of programs for which DASH gives up?
- DASH has been developed by Microsoft to verify NT device drivers, which are usually simpler than ordinary C programs. Can DASH be implemented to handle standard Java constructs? For example, DASH has support for structs as used in the programming language C, but how could one support Java objects? Java also includes an enormous and complex Standard Library, should it be mocked or analyzed by DASH?

1.2 Method

We seek to answer the above questions by using a method inspired by test-driven development [B8]. We start by constructing the smallest possible program, and then construct an implementation that answers correctly on it. The first program we want to answer correctly on is shown in Figure 1.1a. The program contains no code, and it is simple to implement a version of DASH that answers correctly by always answering that a program is correct. However, when the implementation also needs to answer correctly on a second test program seen in Figure 1.1b, much more work is needed. Implementing DASH correctly for these two programs is nontrivial, as it requires us to load the program, instrument it and execute a test that reaches the error statement. We will continue along this methodology by adding increasingly complex test programs and have our implementation answer correctly on them. In this way, our methodology is heavily inspired by test-driven development.

We seek to uncover implementation details not described in DASH by implementing increasingly larger subsets of DASH for a realistic programming language:

```
void testNoError()
{

}
```

(a) First test program

```
void testError()
{
  error;
}
```

(b) Second test program

Figure 1.1: (a) shows the first test program we want our DASH implementation to analyze. The program never fails as there is no code in it. An implementation could answer correctly by always stating that a given program is correct. When we introduce (b), we force ourselves to write much more code to check whether an error statement is reached. The reason is that an implementation of DASH needs to run a concrete test that reaches the error statement, and this requires a good deal of infrastructure. Thus, implementing a version of DASH that answers correctly on both these programs is nontrivial.

- A version of DASH that is able to analyze simple integer programs without any kinds of dependencies and without any procedure calls.
- A version of DASH that is able to analyze integer programs with procedure calls.

These implementations should be able to analyze small integer programs and find subtle errors in them. The small programs will be presented throughout this thesis together with the DASH implementations.

We leave it as future work to add support for Java objects, exceptions and types such as booleans and doubles.

1.3 Outline

We describe the terminology and work leading to the DASH algorithm in Chapter 2 after which we present DASH$_{int}$, a DASH algorithm able to analyze a single procedure that performs integer arithmetic in Chapter 3. In Chapter 4, we present DASH$_{call}$ which extends DASH$_{int}$ with interprocedural analysis. Chapter 5 describes the implementation details for implementing DASH in Java to analyze a limited subset of Java programs. In Chapter 6, we describe what future work could be conducted. We conclude in Chapter 7.

Chapter 2

DASH Background

DASH [A2] is an algorithm that can check safety properties. DASH builds on the SYNERGY algorithm [A5], which combines testing and verification methods. While SYNERGY only analyzes a single C-procedure without pointers, the DASH algorithm can analyze interprocedural programs with pointers.

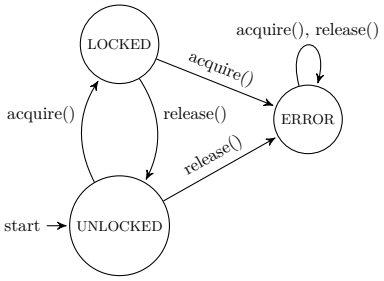
In this chapter we start by describing the concepts used in DASH, what safety properties are, what a property checker does and two broad classes of techniques used by property checkers, namely testing and verification. SYNERGY is then described, which is an algorithm that uses a combination of testing and verification techniques. Finally, a small introduction to DASH is provided at the end of the chapter.

2.1 Temporal safety properties

Safety properties specify that *something bad* does not happen in a program. Examples for which are *deadlock detection*, *releasing a lock that has not been acquired* or *closing a file that has already been closed*. Many such safety properties can conceptually be detected by a finite state machine that runs besides the program and observes what the program does.

As an example we will use the *lock acquire/release* safety property that specifies that locks should not be doubly acquired or released – i.e. calls to acquire() and release() should be interleaved, always starting with a call to acquire(). These requirements are captured by the finite state machine shown in Figure 2.1a. The machine will start in the UNLOCKED state. When the program makes a call to acquire() the state machine will follow the edge to the LOCKED state. If the program acquires the lock again without releasing it, the machine will end up in the ERROR state, which captures that the safety property has been violated.

An example program, borrowed from SLAM [A1], can be seen in Figure 2.1b. The program is meant to process interrupt requests in a Windows device driver. The external variable is shared among many threads and locks are required to enforce mutual exclusion while altering the state of the variable. The lock is acquired only in line 3 and released in line 8 and 14. It is not directly evident that this program satisfies the locking safety property. Property checkers, which



(a) Finite state machine φ_{lock}

```
1  void example() {
2    do {
3      acquire();
4      nOld = n;
5      request = external.listHead;
6      if(request ≠ null) {
7        external.listHead = r.next;
8        release();
9        // do stuff not needing lock
10       ...
11       n++;
12     }
13   } while(nOld ≠ n)
14   release();
15 }
```

(b) Example program $\mathcal{P}_{ex}$

```
void example() {
  state = UNLOCKED;
  do {
    if(state == UNLOCKED) state = LOCKED;
    if(state == LOCKED) state = ERROR;
    acquire();
    nOld = n;
    request = external.listHead;
    if(request ≠ null) {
      external.listHead = r.next;
      if(state == LOCKED) state = UNLOCKED;
      if(state == UNLOCKED) state = ERROR;
      release();

      // do stuff not needing lock
      ...
      n++;
    }
  } while(nOld ≠ n)
  if(state == LOCKED) state = UNLOCKED;
  if(state == UNLOCKED) state = ERROR;
  release();
}
```

(c) Combined program and finite state machine $\mathcal{P}_{ex} \circ \varphi_{lock}$

*Figure 2.1: Checking the safety property "*a lock is not acquired or released two times in row*", can be determined by using the finite state machine φ_{lock} depicted in (a). State machines can be described using the SLIC specification language [B7]. Each time either acquire() or release() is called the finite state machine changes its internal state. If the ERROR state is reached, the safety property has been violated. An example program $\mathcal{P}_{ex}$ seen in (b) uses locks for mutual exclusion. The program shown in (c) shows that the finite state machine φ_{lock} and the program $\mathcal{P}_{ex}$ can be combined to a single program $\mathcal{P}_{ex} \circ \varphi_{lock}$, and checking the safety property is converted to a reachability test of the statement state = ERROR.*

```
int abs(int a)
{
  if(a < 0)
    a = -a;
  assert a >= 0;
  return a;
}
```

Figure 2.2: Java example stating a safety property using an **assert** *statement.*

will be described in the next section, can check if a given program $\mathcal{P}$ satisfies a safety property φ.

The program shown in Figure 2.1c is a combination of the finite state machine φ_{lock} in Figure 2.1a and the original program $\mathcal{P}_{ex}$ in Figure 2.1b and we denote the combined program as $\mathcal{P}_{ex} \circ \varphi_{lock}$. Notice that the state machine has been embedded into the program. The problem of checking if a safety property φ is satisfied by a program $\mathcal{P}$ can thus be converted to a reachability test in the combined program $\mathcal{P} \circ \varphi$:

$$\mathcal{P} \textit{ satisfies } \varphi \equiv \textit{It is impossible to reach the ERROR state in } \mathcal{P} \circ \varphi$$

Such a translation is routinely completed in property checkers such as the SLAM [A1] and BLAST [A3] tools. In a Java program one could use **assert** e statements to state safety properties. The **assert** e statement can, in the context of safety properties, be seen as syntactic sugar for **if**(!e) state = ERROR. An example Java program using an **assert** statement can be seen in Figure 2.2. Figure 2.2 actually contains a subtle bug[1].

If we want to specify that no NullPointerExceptions are allowed to happen in a Java program, we would instrument the program such that all accesses of an object o, like o.var or o.method(), would have the check **assert** o != **null** added before it. Then it is the property checker's job to verify that objects cannot be **null** when accessed.

The next section describes property checkers and techniques used to implement them.

2.2 Property checker

A property checker is a program that is able to check if a given safety property is satisfied. If a property checker is *sound*, it means that whenever it reports a violation of a safety property it is an actual violation, however it may not be able to find all violations. If a property checker is *complete*, it will report all actual violations, but may also report false positives. It is desirable for a property checker to be both sound, complete, and always halt, since then all violations are reported and none of them are spurious.

For any real programming language and safety property the property checking problem is easily seen to be undecidable using Rice's theorem: the language

[1] Pass Integer.MIN_VALUE to abs and it will return the same value due to overflow.

```
void foo(int x, int y)
{
  if(x ≠ y)
    if(2x == x+10)
      assert false;
}
```

Figure 2.3: Low probability to reach an error state using random testing.

is Turing complete and the properties are non-trivial [B9]. This implies that a given property checker cannot answer correctly on all programs. A property checker must thus be unsound, incomplete or may not terminate with an answer for some programs.

Property checkers usually use algorithms that can be divided into two broad categories: *testing* and *verification*. Testing tries to violate the safety property by running the program concretely, with some input parameters, and observe if any of the safety properties are violated. Verification tries to construct a proof that shows a safety property cannot be violated. Each of these techniques have strengths and weaknesses. The techniques are described below.

2.3 Testing

As described above, testing tries to execute the program concretely and observe if any of the safety properties are violated. The tests executed can be seen as an under-approximation of the programs state space, since the possible program state of course includes the state space found executing the tests. Since it is an under-approximation technique it is not always possible to verify that the safety properties are never violated. However, when a safety property is reported as violated, there are no false positives since it was found during concrete execution and therefore testing is sound but not complete.

Random testing is a technique where a program is executed with random input. Intuitively, random testing has poor code coverage and certain program points have a very low probability to be exercised. An example of such a program can be seen in Figure 2.3, which has been borrowed from DART [A4]. To violate the **assert** statement the random testing tool has to pick $x = 10$ and $y \neq 10$, which is unlikely given the vast amount of possible input values. Given that integers are 32 bits large, there are $2^{32} \cdot 2^{32}$ possible input values. Since only values with $x = 10$ and $y \neq 10$ reaches the error, there are $1 \cdot (2^{32} - 1)$ input values which will violate the **assert** statement. Assuming a uniform distribution, there is only a $2.328 \cdot 10^{-10}$ probability to pick one of these input values. Another problem with random testing is that many different tests may exercise the exact same program path. This is redundant and hurts performance. In Figure 2.3 there are three classes of inputs that executes different program paths: those with $x = y$, those where $x \neq y \wedge 2x \neq x + 10$ and lastly those where $x \neq y \wedge 2x = x + 10$. Running anything more than three concrete test runs will thus result in redundant executions.

DART [A4] and CUTE [A6] improve the situation. Instead of randomly gen-

erating input, they try to generate input that directs their execution towards unvisited branches of the code. They accomplish this by executing the program both concretely and symbolically, which they termed concolic execution. Symbolic execution works by abstractly interpreting the program using symbolic variables in place of concrete values. They record branch conditions that they meet, try to negate one of them and use a constraint solver to generate a new input that may reach an unvisited branch. In this way they are able to easily find the error in Figure 2.3.

Using concolic execution raises code coverage considerably, however, most of the time 100% coverage cannot be achieved. There are multiple reasons for not reaching full code coverage. For example, programmers often use defensive programming to catch bugs early, however, most of the time the error conditions cannot be triggered due to the program being correct. Also notice that if **assert** e statements are treated as **if** (!e) state = ERROR then programs which are correct can never reach full code coverage. Also, in complex programs, the path constraint generated cannot be solved automatically. For example, solving sha256(x) == y would be very hard if y has a specific value. The hashing function sha256 is specifically created such that it is hard to find an input x that leads to a specific output y.

2.4 Verification

Verification is a method that strives to provide a proof that a given safety property cannot be violated. Verification tools try to generate an abstraction of the program that shows it is free of errors. Such a model of a program must contain all executions possible in the program, if some executions are missing those could be the ones that lead to an error.

Concretely, a model could be a control flow graph containing predicates about the state of variables at each program point. An example model for the code in Figure 2.3 can be seen in Figure 2.4. Execution starts in START and ends if the ERROR or EXIT statements are reached. Edges correspond to statements and nodes simply record predicates about the state of the program variables at that point. It can be seen that the model records that when node 1 is reached, the variables x and y must be different. It is not possible to verify that ERROR is unreachable since the underlying code from Figure 2.3 indeed contains an error.

A tool could try to construct a precise model of the program, a model that contains a path if and only if that path is possible in the program. Such models tend to capture too many constraints about the program, some of which might not be needed to check a given safety property. Since model checking in general is undecidable we know that such models are often too complex to verify. Instead, verification tools tend to create over-approximate abstractions that are simpler but contains executions that are not possible in the actual program. If it is possible to show that an over-approximate abstraction is error-free, then the program must also be error-free, since the program is contained in the model.

If, however, the model includes a path that leads to an error, it is uncertain

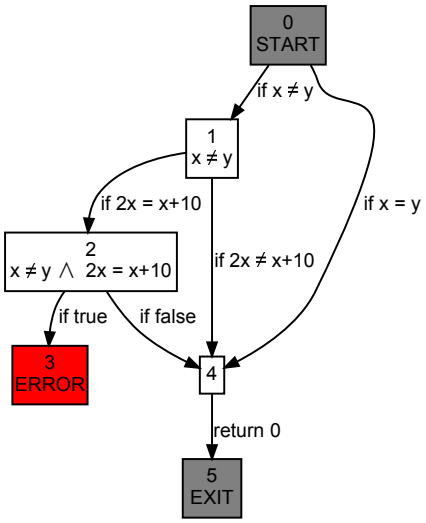


Figure 2.4: Abstraction that models the code in Figure 2.3. The condition inside ***assert false*** *has been negated and a special* ERROR *node has been added. Notice that node 4 cannot limit the values of x nor y since many executions paths with different valuations of the variables join there.*

if it signals a real error or if the path exists because the model is too coarse. If such paths are reported as errors, some of them can be spurious: verification is complete but usually not sound. Verification tools usually start with a very coarse abstraction and then try to refine it by pruning spurious paths that leads to an error.

The tools SLAM [A1] and BLAST [A3] take verification to the next level. They both start with a coarse abstraction of the program. The tools then search for paths leading to errors in their abstract model of the program. If no such paths exist, they are done. However, when a path exists in their abstractions, they check if that path is feasible in the original program by executing the path symbolically and using a constraint solver to validate the constraints generated. If the path is feasible, they have found a true error. If the path is not feasible, then their abstractions are too coarse and they refine their abstractions with new predicates. Such tools are both sound and complete, but unfortunately will not always halt with an answer. Both SLAM and BLAST have small sources of unsoundness and incompleteness in them. For example, BLAST does not model integer overflow, some infeasible paths cannot be proved as such by their decision procedures, and SLAM uses a logical model of memory where pointer arithmetic is ignored.

2.5 Synergy

SYNERGY is a predecessor to DASH and builds on lightweight symbolic execution and testing from CUTE and verification methods as used in SLAM and BLAST. SYNERGY maintains both an under-approximation like CUTE but also

maintains an over-approximation of the program like SLAM and BLAST. The over-approximation is used to guide what tests to execute while the tests guide which refinements of the abstraction are to be made. In a sense the two techniques work together in synergy, thereby the algorithm's name.

The algorithm first finds a path to an error in the over-approximation and then tries to create concrete input to the procedure-under-test such that at least one unexplored step in the path is executed. If SYNERGY is not able to generate such input, it removes the path from the over-approximation. If SYNERGY is able to move at least a single step forward, it concretely executes the program and thereby increases the under-approximation. SYNERGY is able to analyze single-procedure C programs with integer variables and arithmetic.

2.6 DASH

DASH [A2] builds on the ideas of SYNERGY [A5]. DASH can additionally handle interprocedural programs with pointers and has a more efficient implementation. As in SYNERGY, DASH maintains an over- and under-approximation of the program being tested.

There are three possible outcomes of executing DASH on a program instrumented with a safety property: 1) DASH never halts and executes forever, 2) DASH halts and reports that the safety property is violated and 3) DASH states that the program never violates the safety property. If DASH reports that the safety property is violated, then DASH has found a concrete execution the causes a safety property violation. If DASH reports that the safety property is never violated then the over-approximation is a proof thereof, since the states violating the property are no longer present in the over-approximation. This implies that DASH is both sound and complete, if it terminates.

Our presentation of the DASH algorithm is fundamentally different from the original presentation in the DASH article [A2]. Our presentation tries to remove points of confusion while also adding missing details that are not described properly in the article.

To run DASH on a program instrumented with a safety property, as was described in Section 2.1, one constructs the initial over-approximate abstraction and runs DASH on it. The abstraction of the program is initially equivalent to the control flow graph of the program where variables are not constrained yet. When given the abstraction, DASH will perform the same operations as SYNERGY does. It will first try to find a path in the abstraction that leads to an error. It will check if the path is feasible in the original program at least one step beyond a so-called frontier. If the path is feasible, it will be executed concretely and DASH will detect if an error was reached during execution. If the path is not feasible, the over-approximation will be refined to remove the infeasible path. If DASH is not able to find a path to an error in the over-approximation, the program satisfies the safety property.

DASH as presented by the DASH article can load test input for concrete executions when starting up, such that the under-approximation can be kick-started. In this presentation this technique has been left out as it only serves as

an optimization.

Our presentation of DASH is broken up into two implementations:

1. DASH$_{int}$ is an implementation that can handle a single procedure which is allowed to take integer arguments, perform integer arithmetic but it is not allowed to call other procedures. DASH$_{int}$ is essentially equivalent to the SYNERGY algorithm.

2. DASH$_{call}$ is an extension of DASH$_{int}$ that can also handle calls to other procedures. Only the main procedure may use the error statement. This limitation can be rectified by using global variables and checking them in the main procedure. See Section 4.6 for how this could be done.

We describe each of these implementations separately, starting with DASH$_{int}$ in Chapter 3. DASH$_{call}$ is explained in Chapter 4.

Chapter 3

Analyzing integer programs with DASH$_{int}$

This chapter presents the DASH$_{int}$ algorithm, which is able to analyze single procedure integer programs and thus constitutes a subset of the DASH algorithm.

DASH$_{int}$ is divided into multiple procedures that together implement the DASH$_{int}$ algorithm. The main procedure of DASH$_{int}$ is DashLoop. It calls other procedures which in turn may again call other procedures. Figure 3.1 shows the call hierarchy for each procedure of the DASH$_{int}$ algorithm and in which section the procedure is described.

The next sections describe what types of programs DASH$_{int}$ handles and the key region graph data structure. Afterwards the algorithm itself is presented starting with the DashLoop procedure. Then all sub procedures of DashLoop are described, excluding ExtendFrontier and its sub procedures, which has been given their own sections because of their importance. After all of the procedures used by DASH$_{int}$ have been explained, we perform a complete walk through of how DASH$_{int}$ analyzes the example procedure abs introduced in Chapter 2. Finally, we describe key problems encountered and changes we have made to implement DASH$_{int}$ in Section 3.8.

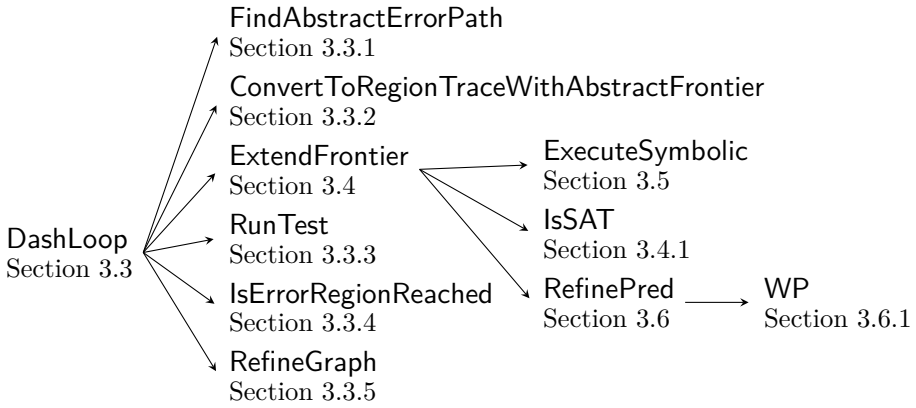


Figure 3.1: Overview over called procedures in DASH$_{int}$. The procedures are listed in call order. For example DashLoop calls ExtendFrontier before RunTest.

⟨*expr*⟩ ::= ⟨*operand*⟩
| '-' ⟨*operand*⟩
| ⟨*operand*⟩ ⟨*binop*⟩ ⟨*operand*⟩

⟨*binop*⟩ ::= '+' | '-' | '*' | '/' | '%' | '~' | '^' | '|' | '&' | '≪' | '≫' | '⋙'

⟨*cond*⟩ ::= '**true**'
| '**false**'
| ⟨*operand*⟩ ⟨*cond-op*⟩ ⟨*operand*⟩

⟨*cond-op*⟩ ::= '==' | '≠' | '<' | '>' | '≤' | '≥'

⟨*operand*⟩ ::= ⟨*identifier*⟩
| ⟨*constant*⟩

Figure 3.2: Grammar showing the allowed forms for expressions and conditionals.

3.1 Program structures supported by DASH$_{int}$

This section describes in detail what types of programs DASH$_{int}$ analyzes and what are assumed about them.

DASH$_{int}$ analyzes programs instrumented with error statements that are executed whenever a desired safety property has been violated. Thus the safety property itself is never passed to DASH$_{int}$ and the objective is instead to ensure that no error statements are reachable. This type of program transformation was presented in Chapter 2, Figure 2.1.

The program $\mathcal{P}$ is restrained to a single procedure where the following statements are allowed:

- Assignments of the form v := e where v is a local variable and e is an expression.
- Conditionals of the form **if** c **goto** l where c is a conditional expression and l is a program location.
- An error statement.

The **return** statement is allowed but seen as no-operation statement. Expressions and conditionals are restricted to simple forms and a grammar is given in Figure 3.2. The expressions and conditionals are constrained, such that they are side effect free and cannot be nested.

Nested expression or expressions with side effects can be rewritten to simple expressions by introducing new temporary variables. The **while** and **for** loop constructs can be desugared into **if** c **goto** l and **goto** l statements. Simple **goto** l statements can be modeled using **if true goto** l.

The example procedure abs from Figure 2.2, here reproduced as Figure 3.3a, functions as a running example program for explaining DASH$_{int}$. Notice that the program is not written in a form that DASH can analyze. Translating it to a

```
int abs(int a)
{
  if(a < 0)
    a = -a;
  assert a ≥ 0;
  return a;
}
```

(a) DASH$_{int}$ example procedure abs.

```
int abs(int a)
{
  if a ≥ 0 goto l1;
  a = -a;

 l1:
  if a ≥ 0 goto l2;
  error;

 l2:
  return a;
}
```

(b) Desugared version of abs.

Figure 3.3: (a) shows the abs procedure in a non-desugared version. (b) shows the same procedure but desugared into a form that DASH$_{int}$ accepts.

version that can be analyzed is straightforward since the **if** statement can easily be converted to an **if** c **goto** l statement and the **assert** statement is analogues to an **if** with an error statement inside. The raw desugared version, which can be analyzed by DASH, is shown in Figure 3.3b.

3.2 Region Graph

A *region graph* is the central data structure used by DASH. It models both the over-approximation and under-approximation of the reachable state space. A region graph is essentially the same as the example abstraction mentioned in Section 2.4 used by verification tools, but with the added feature that it also contains concrete executions that represent the under-approximation. In the DASH paper the under-approximation is contained in a separate data structure called a *forest*. We chose to embed the states from the forest directly into the region graph, due to frequent lookup of which states that are contained in a region.

A region graph is a set of nodes and edges. Nodes are called regions and model a set of states that satisfy the predicate that is attached to the region. Edges between nodes are labeled with the program statements. Initially the region graph is equivalent to the control flow graph for the program. The first region is called the *initial region*, which is the region that exists before any statements have been executed. If a region has an outgoing edge labeled with an error statement, the region is termed an *error region*. Our regions keep the edges with an error statement, but these could have been removed and concrete execution would have to halt if an error region was reached.

When creating a region graph, **if** c **goto** l statements become two edges with assumes, one with **assume** c and the other with **assume** ¬c. When presenting the graphs, the following conventions apply:

- Regions are displayed as boxes, and edges between them correspond

to potential program state transitions. Regions initially have a unique identification number assigned but it is only used for presenting the graphs. Edge labels correspond to statements in the program.

- For **assume** c statements, only the condition c is written as edge labels to preserve space. To distinguish them from other edge labels, the condition c is written with a **blue** font.

- The initial region is colored **gray**. The special error regions, the regions before an error statement, are colored **red**. Regions and edges that cannot be reached from the initial region are colored **yellow**. Unreachable error regions are colored **light red**.

- A region models all states that satisfies the attached predicate. An example predicate could be $a > 0$. Predicates are shown below the region identification number. If the region predicate is **true**, which is trivially satisfied by all states, it is omitted for brevity.

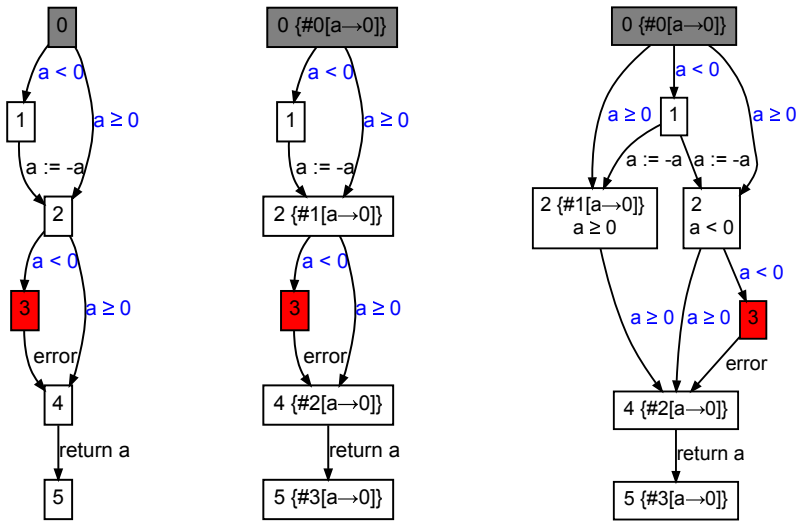
- Regions can have concrete states attached to them originating from a concrete execution of the procedure. The list of states are shown in curly braces $\{\dots\}$, with individual states in square brackets $[\dots]$. They constitute the under-approximation maintained by DASH. A number $\#n$ represents how many statements have been executed before reaching the state. An example state could be $\#3[a \mapsto 0]$. Without $\#n$ it can be hard to distinguish states inside a loop from each other.

Figure 3.4a shows the initial region graph for the abs procedure presented in Figure 3.3a. Notice that the region graph is identical to the control flow graph of the program.

Figure 3.4b shows the region graph after a test has been executed and the concrete states have been added to the region graph. It can be seen that abs was called with $a \mapsto 0$. It can also be seen that the error region has not been reached by the test, since the error region contains no states. The concrete states of a particular concrete execution are linked together as a doubly-linked list. It is therefore possible to find both the parent and child state of a particular state.

Formally, let Σ be the state space for a procedure P. It contains all the different valuations of the variables known by P. Program state can be altered when statements are executed which naturally defines a function $\rightarrow\colon \Sigma \rightarrow \Sigma$ from one state to another for P.

The region graph models equivalence classes of states $\Sigma_{\simeq}$, such that states in an equivalence class are thought to have the same properties. An example could be that all states with $x > 0$ are modeled to be in the same equivalence class. Figure 3.4c shows how such equivalence classes can be expressed in the region graph. Region 2 has been split into two. One with the predicate $a < 0$ and the other with the negation $a \geq 0$. The region with the predicate $a \geq 0$ cannot enter the error region 3, since an edge has been removed. The edge would have been labeled with an **assume** $a < 0$ statement, which cannot be satisfied under the condition $a \geq 0$.



(a) Initial graph (b) States added (c) With predicates, splitting region 2

Figure 3.4: (a) shows the initial region graph for the abs procedure seen in Figure 3.3a. (b) shows how states from a concrete execution are added to the region graph. It can be seen that abs was called with $a \mapsto 0$. (c) shows how equivalence classes of states are modeled in the graph, here by splitting region 2.

Edges in the region graph correspond to possible state changes between equivalence classes and we denote the edges $\rightarrow_{\simeq}$: $\Sigma_{\simeq} \times \Sigma_{\simeq}$.

When DASH begins execution, the region graph is equivalent to the control flow graph of the program and all regions contain the single predicate **true**, which allows any variable to take any value. Whenever the DASH algorithm refines the abstraction, a region is split into two using a predicate that is suitable for refinement. The main algorithm, including refinement, will be explained in following sections.

3.3 The main loop

DashLoop is the main loop of DASH$_{int}$ and pseudocode is provided in Algorithm 3.1. It takes a procedure P and a region graph G as input and will return PASS or FAIL depending on whether an error region is reachable. If it reports a failure, it will also return the input values needed to reach the error region. In case the program passes, the refined region graph will be returned as proof of the error region being unreachable.

DashLoop starts by calling FindAbstractErrorPath to find an abstract error path τ in the region graph G that leads to an error region. Whenever we refer to a *path* or write τ it is the abstract error path returned by FindAbstractErrorPath we are referring to. If FindAbstractErrorPath is unable to find a path that leads to an error region, then the graph G proves that no error state is reachable in P since G is an over-approximation of P. If a path is found, then ConvertToRe-

Algorithm 3.1 DashLoop($P, G = \langle \Sigma_{\simeq}, \rightarrow_{\simeq} \rangle$)

Returns:
(FAIL, t), where t is a list of input values for reaching error; or
(PASS, G), where G is a proof that the error cannot be reached.

```
 1: loop
 2:     τ := FindAbstractErrorPath(G)
 3:     if τ = NO-PATH then
 4:         return (PASS, G)
 5:     end if
 6:
 7:     τ_c := ConvertToRegionTraceWithAbstractFrontier(τ, G)
 8:     ⟨t, ρ⟩ := ExtendFrontier(τ_c, P)
 9:     if t ≠ UNSAT then
10:         G := RunTest(t, P, G)
11:         if IsErrorRegionReached(G) then
12:             return (FAIL, t)
13:         end if
14:     else
15:         G := RefineGraph(ρ, τ_c, G)
16:     end if
17: end loop
```

gionTraceWithAbstractFrontier finds the last region in the abstract error path τ with a concrete state. This region is denoted S_{k-1}. The state found in S_{k-1} is followed back to the initial region and this leads to the trace $\langle S_0, \ldots, S_{k-1} \rangle$ that constitutes the first part returned by ConvertToRegionTraceWithAbstractFrontier. The last part of the trace is the region in τ right after S_{k-1} which we denote S_k. The edge between S_{k-1} and S_k is termed the *frontier*. The new trace $\tau_c = \langle S_0, \ldots, S_{k-1}, S_k \rangle$, returned by ConvertToRegionTraceWithAbstractFrontier, is created in this way such that the trace leading up to the frontier is known to be feasible. If the trace is found infeasible, it must be because of the frontier edge. Whenever we refer to a *trace* or use the symbol τ_c, then we are referring to the trace returned by ConvertToRegionTraceWithAbstractFrontier.

The next step is to attempt to extend the frontier by finding a test input that will cross the frontier edge when executed concretely. This will bring us at least one step closer to the error region and to showing that the error region is reachable. The ExtendFrontier procedure attempts to find such test input, but if such a test does not exist, it finds a predicate that can be used to refine the region graph, such that the current trace τ_c is eliminated from G. If ExtendFrontier finds a test input, then it will return the pair $\langle t, \mathbf{true} \rangle$, where t is the input values that will push execution over the frontier edge. RunTest is used to execute the test on an instrumented version of P, where for each executed statement the concrete state of the variables is saved in the reached region. An updated region graph is returned from RunTest, with the new concrete states added.

If an error region is reached during concrete execution, then RunTest has added a state to it. The procedure IsErrorRegionReached returns **true** if there is an error region that contains a concrete state. If IsErrorRegionReached returns **true** then the input values t must be the input that led to the error being reached.

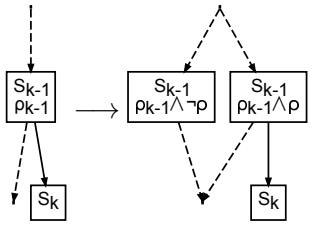


Figure 3.5: Refinement process of region S_{k-1}, which contains the region predicate ρ_{k-1}, using the refinement predicate ρ. Notice that the region with $\neg\rho$ added has the edge to region S_k removed.

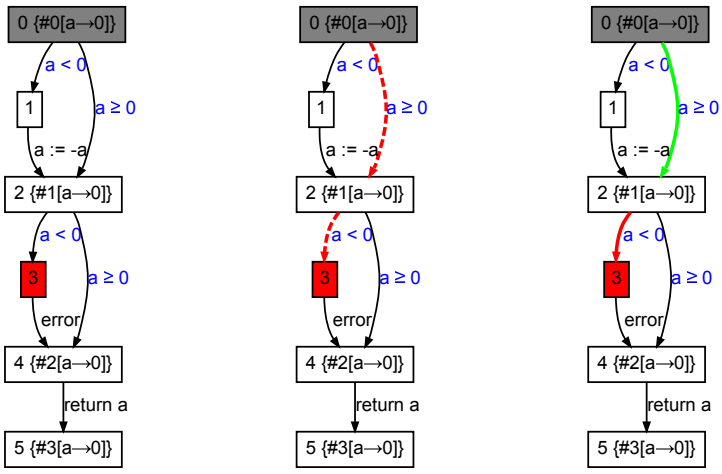
This is evident since the error regions are checked after each RunTest invocation, and thus before RunTest was called with t, the error regions did not contain any states. Therefore the concrete state must have arisen because of the latest RunTest invocation with t as input.

Even if the test did not reach the error region sought by the path τ, the frontier has been advanced and concrete states will have come closer to the error region. Given that FindAbstractErrorPath uses a deterministic graph search algorithm, which is not a requirement, the same abstract path will be found in the next iteration of DashLoop. However ConvertToRegionTraceWithAbstract-Frontier will find a longer trace where the frontier has been pushed forward. Remember that the last test crossed the last frontier, and therefore a state was added to region S_k. This pushes the next frontier found by ConvertToRegionTraceWithAbstractFrontier forward by at least one region and in this way progress is achieved.

If ExtendFrontier is not able to find a test input, then it returns a suitable refinement predicate ρ. The definition of a suitable refinement predicate is provided in Section 3.6, but the basic property is that it excludes the current trace τ_c and if the predicate is not satisfied at region S_{k-1} then region S_k cannot be reached. RefineGraph refines the region graph by using these properties of the refinement predicate. Specifically it splits region S_{k-1} into two. One which has the refinement predicate ρ added and one with the negation $\neg\rho$ added. Because of the properties that ρ has, the region with $\neg\rho$ added cannot reach S_k and therefore the edge to S_k is removed. The refinement process of splitting region S_{k-1} into two can be seen in Figure 3.5 and is also explained in greater detail in Section 3.6.

The resulting graph is a better approximation of the program. The graph reflects that to follow the edge (S_{k-1}, S_k) then ρ must be satisfied. DashLoop starts the next iteration and tries to find a new abstract error path to the error region, since the current one has now been eliminated.

DashLoop has now been described. We now begin to describe the procedures that DashLoop calls, starting with FindAbstractErrorPath.



(a) Region graph

(b) Abstract error path $\langle 0, 2, 3 \rangle$

(c) Concrete error trace with frontier

Figure 3.6: (a) shows the region graph for the abs *procedure in Figure 3.3a where* abs *has been called with the concrete input* $a \mapsto 0$ *and concrete states have been added to the graph. (b) shows the abstract error path with dashed red lines found by* FindAbstractErrorPath*. (c) shows the path converted to a trace. The part that follows a concrete execution is shown with green edges while the frontier edge is shown in red.*

3.3.1 FindAbstractErrorPath

The goal of FindAbstractErrorPath is to find a path in the region graph G from the initial region to an error region. If the error regions are unreachable, then the special value NO-PATH is returned. The DASH article does not describe how the path should be found. It could be implemented using a standard breadth-first search from the initial region, stopping when an error region is encountered.

One could also use a depth-first search, and this should not make a difference to how DASH works. However, for a developer, it is most beneficial to describe the shortest path leading to an error, instead of a longer one involving irrelevant operations.

Figure 3.6a shows the region graph for the abs procedure after an initial test has been executed, without reaching the error region. The path $\langle 0, 2, 3 \rangle$ is shown in Figure 3.6b using dashed red edges (- - -). It is the shortest abstract error path found by FindAbstractErrorPath using a breadth-first algorithm. By convention, whenever we present a path in a region graph, the path is always shown using dashed red edges.

The path is converted by ConvertToRegionTraceWithAbstractFrontier to a trace that follows a concrete execution. We describe ConvertToRegionTraceWith-AbstractFrontier in the next section.

3.3.2 ConvertToRegionTraceWithAbstractFrontier

ConvertToRegionTraceWithAbstractFrontier is responsible for converting an abstract error path τ into a trace that follows a concrete execution for as long as possible, and then takes one extra step in the direction of the abstract error path. This is completed by finding the last region in τ that contains a state, which we name S_{k-1} for the region and s_{k-1} for the particular state. If the region contains multiple states, then the newest one is chosen. The reason for picking the newest state is described in Section 3.8.7.

The parent relationship of s_{k-1} is used to search backwards to the initial region and the execution trace $\langle s_0, \ldots, s_{k-1} \rangle$ is obtained. It is assumed that one can find the regions which the states are attached to, such that the trace $\langle S_0, \ldots, S_{k-1} \rangle$ of regions can be obtained. This is the part of the trace that follows a concrete execution. Notice that $\langle S_0, \ldots, S_{k-1} \rangle$ might not follow the path taken by τ, which we describe in Section 3.8.6. Next the frontier edge is added, by adding region S_k, which is the region located after region S_{k-1} in the abstract path. The trace $\tau_c = \langle S_0, \ldots, S_{k-1}, S_k \rangle$ is returned. The frontier is unexplored in the sense that no concrete test have ever crossed that particular edge in the abstraction.

Figure 3.6c shows the trace $\tau_c = \langle 0, 2, 3 \rangle$ found when the path shown in Figure 3.6b is converted. The part of the trace $\langle S_0, \ldots, S_{k-1} \rangle$ that follows a concrete execution is shown using **green** edges while the frontier (S_{k-1}, S_k) is shown in **red**. The path τ, from which τ_c is created, is nearly always longer than τ_c. The only case where they have the same length, is when S_k is an error region and S_{k-1} already contains a state from a concrete execution. In the cases where the path is longer than the trace, the remaining part of the path, the part from region S_k to the error region, is shown using red dashed edges (**- - -**) to aid the reader in where the path τ sought to reach. Thus, the reader should remember that when a trace τ_c is depicted in a figure, then the dashed lines are *not* part of the trace.

The reason that DASH converts an abstract error path into a region trace with a frontier is because then it is known that the part of the trace that follows a concrete execution is feasible. Thus DASH knows that region S_{k-1} is reachable. If DASH finds that the whole trace is infeasible, then DASH knows that it is the frontier edge that made it infeasible. Then the frontier is refined by RefineGraph, as explained later in Section 3.3.5, such that the trace τ_c is excluded from the region graph and the abstraction is made more precise.

The first time an abstract error path τ is converted to a trace τ_c there are no regions in G that has been reached by concrete execution. For this case ConvertToRegionTraceWithAbstractFrontier assumes that there is a concrete state in the initial region, even though there is not. The trace τ_c does then only contain the frontier edge, which is the first edge contained in the path τ. This assumption is not problematic since all test executions start in the initial region and therefore trivially reaches it.

3.3.3 RunTest

RunTest is responsible for executing a procedure P concretely and record the concrete states for each executed statement. RunTest is given a list of input values t, the procedure P and the graph G. It runs a concrete test on P with t as input. During execution the state of the variables is recorded and added to the graph G. The concrete states are later used to check if the test reached an error region and to compute traces as described in Section 3.3.2.

The states can be recorded by executing an instrumented version of P and recording concrete values of variables at each program point. Alternatively one could simulate the code and then record the state at each step in the simulator.

Common for both solutions are that they need to follow the concrete execution in the region graph to insert the states in the correct regions. For example, if execution is at a region which has multiple outgoing edges, RunTest needs to figure out what edge the current concrete execution takes. Since the equivalence classes modeled are disjoint there is only a single edge that can be traversed. To check if an **assume** c edge is traversable, the conjunction of the region predicate and the condition c has to evaluate to **true** by inserting the concrete values of all variables into them. For assignment edges the state needs to be updated first and then the region predicates can be evaluated, to find the single one that evaluates to **true**.

The implementation of DASH in YOGI used a simulator to both concretely execute tests and to perform symbolic execution, whereas our Java implementation uses an instrumented version to execute the test [A2]. We describe our RunTest implementation in Section 5.4.

3.3.4 IsErrorRegionReached

IsErrorRegionReached is responsible for checking if an error region has been reached. IsErrorRegionReached takes as input the region graph G. It then runs through all regions and checks if a concrete state has been added to an error region, in which case the error region has been reached. If a state is found in an error region then the procedure returns **true**, otherwise it returns **false**. The pseudocode for IsErrorRegionReached is shown in Algorithm 3.2. Notice that the syntax **let** $\langle \_, states \rangle = S$ is used to unpack region S into its region predicate and its contained states. However, an underscore _ is used to denote that the region predicate is unused for this particular procedure.

To prevent an expensive search through all regions an actual implementation should maintain a set of error regions and only loop through that particular set to check if one of them contains a newly added state. Another solution could be to let RunTest report if an error region was reached during execution. RunTest could do this be setting a flag if it adds a state to an error region.

3.3.5 RefineGraph

RefineGraph is responsible for altering the region graph such that a trace τ_c that was found infeasible, i.e. test input could not be generated for it, is removed from the graph and therefore from the over-approximation. A suitable refinement

Algorithm 3.2 IsErrorRegionReached($G = \langle \Sigma_{\simeq}, \rightarrow_{\simeq} \rangle$)

Returns:
true, if a test has reached an error region; or
false, if no test has.

```
1: for S ∈ Σ_≃ do
2:     let ⟨_, states⟩ = S
3:     if IsErrorRegion(S) ∧ states ≠ ∅ then
4:         return true
5:     end if
6: end for
7: return false
```

predicate is used to refine the region graph. How the suitable refinement predicate is obtained is described in Section 3.6.

RefineGraph takes as input an infeasible trace τ_c, a suitable refinement predicate ρ and the region graph G to refine. The pseudocode for RefineGraph is given in Algorithm 3.3.

Algorithm 3.3 RefineGraph($\rho, \tau_c = \langle S_0, \ldots, S_{k-1}, S_k \rangle, G = \langle \Sigma_{\simeq}, \rightarrow_{\simeq} \rangle$)

Returns: $\langle \Sigma_{\simeq}, \rightarrow_{\simeq} \rangle$, the refined region graph.

```
1: let ⟨ρ_{k-1}, states⟩ = S_{k-1}
2:
3: if k = 1 then                                          ▷ Initial region refinement
4:     return ⟨Σ_≃, →_≃ \(S_0, S_1)⟩
5: end if
6:
7: Σ*_≃ := Σ_≃ \ {S_{k-1}}                                ▷ Remove S_{k-1}
8: →*_≃ := →_≃ \{(S, S_{k-1}) | S ∈ Parents(S_{k-1})}
9: →*_≃ := →*_≃ \{(S_{k-1}, S) | S ∈ Children(S_{k-1})}
10:
11: ρ*_{k-1} := Simplify(ρ_{k-1} ∧ ¬ρ)
12: S*_{k-1} := ⟨ρ*_{k-1}, states⟩
13: Σ*_≃ := Σ*_≃ ∪ {S*_{k-1}}                             ▷ Insert S*_{k-1}
14: →*_≃ := →*_≃ ∪{(S, S*_{k-1}) | S ∈ Parents(S_{k-1})}
15: →*_≃ := →*_≃ ∪{(S*_{k-1}, S) | S ∈ Children(S_{k-1})}
16:
17: ρ**_{k-1} := Simplify(ρ_{k-1} ∧ ρ)
18: S**_{k-1} := ⟨ρ**_{k-1}, ∅⟩
19: Σ*_≃ := Σ*_≃ ∪ {S**_{k-1}}
20: →*_≃ := →*_≃ ∪{(S**_{k-1}, S) | S ∈ Children(S_{k-1})}
21: if IsSAT(ρ**_{k-1}) ≠ UNSAT then                      ▷ Add incoming edges if ρ**_{k-1} is satisfiable
22:     →*_≃ := →*_≃ ∪{(S, S**_{k-1}) | S ∈ Parents(S_{k-1})}
23: end if
24:
25: →*_≃ := →*_≃ \{(S*_{k-1}, S_k)}                       ▷ Remove frontier edge from S*_{k-1}
26: return ⟨Σ*_≃, →*_≃⟩
```

Lines 3-5 handle the special case of refining the initial region. We first describe the general refinement idea and will come back to this special case afterwards.

The general idea is to split the region before the frontier edge, such that the equivalence class modeled by that region is split into two: one class that will

not enter the region after the frontier and one class that might enter it. This is completed by using the given suitable refinement predicate ρ. RefineGraph creates two new regions that contain the same predicate as the original, but where one region has ρ added and one has $\neg\rho$ added. Thus the equivalence class is split into two. The frontier edge is removed from the region that has $\neg\rho$ added as that edge is now impossible to follow, because of the properties of ρ.

More formally, a refinement at the frontier edge (S_{k-1}, S_k) is completed by splitting region S_{k-1} into two. Let ρ_{k-1} be the region predicate of S_{k-1} and let *states* be the concrete states attached to the region. We can then write region S_{k-1} as $S_{k-1} = \langle \rho_{k-1}, states \rangle$. When region S_{k-1} is split, two new regions are created: $S^*_{k-1} = \langle \rho_{k-1} \wedge \neg\rho, states \rangle$ and $S^{**}_{k-1} = \langle \rho_{k-1} \wedge \rho, \emptyset \rangle$. Notice that region S^*_{k-1}, which has $\neg\rho$ added as a predicate, contains all the concrete states that was attached to region S_{k-1}. The reason for this is that RefinePred, as described in Section 3.6, computes a refinement predicate that moves all the concrete states to S^*_{k-1}. The two new regions replace the region S_{k-1} that was split. The refinement process was depicted in Figure 3.5, when DashLoop was described.

Lines 7-9 in the pseudocode remove the split region S_{k-1} from the region graph. Lines 11-15 create and insert region S^*_{k-1}. Lines 17-23 insert region S^{**}_{k-1}. As can be seen in lines 11 and 17, the predicates attached to the new regions are simplified. We elaborate on how these predicates are simplified in Section 3.8.3. Simplification is optional but can be used to keep the predicates concise.

In some cases there are no states that can satisfy the predicate $\rho_{k-1} \wedge \rho$ added to region S^{**}_{k-1}. Thus the equivalence class that is modeled is empty. In other words the predicate $\rho_{k-1} \wedge \rho$ is unsatisfiable and thus equivalent to **false**. This is detected in line 21 with a call to IsSAT, which returns UNSAT if the predicate is unsatisfiable. For this case we add the region to the graph but refrain from adding incoming edges. The region is thus carved out of the graph and becomes unreachable. Had we added the incoming edges then FindAbstractErrorPath would be able to find a path through the region. Without this check some programs are refined infinitely, as we describe in Section 3.8.3. We could have removed the region altogether, however, we keep it as it makes it easier to interpret a series of graphs when regions are not disappearing.

Refinement of the initial region

We have had troubles with how DASH, as described in the DASH article, handles refinements of the initial region. We elaborate on this in Section 3.8.10.

When RefineGraph refines an outgoing edge from the initial region, the edge is removed without splitting the initial region. This is completed in lines 3-5 in the RefineGraph pseudocode.

In general, if there exists only one path that can reach an edge, and have it as the frontier, then if the path is found infeasible, the edge can be removed altogether. Since no other paths can lead to it, no harm has been made by removing it. If another path could lead down to the edge, then that path may lead to a different state that could allow the edge to be traversed. We require that only one path leads to the initial region, and then removing the edge when refining the initial region is a valid refinement.

There are cases where the initial region may be reached by more than one trace. For example, when the initial region is part of a loop, a trace could reach the initial region by taking x iterations of the loop. Thus more than one trace lead to the initial region. For such cases we could hoist the initial region by constructing a dummy initial region above with an **assume true** edge leading to the original initial region.

Refinement example

Figure 3.7 provides an example of refinement. Figure 3.7a shows the trace $\tau_c = \langle 0, 2, 3 \rangle$ that is being analyzed. The trace is found to be infeasible due to two inconsistent assumptions on the edges, and region 2 is split in two. The result of the split is seen in Figure 3.7b. The two regions, marked with red borders, are those that were created from the split of region 2. The refinement predicate $\rho = a{<}0$ was used to perform the split and it is placed on one of the regions while the negation $\neg\rho = a{\geq}0$ is placed on the other region. The region with $\neg\rho$ added has had the frontier edge removed. Thus there is no edge from it to region 3 anymore.

The trace $\tau_c = \langle 0, 2{:}a{<}0, 3 \rangle$ found in the refined graph can be seen in Figure 3.7c. Since there are two regions with region number 2 we disambiguate the regions by additionally specifying the region predicate using the syntax $regionNumber{:}predicate$. The new trace passes through the region with ρ added. As can be seen, the frontier edge has been pushed one step backwards in the trace compared to the trace in Figure 3.7a.

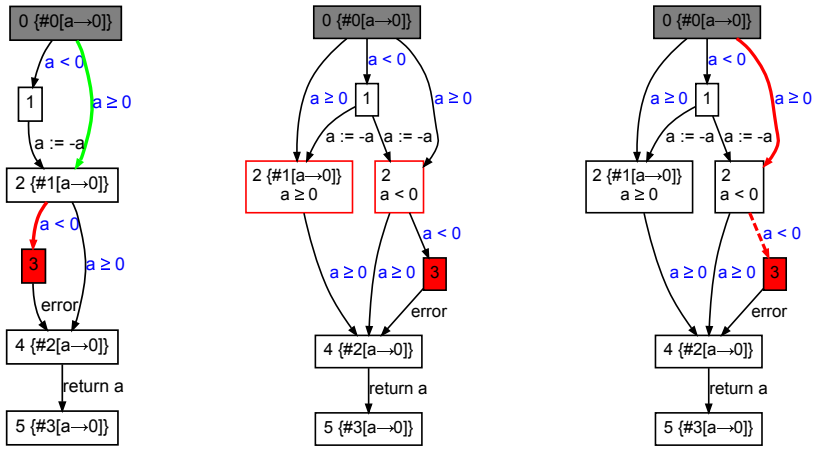
3.4 ExtendFrontier

ExtendFrontier takes as input a trace τ_c constructed by ConvertToRegionTraceWithAbstractFrontier and the procedure P. It either returns a test input that follows τ_c, which effectively advances the frontier closer to the error region, or if such a test input does not exist, it returns a suitable refinement predicate that is used by RefineGraph to eliminate the trace from the region graph. The pseudocode for ExtendFrontier is given in Algorithm 3.4.

ExtendFrontier starts by having the trace τ_c symbolically executed by calling ExecuteSymbolic. The call results in the path constraint ϕ, that models the requirements for executing the trace τ_c up to and including the frontier in the program P. The path constraint is passed to a SAT solver with a call to the IsSAT procedure. The SAT solver attempts to find values for all the variables referenced in the constraint, such that the path constraint ϕ is satisfied. If the SAT solver succeeds in finding such values, then those are assigned to t. The test input t includes all the arguments needed to call P such that the trace τ_c is executed. The test input t is returned as $\langle t, \textbf{true} \rangle$.

If the SAT solver was unable to satisfy the constraint then UNSAT is returned by IsSAT. The procedure RefinePred is then called to compute a suitable refinement predicate ρ, which is returned as $\langle \text{UNSAT}, \rho \rangle$ from ExtendFrontier.

The smaller sub procedure IsSAT is described next whereas the larger ExecuteSymbolic is described in Section 3.5. RefinePred is described in Section 3.6.



(a) Before refinement (b) After refinement (c) New trace

Figure 3.7: (a) shows an example region graph before refinement. DASH tries to find a test that crosses the frontier marked with the red edge into the error region but the trace is infeasible and the graph is refined. (b) shows that region 2 has been split and $\rho = 0 < a$ has been added to right region and $\neg\rho$ has been added to the left region. (c) shows the a new trace found in the next iteration. The frontier edge has been pushed backwards.

Algorithm 3.4 ExtendFrontier(τ_c, P)

Returns:
$\langle t,$ **true**$\rangle$, if the frontier can be extended; or
$\langle$UNSAT$, \rho\rangle$, if the frontier cannot be extended

1: $\phi :=$ ExecuteSymbolic(τ_c, P)
2: $t :=$ IsSAT(ϕ, P)
3: **if** $t =$ UNSAT **then**
4: $\quad \rho :=$ RefinePred(τ_c)
5: **else**
6: $\quad \rho :=$ **true**
7: **end if**
8: **return** $\langle t, \rho\rangle$

3.4.1 IsSAT

IsSAT takes as input the path constraint ϕ and the procedure P. It then tries to find an assignment of values to the variables in the constraint, such that the constraint is satisfied. This is completed by using a theorem prover. The DASH implementation in YOGI and our own implementation of DASH use the Z3 theorem prover[1].

If the constraint can be satisfied, then the values of all the variables are extracted and returned. If the constraint cannot be satisfied, then UNSAT is returned to signal that no assignment to the variables could satisfy the constraint.

Sometimes the parameters of procedure P are not mentioned in the path

[1] Z3 can be found at `http://z3.codeplex.com/`

constraint ϕ. This occurs when a parameter has not been used in any **assume** statements in the trace. However, an argument value is required for RunTest to be able to concretely execute P. For this case, IsSAT is given the procedure P such that it can find the missing parameters and give them a default value of zero.

3.5 ExecuteSymbolic

ExecuteSymbolic is responsible for executing a trace τ_c symbolically and thereby computing the path constraint ϕ, that should be feasible if and only if it is possible to execute τ_c, including the frontier, in P. ExecuteSymbolic maintains two pieces of information:

- A map $\mathcal{S}$ from symbolic variables to symbolic expressions. The notation $\mathcal{S} := \mathcal{S}[f \mapsto q]$ overwrites $\mathcal{S}$'s entry for f, such that f now has the value q.
- A path constraint ϕ, which accumulates individual path constraints and region predicates that are encountered during symbolic execution of τ_c.

The pseudocode for ExecuteSymbolic is provided in Algorithm 3.5. The algorithm has two main parts: 1) initialization (lines 1-3) and 2) execution of the trace τ_c (lines 4-13). As can be seen, the auxiliary sub procedure SymbolicEval is called throughout the code. The next section describes SymbolicEval and thereafter the two main parts of ExecuteSymbolic are presented in detail.

Algorithm 3.5 ExecuteSymbolic($\tau_c = \langle S_0, \ldots, S_k \rangle, P$)

Returns: ϕ, the path constraint for reaching and crossing the frontier

```
 1: let ⟨ρ_0, _⟩ = S_0
 2: S := [v ↦ v_0 | v ∈ params(P)]
 3: φ := SymbolicEval(ρ_0, S)
 4: for i = 0 to k − 1 do
 5:     op := Op(S_i, S_{i+1})
 6:     match op
 7:         case(v := e):
 8:             S := S[v ↦ SymbolicEval(e, S)]
 9:         case(assume c):
10:             φ := φ ∧ SymbolicEval(c, S)
11:     let ⟨ρ_{i+1}, _⟩ = S_{i+1}
12:     φ := φ ∧ SymbolicEval(ρ_{i+1}, S)
13: end for
14: return φ
```

3.5.1 SymbolicEval

SymbolicEval takes two arguments, where the first is an expression, which can be a region predicate, a conditional expression, or an arithmetic expression. The second argument is the symbolic map $\mathcal{S}$ which functions as is the environment that the first argument is to be evaluated under. The return value is a symbolically evaluated expression.

Example arguments to SymbolicEval could be the region predicate or conditional expression $a > 2$ and the map $\mathcal{S} = \{a \mapsto 4\}$. In this example SymbolicEval will replace a with 4, returning the predicate $4 > 2$.

An arithmetic example could be the expression $x + 1$ and the map $\mathcal{S} = \{x \mapsto (a + 4) * 3\}$. When the expression is symbolically evaluated it becomes $(a + 4) * 3 + 1$. Thus, SymbolicEval simply substitutes variables mentioned in an expression with their symbolic expressions in the map $\mathcal{S}$.

3.5.2 Initialization of symbolic execution

Initialization of ExecuteSymbolic is completed by the lines 1-3 in Algorithm 3.5. The region predicate ρ_0 is unpacked from region S_0 in line 1. Initial symbolic variable v_0 are assigned to each of the input variables $v \in \mathsf{params}(P)$ in line 2. Line 3 assigns the symbolically evaluated region predicate of S_0 to the path constraint ϕ. Notice that in DASH$_{int}$, the predicate attached to region S_0 is always **true** due to how RefineGraph handles refinement of the initial region described in Section 3.3.5. However, in DASH$_{call}$ initial regions can have non-**true** predicates assigned to them, and we therefore symbolically executes them in DASH$_{int}$ as well.

3.5.3 Executing the trace τ_c

The **for** loop in lines 4-13 is responsible for executing τ_c, which includes the frontier. If the statement is an assignment statement v := e, then e is evaluated by SymbolicEval and the resulting expression is used to update the symbolic map $\mathcal{S}$ for the variable v. This occurs in lines 7-8. Lines 9-10 detect if the statement is an **assume** c statement, in which case the condition c is symbolically evaluated by SymbolicEval and the result is added to the path constraint ϕ. Additionally, all region predicates on the path are evaluated and added to the path constraint ϕ in line 12.

The path constraint ϕ is returned in line 14. At this point ϕ is the full path constraint for following the complete trace τ_c. One should notice that if the frontier edge in τ_c is not included in ϕ, then ϕ is guaranteed to be satisfiable. The reason is that the trace τ_c up until the frontier follows a concrete execution, which assures that the path constraint is satisfiable. An example of symbolic execution is presented in the next section.

3.5.4 Examples of symbolic execution

This section contains two examples for symbolic execution. The first example emphasizes what the symbolic map $\mathcal{S}$ contains and how it is updated. The second example executes a trace and constructs the path constraint ϕ for it.

Symbolic map example

The first example is the symbolic execution of the code in Figure 3.8a. Figure 3.8b contains a table describing the symbolic map after each statement. The first entry in the table shows the initialization of the symbolic map $\mathcal{S}$. The parameter

```
1  int inc(int a)
2  {
3    int q := a;
4    int r := q + 1;
5    return r;
6  }
```

(a) Example program

Line number	$\mathcal{S}$ after executing statement
2 *(initialization)*	$\{a \mapsto a_0\}$
3	$\{a \mapsto a_0, q \mapsto a_0\ \}$
4	$\{a \mapsto a_0, q \mapsto a_0, r \mapsto a_0 + 1\ \}$
5	$\{a \mapsto a_0, q \mapsto a_0, r \mapsto a_0 + 1\ \}$

(b) Memory map S

Figure 3.8: (a) shows an example program and (b) shows the contents of the symbolic memory map $\mathcal{S}$ after each line has been symbolically executed.

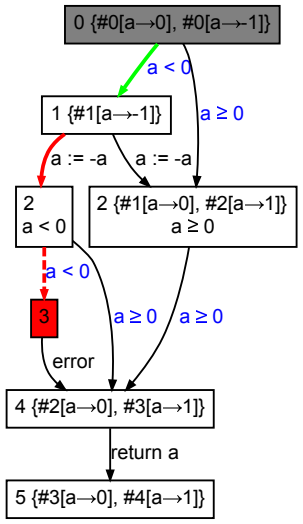


Figure 3.9: The trace used as an example to show symbolic execution. The green part of the trace comes from following a concrete execution whereas no concrete states have crossed the red frontier statement. The red dashed line originates from the abstract error path, from which the trace was generated and is included only to show the direction the path was headed.

a is given the initial symbolic variable a_0. When the assignment in line 3 is executed, the symbolic map is updated by adding the mapping $q \mapsto a_0$. When executing `r := q + 1`, the symbolic map is updated again such that $r \mapsto a_0 + 1$. Notice that q was symbolically executed, inserting the value a_0 in its place.

Symbolically executing a trace

This example will symbolically execute a trace in the `abs` procedure. The trace executed is $\tau_c = \langle 0, 1, 2{:}a{<}0 \rangle$, as shown in Figure 3.9.

First, initialization is performed. The first step is to initialize the symbolic map $\mathcal{S}$ with initial symbolic variables for each parameter. The `abs` procedure only takes a as an input parameter. The symbolic map $\mathcal{S}$ is therefore initialized to $\{a \mapsto a_0\}$. The next step is to symbolically evaluate the region predicate for region 0, which is **true**. After symbolic evaluation the expression is still **true**,

which is the value ϕ is initialized to in line 3 of ExecuteSymbolic.

The first iteration of the **for** loop starts by executing the green edge $(0, 1)$, which includes an **assume** $a < 0$ statement. Symbolically evaluating the expression yields $a_0 < 0$. The expression is added to the path constraint ϕ which becomes **true** $\wedge\ a_0 < 0$. The last step of the **for** loop is to symbolically evaluate the region predicate from region 1, which is **true** and adding it to ϕ yields **true** $\wedge\ a_0 < 0 \wedge$ **true**.

The second iteration of the **for** loop executes the edge $(1, 2{:}a{<}0)$ with the assignment a := −a. SymbolicEval symbolically evaluates the expression $-a$, which results in $-a_0$ and assigning it to a updates the symbolic map $\mathcal{S}$ such that it becomes $\{a \mapsto -a_0\}$. The region predicate $a < 0$ is then symbolically evaluated to $-a_0 < 0$ and added to ϕ. The final path constraint ϕ returned by ExecuteSymbolic is **true** $\wedge\ a_0 < 0 \wedge$ **true** $\wedge\ -a_0 < 0$. Simplifying the path constraint ϕ, by removing the redundant **true** predicates from it, yields $a_0 < 0 \wedge -a_0 < 0$. This captures the constraints that need to be satisfied by a given value for a_0 if the trace seen in Figure 3.9 is to be followed.

3.6 RefinePred

RefinePred is called by ExtendFrontier when the path constraint ϕ for a trace τ_c has been found UNSAT. Thus the trace is infeasible and RefinePred's task is to find a suitable refinement predicate that can be used by RefineGraph.

RefinePred takes as input the trace τ_c that has been found infeasible. This implies that the frontier could not be crossed. RefinePred must find a suitable refinement predicate ρ that eliminates any future concrete executions along τ_c. The refinement predicate is used to split region S_{k-1}, the region before the frontier, into two new regions S^*_{k-1} and S^{**}_{k-1} as described in Section 3.3.5. The pseudocode for RefinePred is provided in Algorithm 3.6. Lines 4-8 provide a loop optimization, which is explained in Section 3.6.3.

Algorithm 3.6 RefinePred($\tau_c = \langle S_0, \ldots, S_{k-1}, S_k \rangle$)

Returns: ρ, a suitable predicate for refinement.

```
1: let ⟨_, states_{k−1}⟩ = S_{k−1}
2: let ⟨ρ_k, _⟩ = S_k
3: op := Op(S_{k−1}, S_k)
4: if op matches assume c then
5:     if k > 1 ∧ ∀s ∈ states_{k−1} : Eval(¬ρ_k, s) = true then
6:         return ρ_k
7:     end if
8: end if
9: return WP(op, ρ_k)
```

A suitable predicate ρ must have the following characteristics with respect to the trace τ_c, that has the frontier edge (S_{k-1}, S_k), before it is a good candidate for a refinement predicate:

1. All concrete executions that follow τ_c must satisfy $\neg\rho \wedge \rho_{k-1}$ at S_{k-1}. S_{k-1} is the region before the frontier and ρ_{k-1} is the region predicate of S_{k-1}.

2. It is not possible to reach the region after the frontier S_k for any execution that reaches S^*_{k-1}, which has the predicate $\neg\rho \wedge \rho_{k-1}$. Therefore the edge between S^*_{k-1} and S_k can be removed.

Ignoring the loop optimization in lines 4-8, RefinePred always returns the weakest precondition computed by the procedure WP. The next section describes how the weakest precondition ρ_{wp} is computed. After having introduced the weakest precondition we argue that ρ_{wp} is a suitable predicate.

3.6.1 Weakest precondition computed by WP

The goal of the WP procedure is to compute the weakest precondition for a postcondition p and an operation *op*. It has two properties:

- It is a precondition, such that if $\rho_{wp} = \mathsf{WP}(op, p)$ holds before *op* then p holds after *op* has been executed.
- It is the weakest of all preconditions, such that for all other preconditions P' that holds for *op* and p, then P' implies ρ_{wp}.

There are only two possible operations in DASH$_{int}$, namely assignments v := e and **assume** c statements.

For assignments, WP is calculated by replacing the assigned variable v in p with e, which is written as p[e/v]. Calculating the weakest precondition for the assignment x := 4 with the postcondition $z > x$ proceeds like:

$$\begin{aligned} \mathsf{WP}(\mathrm{x} := 4, z > x) &\longrightarrow (z > x)[4/x] \\ &\longrightarrow z > 4 \end{aligned}$$

For **assume** c statements, the condition c is added to p: $\mathsf{WP}(\textbf{assume}\ \mathrm{c}, p) = \mathrm{c} \wedge p$. For example:

$$\mathsf{WP}(\textbf{assume}\ \mathrm{x} > 4, x < 10) \longrightarrow x > 4 \wedge x < 10$$

Notice that the DASH article uses a special procedure WP_α where aliasing is considered. DASH$_{int}$ does not support pointers and therefore aliasing cannot occur which simplifies WP considerably.

The pseudocode for WP is shown in Algorithm 3.7. We argue in the next section that the weakest precondition ρ_{wp} is a suitable predicate.

Algorithm 3.7 WP(op, p)

Returns: ρ_{wp}, the weakest precondition for *op*, such that p evaluates to **true** after executing *op*.

```
1: match op
2:     case(v := e):
3:         return p[e/v]
4:     case(assume c):
5:         return c ∧ p
```

3.6.2 The weakest precondition is a suitable predicate

The classical weakest precondition ρ_{wp} can be used as a suitable predicate:

1. We know that no execution of τ_c is able to cross the frontier since we were not able to find a test input with the SAT solver in ExtendFrontier. Assume for contradiction that there exists an execution in τ_c that satisfies ρ_{wp} at S_{k-1}, then that execution would be able to cross the frontier, by the definition of ρ_{wp}. The SAT solver would then have found the input, but the SAT solver did not find such an input. Therefore no concrete execution in τ_c satisfies ρ_{wp} and consequently all concrete executions that follow τ_c must satisfy $\neg\rho_{wp}$ at region S_{k-1}.

2. Let s be a state that satisfies $\neg\rho_{wp}$ and let ρ_k be the region predicate on region S_k. Assume for contradiction that ρ_k is satisfied after executing the frontier operation *op*, when in the state s. Let then P_s be a predicate that is only satisfied by s. P_s is then a precondition for *op* and ρ_k, due to the assumption that s satisfies ρ_k after having executed *op*. Since P_s is a precondition, and ρ_{wp} is the weakest precondition, then by the definition of a weakest precondition, P_s implies ρ_{wp}, which means that s must satisfy ρ_{wp}. This is a contradiction, since we assumed that s satisfied $\neg\rho_{wp}$. Therefore no state can satisfy $\neg\rho_{wp}$ and satisfy ρ_k after executing *op*, which was the frontier operation. The result is that there are no possible transitions from region S_{k-1}, with the predicate $\neg\rho_{wp}$ added, to region S_k.

The definition of suitable predicates mentions the region predicate ρ_{k-1} from region S_{k-1}. However, ρ_{k-1} has been left out of the argumentation for ρ_{wp} since any state that reaches S_{k-1} must trivially satisfy ρ_{k-1} because it is the region predicate.

The predicate ρ_{wp}, computed by the procedure WP, is therefore a suitable predicate and can be returned for use in RefineGraph.

3.6.3 RefinePred loop optimization

As seen, the weakest precondition for an **assume** c statement WP(**assume** c, p) is c $\wedge$ p. During refinement of a loop, the condition of the loop or any other condition inside the loop will thus always be added for each round of the loop that the trace follows. However, there are properties where the work performed inside the loop is irrelevant for the safety property being checked. DASH may try more and more iterations of the loop to reach the safety property, but as mentioned, sometimes the loop is irrelevant for the safety property. This can keep DASH refining infinitely as we describe in Section 3.8.8.

To overcome this special case, DASH uses a trick. When computing the refinement predicate for an **assume** c statement, DASH tries to see if the region predicate ρ_k from region S_k is strong enough to be used as a refinement predicate. It thus tries to see if the **assume** condition c can be ignored.

Checking if the region predicate is strong enough is performed by evaluation. DASH checks if each of the concrete states contained in S_{k-1} satisfies the condition

$\neg\rho_k$. If this is the case, then the region predicate may be able to satisfy the first condition of a suitable predicate. Only the currently known states in S_{k-1} are checked to see if they are moved to S^*_{k-1} with $\neg\rho_k$, and thus not all states that can be obtained by following the trace τ_c are guaranteed to be moved. Thus the region predicate is not guaranteed to be a suitable predicate. The second condition of a suitable predicate is satisfied. No state that satisfies $\neg\rho_k$ is able to satisfy ρ_k. Thus, the region after the frontier cannot be reached from S^*_{k-1} with $\neg\rho_k$ added.

The loop optimization is disallowed when refining an outgoing edge from the initial region. The reason why is described in Section 3.8.9. The loop optimization could also result in an infinite refinement loop in DASH$_{call}$, if the loop optimization is allowed on the initial region. This problem is described in Section 4.7.1.

The requirements for using the loop optimization is checked in RefinePred lines 4 and 5, and if satisfied the region predicate is returned in line 6.

There are of course cases where the **assume** c condition is required for the analysis. If this is the case, then DASH will find it at a later point. This is evident since if the same edge becomes a frontier at a later point, then a state must exist in the region before it. The state must satisfy the region predicate ρ_k that was previously returned by the loop optimization. Thus, this time the region predicate ρ_k cannot be strong enough to move all states to S^*_{k-1}. RefinePred cannot use the loop optimization and is forced to use the weakest precondition returned by WP wherein the **assume** condition is added.

3.7 Complete example

At this point all the procedures that DASH$_{int}$ uses have been presented. This section describes in detail how the abs procedure, shown again for convenience in Figure 3.10a, is analyzed by DASH$_{int}$. As should already be evident, abs contains an overflow error where passing the smallest value of an integer to abs will result in an error.

Initially the DASH algorithm starts by constructing the region graph for the abs procedure. The resulting graph is shown in Figure 3.10b.

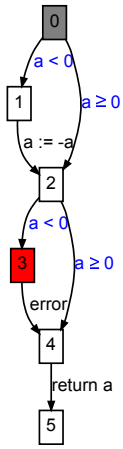
3.7.1 First iteration – Execute a test

DASH$_{int}$ starts by trying to find an abstract error path to an error region and finds $\tau = \langle 0, 2, 3\rangle$ which is depicted in Figure 3.10c as red dashed lines. The abstract error path τ is converted by ConvertToRegionTraceWithAbstractFrontier to a trace τ_c that follows a concrete execution trace. Such a concrete execution does not yet exist and as a special case, ConvertToRegionTraceWithAbstractFrontier returns the trace $\tau_c = \langle 0, 2\rangle$ where the initial region 0 is assumed to have a concrete state even though it does not. The trace is depicted in Figure 3.10d. Notice that the dashed red line is not part of τ_c, but it shows the path from which τ_c was generated.

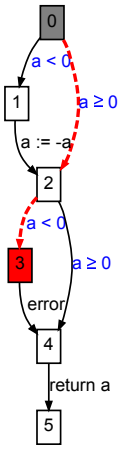
At this point DASH$_{int}$ calls ExtendFrontier with the trace τ_c which in turn calls ExecuteSymbolic. ExecuteSymbolic starts by assigning all input variables a

```
int abs(int a)
{
  if(a < 0)
    a = −a;
  assert a ≥ 0;
  return a;
}
```

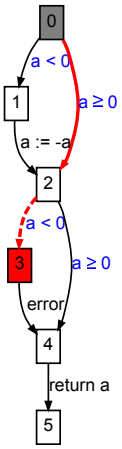
(a) abs procedure



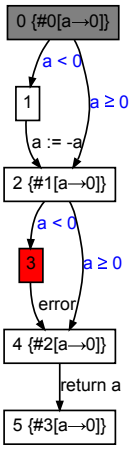
(b) Initial



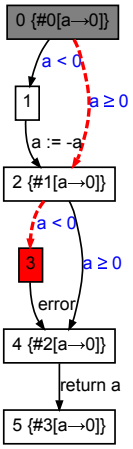
(c) Error path



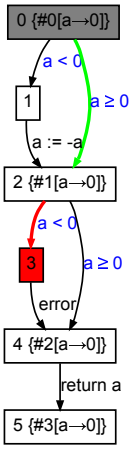
(d) Trace



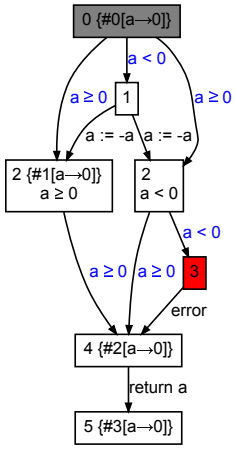
(e) First test



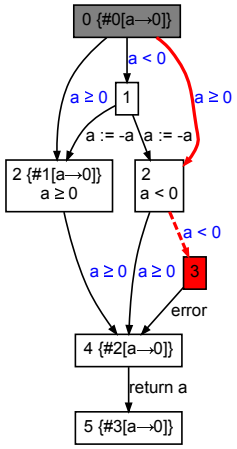
(f) Error path



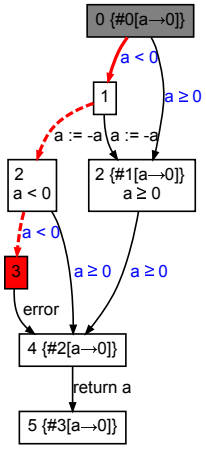
(g) Trace



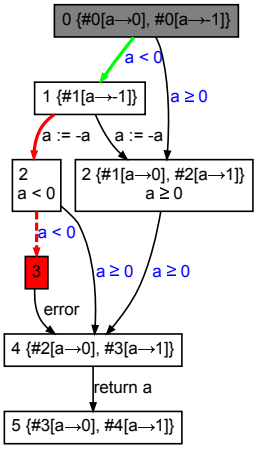
(h) Refinement



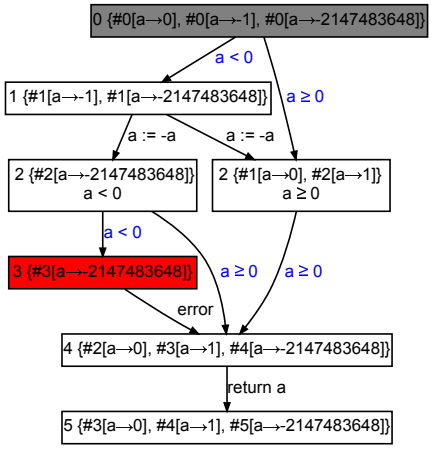
(i) Trace



(j) 4^{th} iteration



(k) 5^{th} iteration



(l) Error region reached

Figure 3.10: Example graphs for the abs procedure.

symbolic free variable, such that the symbolic map contains the mapping $a \mapsto a_0$. When symbolically executing τ_c it evaluates the **assume** $a \geq 0$ statement and inserts a_0 in place of a. Thus ExecuteSymbolic returns the predicate $\phi = a_0 \geq 0$. No region predicates are added since these are all **true**.

DASH$_{int}$ now needs to check if the predicate ϕ can be satisfied. It uses a SAT solver via the IsSAT call. Since $a_0 \geq 0$ is satisfiable IsSAT returns a valuation of the variables, and in this case it returns $[a_0 \mapsto 0]$. This makes ExtendFrontier return with the result $\langle [a_0 \mapsto 0], \textbf{true} \rangle$.

DashLoop checks whether the test input could be generated by ExtendFrontier, and in this case it could. The test input $[a_0 \mapsto 0]$ is given to RunTest, which runs an instrumented version of abs that records the concrete states found during execution.

RunTest starts by adding the initial state to the first region. From the initial region it is possible to transition to either region 1 or region 2. RunTest looks at the **assume** statements and the region predicates in the regions and evaluates the predicates. Since only $a \geq 0$ evaluates to **true**, the next region that RunTest transitions to must be region 2. RunTest continues the execution by finding the regions that are reached and adding the concrete state to them, which results in Figure 3.10e. Since a test was executed IsErrorRegionReached is called to check if the error region was reached during testing. It returns **false** since no states are attached to the error region. This concludes the first iteration.

3.7.2 Second iteration – Infeasible trace refined

We are now at a point where DASH$_{int}$ have expanded the under-approximation of the program by executing a concrete test. As we shall see in this iteration, the under-approximation dictates where the graph is refined by having moved the frontier forward.

DASH$_{int}$ starts by finding the same abstract error path as in the first iteration, which is depicted in Figure 3.10f. The only difference is that this graph contains concrete states, and this affects how ConvertToRegionTraceWithAbstractFrontier creates the trace τ_c. The first region with states, found by searching backwards in the abstract error path, is region 2. The concrete execution that leads to region 2 is spliced together with the step $\langle 2, 3 \rangle$ which is the frontier, yielding $\tau_c = \langle 0, 2, 3 \rangle$ depicted in Figure 3.10g. The green part follows a concrete execution while the red edge is the frontier. Notice that the frontier has been pushed forward by the RunTest execution made in the first iteration.

At this point DASH$_{int}$ calls ExtendFrontier which in turn calls ExecuteSymbolic. The initial value a_0 is then added for the input variable a such that the symbolic map contains $a \mapsto a_0$. The first step in the trace adds $a_0 \geq 0$ to ϕ and the second step adds $a_0 < 0$. These predicates are clearly in conflict and IsSAT therefore returns UNSAT signaling that the predicates are unsatisfiable. The trace τ_c was therefore infeasible and no input could be generated that would follow it. Since the path leading up to the frontier is certainly feasible, as it followed a concrete execution, we know that it is the frontier edge that made the trace infeasible. At this point RefinePred is called to generate a refinement predicate. RefinePred calls WP with the **assume** $a < 0$ statement and region 3, which is the region

after the frontier edge. WP now finds the weakest precondition for the region predicate **true** in region 3, that holds when executing the statement **assume** $a < 0$. In this case the weakest precondition is the **assume** statement predicate, namely $\rho = a < 0$. Notice that the loop optimization mentioned in Section 3.6 was not used in this case, since the region predicate for region 3 was **true**, which is not strong enough to be used as a refinement predicate.

The predicate ρ found by RefinePred is returned to DashLoop which calls RefineGraph. RefineGraph splits region 2 into two new regions, one with ρ added and one with the negation $\neg\rho$ added. The region with $\neg\rho$ keeps all concrete states from region 2 and the edge between it and region 3 is removed. The resulting graph can be seen in Figure 3.10h.

3.7.3 Third iteration – Refinement

The graph has been refined and an infeasible trace has been removed, however there are still paths that lead to the error region. We have shown the path, the trace and the resulting region graph for the first two iterations of DASH$_{int}$. For brevity the next iterations will only show the trace.

The path found by the procedure FindAbstractErrorPath is $\tau = \langle 0, 2{:}a{<}0, 3\rangle$ that is converted to the trace $\tau_c = \langle 0, 2{:}a{<}0\rangle$ depicted in Figure 3.10i. This trace is infeasible since the first edge contains an **assume** $a \geq 0$ and the predicate in region 2 specifies $a < 0$, which is the negation. For this case WP returns $a < 0 \wedge a \geq 0$ as the refinement predicate, which is equivalent to **false**. The loop optimization is not attempted to be used by RefinePred, since the edge is going out from the initial region. However, when refining the initial region, as performed by RefineGraph, the edge is simply removed and the refinement predicate is not used. The refined graph, with the frontier edge removed, can be seen in Figure 3.10j. Notice that the trace and path for the next iteration is also shown.

3.7.4 Fourth iteration – Execute a test

In this iteration, ConvertToRegionTraceWithAbstractFrontier returns the trace $\tau_c = \langle 0, 1\rangle$ shown in Figure 3.10j. ExecuteSymbolic returns the path constraint $\phi = a_0 < 0$, which originates from the **assume** $a < 0$ statement between the regions.

When the path constraint is given to IsSAT it returns the solution $[a_0 \mapsto -1]$ which will cross the frontier. Afterwards, DASH calls RunTest that runs the concrete test. When RunTest reaches region 1, the statement a := −a is executed and a becomes 1. Therefore RunTest ends up in region 2:$a{\geq}0$, which leads away from the error region. This is not a problem, since we only required the test to reach the region after the frontier. The resulting graph, where RunTest has completed its execution, can be seen in Figure 3.10k. The error region has not been reached by concrete execution and therefore DASH continues to the next iteration.

3.7.5 Fifth iteration – Reach the error region

The previous iteration executed a test which added a concrete state to region 1 that results in the frontier being pushed forward. The trace found is $\tau_c = \langle 0, 1, 2{:}a{<}0 \rangle$, which is depicted in Figure 3.10k.

The trace is symbolically executed by passing it to ExecuteSymbolic. Notice that this exact trace was used as an example for symbolic execution in Section 3.5.4. The path constraint ϕ generated in the example, and thus for this trace, is $a_0 < 0 \wedge -a_0 < 0$.

When the path constraint is given to IsSAT it finds the solution $[a_0 \mapsto -2{,}147{,}483{,}648]$, which is the smallest possible value of an 32 bit integer variable. One would think that $-(-2{,}147{,}483{,}648)$ is positive with the value of 2,147,483,648, but the constraint requires that it is negative. The problem is that the largest possible value an integer variable can take is one less of 2,147,483,648. The expression $-a_0$ actually overflows with exactly one, such that $-a_0 = a_0$. Thus $a_0 \mapsto -2{,}147{,}483{,}648$ is a valid solution.

Calling RunTest executes the test and the resulting graph is shown in Figure 3.10l. It can be seen that the error region now contains a concrete state and IsErrorRegionReached thus returns **true**, making DASH report that the error region is reachable with an input value of $-2{,}147{,}483{,}648$.

This concludes the complete example and the description of DASH$_{int}$. DASH$_{int}$ is able to prune edges from the graph, such that it is not possible to reach the error region, and the program is then correct, or it might be able to direct a test such that the error region is reached, in which case the program violates the safety property with which it was instrumented. The last option is that DASH$_{int}$ keeps analyzing forever.

The next sections describe the challenges and modifications we have made while implementing DASH$_{int}$.

3.8 Challenges and modifications

This section describes challenges we have encountered while implementing DASH$_{int}$ and what modifications we have made to it compared to how DASH is presented in the DASH article [A2].

3.8.1 Combined data structure: Region graph

Our presentation of DASH uses a region graph for both the over- and under-approximation. The DASH article keeps the concrete states in a so-called *forest*. The reason that they term it a forest, is probably because they have nondeterminism in their programs and as such two executions with the same input variables may follow different execution paths. Since our programs do not contain any non-determinism we do not need a forest.

When converting an abstract error path to a trace we need to find the last region that contains a concrete state. This means that if we kept states in a separate data structure we would somehow need to know which region a given state belongs to. It therefore did not make sense to keep them divided from

each other. Our presentation has been simplified significantly by having states directly attached to the regions in the region graph.

In fact, without non-determinism, it does not make sense to execute a test with the same input values more than once, and therefore we have an error-check in our implementation that makes sure that when executing a test, then the test has never been executed before.

3.8.2 Ambiguity: Using S_i both for regions and predicates

The DASH article uses S_i both as a region, as in a trace $S_0, \ldots, S_n$, and as a predicate as in $S_0 \wedge \rho$. This is evident when looking at the use of S_i in Algorithm 3.8 and Algorithm 3.9. Since this is sometimes confusing we have removed such ambiguities from our presentation. Instead we explicitly unpack our regions S_i to retrieve the region predicate ρ_i, or the states $states_i$, using the syntax:

$$\textbf{let } \langle \rho_i, states_i \rangle = S_i$$

3.8.3 RefineGraph: UNSAT regions have incoming edges removed

The DASH article refines the region graph directly inside the DashLoop procedure. We have extracted the relevant pseudocode and presented it in Algorithm 3.8. Remember that states are not attached to the graph in the original DASH article, and as such their algorithm does not move the states.

Algorithm 3.8 $\mathsf{RefineGraph}_{original}(\rho, \tau_c = \langle S_0, \ldots, S_{k-1}, S_k \rangle, G = \langle \Sigma_{\simeq}, \rightarrow_{\simeq} \rangle)$

Returns: $\langle \Sigma_{\simeq}, \rightarrow_{\simeq} \rangle$, the refined graph.

1: $\Sigma_{\simeq} := (\Sigma_{\simeq} \setminus \{S_{k-1}\}$
2: $\quad \cup \{S_{k-1} \wedge \rho, S_{k-1} \wedge \neg\rho\}$
3: $\rightarrow_{\simeq} := (\rightarrow_{\simeq} \setminus \{(S, S_{k-1}) \mid S \in \mathsf{Parents}(S_{k-1})\})$
4: $\quad \setminus \{(S_{k-1}, S) \mid S \in \mathsf{Children}(S_{k-1})\}$
5: $\rightarrow_{\simeq} := \rightarrow_{\simeq} \cup \{(S, S_{k-1} \wedge \rho) \mid S \in \mathsf{Parents}(S_{k-1})\}$
6: $\quad \cup \{(S, S_{k-1} \wedge \neg\rho) \mid S \in \mathsf{Parents}(S_{k-1})\}$
7: $\quad \cup \{(S_{k-1} \wedge \rho, S) \mid S \in \mathsf{Children}(S_{k-1})\}$
8: $\quad \cup \{(S_{k-1} \wedge \neg\rho, S) \mid S \in (\mathsf{Children}(S_{k-1}) \setminus \{S_k\})\}$
9: **return** $\langle \Sigma_{\simeq}, \rightarrow_{\simeq} \rangle$

The pseudocode in Algorithm 3.8 is very specific in how the edges of the graph are updated. Nevertheless the graph is *not* refined like shown in Algorithm 3.8 in their DASH implementation.

We have found, by personal correspondence with the authors of DASH that they deviate from their pseudocode in two fundamental ways:

1. They simplify region predicates when the predicate is altered.

2. If a region predicate is equivalent to **false** they remove the region from the graph.

We have added those missing details to our RefineGraph pseudocode as presented earlier in Algorithm 3.3.

A question that rises is how DASH simplifies the predicates. According to the authors they use a simplifying routine in the Z3[2] theorem prover to discover if a predicate is equivalent to **false**. However, there are two such routines, one which is a fast bottom-up rewriter and a second much more expensive implementation called ctx-solver-simplify. The first can perform simple rewrites while the other is much more powerful and will simplify more complex predicates[3]. We have found that the first implementation is not good enough to simplify the predicates to **false** where needed, so they must be using the second. The DASH article is very clear in that it uses only one SAT call for each iteration. However, simplifying a predicate using ctx-solver-simplify is so slow, nontrivial and complex that we believe such a call is computationally equivalent to a SAT call. We have, in our implementation, frequently observed that ctx-solver-simplify is as slow as many of our SAT calls. Timing information is provided in Section 5.7.

The pseudocode in Algorithm 3.3 for RefineGraph calls a Simplify routine, but also uses the IsSAT routine to check if the predicate is equivalent to **false**. In our implementation we use ctx-solver-simplify for the Simplify operation. We do this since we have found that the ctx-solver-simplify routine has been able to simplify all predicates we have seen to **false** whenever IsSAT says a predicate is unsatisfiable. However, the pseudocode uses the IsSAT call since the contract for a satisfiability checker is to return UNSAT when a predicate cannot be solved. It is not clear if ctx-solver-simplify will always return **false** when a predicate is unsatisfiable. In our implementation we check that if ctx-solver-simplify does not simplify to **false**, then IsSAT has a solution to the predicate. The price is a significant slowdown of our implementation, but since this study is not about speed, we have kept it as an assertion.

3.8.4 Confusing terminology: *ordered path* and unused parts

The DASH article uses the term *ordered path* τ_o about what we call the *region trace with abstract frontier* τ_c, or in short form the *trace* τ_c. We never found a reason to why they call τ_o *ordered*. It is a list of regions that follows a concrete execution and regions from the rest of the abstract error path, but that does not make it *ordered*. We have therefore chosen to use the term *trace* instead.

We also slightly altered what a trace is. The DASH article has an ordered path $\tau_o = S_0, \ldots, S_{k-1}, S_k, \ldots S_n$ which is a concatenation of regions that follow a concrete execution $s_0, \ldots, s_{k-1}$ and the rest of the abstract error path $S_k, \ldots, S_n$. However, only the frontier edge from the abstract error path, located at (S_{k-1}, S_k), is used by DASH. We found it confusing to create a trace where the last part $S_{k+1}, \ldots, S_n$ is never used, and as such we have chosen to only include the single edge that becomes the frontier. This has simplified the pseudocode in that we do not need to call an auxiliary Frontier procedure to determine where the frontier edge is located in τ_o. The frontier edge in our trace τ_c is always the last edge.

[2] Z3 can be found at `http://z3.codeplex.com/`

[3] Answer by Z3's author on `http://stackoverflow.com/a/14058292/477854`

3.8.5 ExecuteSymbolic: assignments added to path constraints

The original version of ExecuteSymbolic, altered to how we present pseudocode, can be seen in Algorithm 3.9. Notice that they do not have local variables, only heap pointers. There are multiple differences between this code and how we presented ExecuteSymbolic in Algorithm 3.5. First of all they execute the trace up until the frontier in one loop and then execute the frontier statement for itself. We also need to do this in the DASH$_{call}$ algorithm, which is presented in Chapter 4. They return two predicates ϕ_1 and ϕ_2, but they are only used together in the call to IsSAT. There is no technical reason to keep them apart. However, for presentation purposes it could be helpful to point out to the reader that ϕ_1 is a constraint that can always be solved, since it originates from a concrete execution, but together with ϕ_2 it might be unsatisfiable. We joined them together such that $\phi = \phi_1 \wedge \phi_2$.

Algorithm 3.9 ExecuteSymbolic$_{original}(\tau_c = \langle S_0, \ldots, S_k \rangle, P)$

Returns: $\langle \phi_1, \mathcal{S}, \phi_2 \rangle$.

```
 1: S := [v ↦ v_0 | *v ∈ inputs(P)]
 2: φ_1 := SymbolicEval(S_0, S)
 3: φ_2 := true
 4: i := 0
 5: while i ≠ k − 1 do
 6:     op := Op(S_i, S_{i+1})
 7:     match op
 8:         case(*m=e):
 9:             S := S[SymbolicEval(m, S) ↦ SymbolicEval(e, S)]
10:         case(assume c):
11:             φ_1 := φ_1 ∧ SymbolicEval(c, S)
12:     i := i + 1
13:     φ_1 := φ_1 ∧ SymbolicEval(S_i, S)
14: end while
15: op := Op(S_{k−1}, S_k)
16: match op
17:     case(*m := e):
18:         φ_2 := φ_2 ∧ *(SymbolicEval(m, S)) = SymbolicEval(e, S)
19:         S' := S[SymbolicEval(m, S) ↦ SymbolicEval(e, S)]
20:     case(assume c):
21:         φ_2 := φ_2 ∧ SymbolicEval(c, S)
22:         S' = S
23: φ_2 := φ_2 ∧ SymbolicEval(S_k, S')
24: return ⟨φ_1, S, φ_2⟩
```

The predicate that the authors of DASH gives to IsSAT is $\mu = \phi_1 \wedge \mathcal{S} \wedge \phi_2$ where every entry in the symbolic map $\mathcal{S}$ is seen as an equality predicate. Because the map $\mathcal{S}$ is from before the frontier edge has been symbolically executed, they add the equality predicate for frontier assignments in line 18 in Algorithm 3.9. It makes no sense to include $\mathcal{S}$ when we only consider local variables, and as such the constraint we give to IsSAT is $\mu = \phi_1 \wedge \phi_2$ which is equivalent to our definition of ϕ. We have not yet discovered why $\mathcal{S}$ in μ is needed, but from personal correspondence with the authors, we believe it is related to heap aliasing constraints.

3.8.6 Problem: Should the trace τ_c follow the path τ

The quote below is from the DASH article and it explains how they convert what we call an error path into a trace, which is what ConvertToRegionTraceWithAbstractFrontier does:

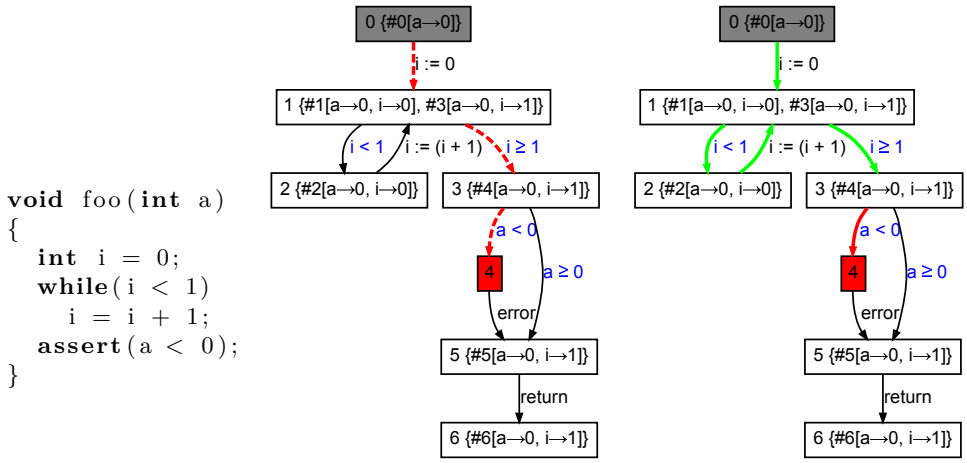
> "The auxiliary function GetOrderedAbstractTrace converts an arbitrary abstract trace τ into an ordered abstract trace τ_o. This works by finding the last region in the abstract trace that intersects with the forest F, which we call S_f. The algorithm picks a state in this intersection and follows the *parent* relation back to an initial state. This leads to a concrete execution $s_0, s_1, \ldots, s_{k-1}$ that corresponds to an abstract trace $S_0, S_1, \ldots, S_{k-1}$ where $S_{k-1} = S_f$. By splicing together this abstract trace and the portion of the abstract error trace from S_f to S_n, we obtain an ordered abstract error trace. It is crucial that the ordered abstract error trace follows a concrete execution up to the frontier, as this ensures that it is a feasible trace up to that point."[4]

From the quote it seems rather clear that to convert an error path to a trace one goes backwards in the path and finds the first region that contains a concrete state. However, we have seen instances where the trace found $s_0, s_1, \ldots, s_{k-1}$ does not follow the error path $S_0, S_1, \ldots S_n$.

A small example is shown in Figure 3.11. The path found in Figure 3.11b is $\tau = \langle 0, 1, 3, 4\rangle$ while the trace follows a different route $\tau_c = \langle 0, 1, 2, 1, 3, 4\rangle$ shown in Figure 3.11c. It seems counterintuitive that the trace follows a different set of edges compared to the path from which it is constructed.

We were in doubt if the trace should always follow the path from which it was constructed. It led us to implement at version where the traces always followed the path. However, the implementation failed in numerous of our test cases due to timeouts and this is also evident for the path in Figure 3.11b. If the trace needs to follow the path, we can only generate a trace $\tau_c = \langle 0, 1, 3\rangle$ where the frontier is the edge $(1, 3)$. However, the trace is clearly infeasible, since it is not possible to take zero iterations of the **while** loop. Eventually the DASH algorithm will have refined the region graph such that a trace to the error region 4 can be created. However, if we change the **while** condition from $i < 1$ to $i < 1000$, then the algorithm becomes painfully slow. In this case it needs to refine the graph so many times, that only a trace that takes 1000 iterations of the loop leads to the error region. If we allow the trace to follow a path different from the abstract error path, then we can immediately generate a test by choosing the state that exists in region 3. This observation allows us to conclude that the trace τ_c is allowed to follow a different path compared to the error path τ.

[4] Page 8 in DASH [A2]



(a) Example code (b) Error path τ (c) The trace τ_c found

Figure 3.11: (a) shows a small example where the code has been borrowed from SYNERGY and altered. (b) shows a path in the region graph where one test with $a \mapsto 0$ has been executed. The trace found is shown in (c). Notice that the trace does not strictly follow the path in (b).

3.8.7 Problem: Which state to pick when creating traces

The previous section described that when converting an abstract error path to a trace, one searches back in the path to find the first region that contains concrete states. The DASH authors wrote the following about choosing a state in that region:

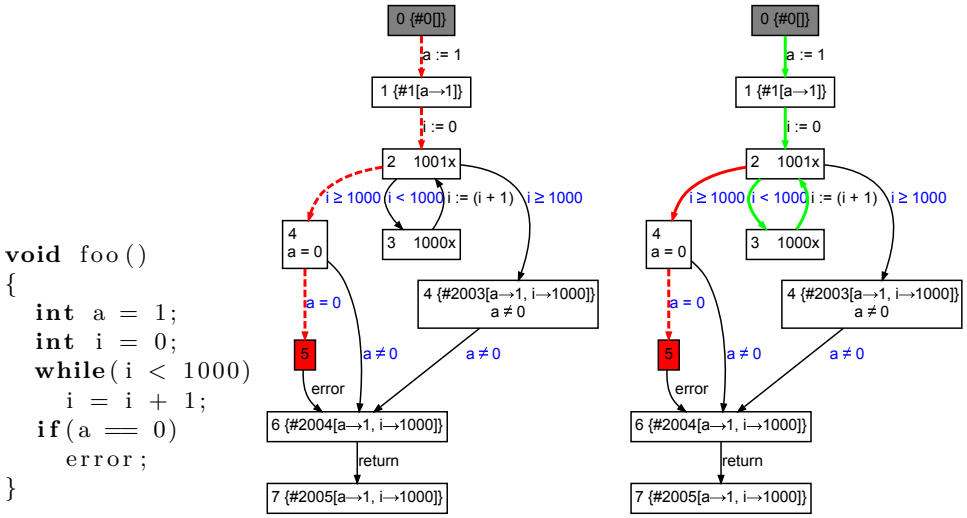
> "The algorithm **picks** a state in this intersection and follows the *parent* relation back to an initial state."[5]

Bold text has been added to highlight that the DASH article is vague in how a state is picked. If there is only a single state in a region, then it is obvious that one should use that state to construct the trace from. However, it is unclear which state should be used to construct the trace when there are multiple to pick from. From a correctness perspective it does not matter which state is picked, eventually the algorithm will lead to the same answer. We have thought of multiple different strategies to pick the state:

- Pick a random state
- Pick the last state (which our implementation does)
- Pick a state that enters the sought region

Since we did not know how to pick a state, we initially picked a random state as the state to generate the trace from. This works well for many test cases. However, there are cases where picking a random state seems counterintuitive.

[5] Page 8 in DASH [A2]



(a) Example code (b) Error path τ (c) Many choices for τ_c

Figure 3.12: Example where many different traces can be generated by picking different states of region 2 in (b).

Figure 3.12 shows such an example. The code is shown in Figure 3.12a and it should be clear that the error statement cannot be reached. At one point in DASH$_{int}$ the path in Figure 3.12b needs to converted to a trace. There are 1001 states in region 2 from which a trace can be generated. All of them are infeasible, when the frontier edge $(2, 4{:}a{=}0)$ is added, but it should be noted that one of them stands out. 1000 of the states are infeasible since they take less than the required iterations of the **while** loop. One state, the last state, takes the required 1000 iterations of the loop, but it is still infeasible because of the region predicate $a = 0$, which originates from region $4{:}a{=}0$.

In the above example it does not make a difference which state is picked, since all of them will result in the same refinement of the graph. However, it seems counterintuitive to choose traces that are clearly not feasible. We should pick a state that optimizes the chances for the trace to be feasible. Given the example we choose to always use the last state to generate the trace τ_c from. Using this implementation we get the trace $\tau_c = \langle 0, 1, 2, (3, 2)^{1000}, 4\rangle$ which is shown in Figure 3.12c. Using this heuristic we always choose to generate a trace from the newest test execution and therefore from the last iteration of loops.

One idea that we had, but did not implement, was to look through all the states and find one that enters a region with the same region number as the region we seek to enter. For example, in Figure 3.12b we seek to enter region 4 with the predicate $a = 0$. However, no states actually lead there yet (as the frontier edge leads there). Instead, we can search for a state that is a good candidate for reaching the region, and if a state enters one of the regions with the region number 4, then it is a better candidate for reaching $4{:}a{=}0$. In this way, it is only the predicates on the regions that are different. Looking for states that enters a region with the same identification number might help in cases

where it is not the last state that enters the sought region.

For the example in Figure 3.12b, it would be the last state that is picked, since it enters region 4, even though the concrete state in it satisfies $a \neq 0$, which is the negated predicate from region 4:a=0. For this strategy we might have multiple states that enters region 4, in which case one has to either pick a random one or find a clever way to choose among them. However, since we did not implement this strategy, we are not sure if it would bring improvements over the strategy that picks the last state.

3.8.8 Problem: Infinite refinement without loop optimization

Our pseudocode for RefinePred, shown in Algorithm 3.6, includes a small test to see if a weaker predicate can be used as the refinement predicate for **assume** c edges. This loop optimization is mentioned in the evaluation section of the DASH article, i.e. after the DASH authors have described their algorithm. They write the following about the optimization:

> "When faced with an if-branch in a program, DASH will perform an inexpensive test to see whether the WP_α of a weaker predicate, one that ignores the branch condition, still satisfies the template described in Figure 9. The effect of this optimization is that we avoid getting "stuck" in irrelevant loops. We have left the consideration of more thorough generalization techniques for future work."[6]

When implementing DASH$_{int}$ we initially did not include the loop optimization because it was mentioned so late in the paper. However, when we discovered simple test cases that could not be solved without it, we felt forced to see if the loop optimization would help DASH solve the test cases. It was not easy to implement, as we show in Section 3.8.9.

A test case is shown in Figure 3.13a. The code sets a variable b to zero and takes exactly one iteration in the **while** loop, which does not affect b. Afterwards the error statement is executed if b is equal to one. Since b is never affected after the initial assignment the error statement can never be reached. However, without the optimization DASH keeps analyzing infinitely and creates graphs such as the one seen in Figure 3.14b. We now show what difference the loop optimization makes to the presented example.

Figure 3.13b shows a trace that is found when analyzing the code in Figure 3.13a. So far the loop optimization has not made any differences. For this particular trace the refinement predicate returned by RefinePred differs when the loop optimization is enabled.

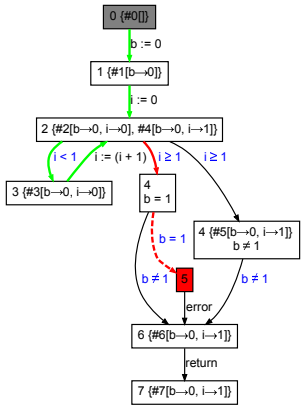
Without the loop optimization, RefinePred returns the refinement predicate $b = 1 \wedge i \geq 1$. This refinement predicate contains both the region predicate $b = 1$ and the **assume** $i \geq 1$ condition. The graph returned by RefineGraph, using the refinement predicate, is shown in Figure 3.13c.

When the optimization is enabled the refinement predicate returned by RefinePred is only the region predicate $b = 1$. It thus ignores the **assume** $i \geq 1$

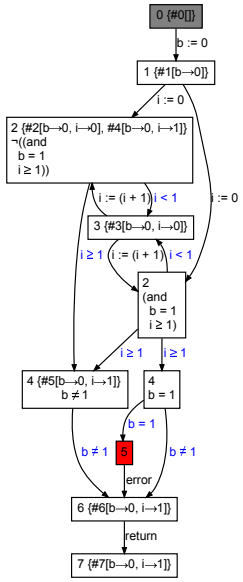
[6] Page 12 in DASH [A2]

```
void test()
{
  int b = 0;
  int i = 0;
  while (i < 1)
    i++;
  if (b == 1)
    error;
}
```

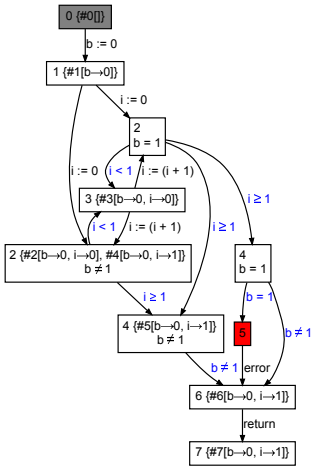
(a) Example code



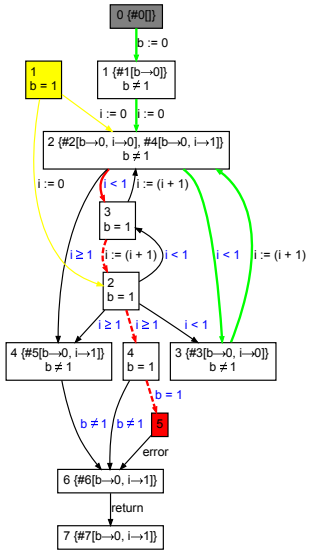
(b) Before



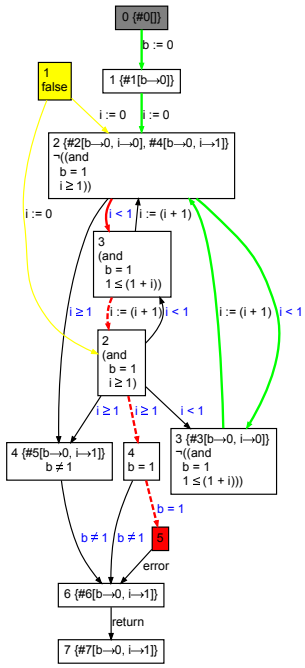
(c) After without optimization



(d) After with optimization



(e) Later with optimization



(f) Later without optimization

Figure 3.13: Region graphs showing how the loop optimization helps DASH.

condition. The optimization checks that the region predicate $b = 1$ is "strong enough" to be used as a refinement predicate. When the region before the frontier is split, the region that keeps the frontier edge must not contain any concrete states. This is to ensure progress is achieved. Using the weaker predicate results in the region graph in Figure 3.13d. This shows that the optimization makes a difference, but so far, in this test case, we have not seen that it makes DASH$_{int}$ terminate.

What makes DASH$_{int}$ terminate is the impact that the refinement predicate has on a later iteration. Both versions will come to a trace that results in refining region 2. The traces are shown in Figure 3.13e and Figure 3.13f, with and without the optimization, respectively. The traces are the same, both with a frontier going out of region 2 and the traces are both infeasible since b is required to be 1. The result of the two refinements are, however, very different. In this case the predicates returned by RefinePred are different but since it does not make a difference here, we assume that they are the same. We assume that the refinement predicate is only the region predicate $b = 0$. What matters here is that the predicates in region 2 are different. The one without the optimization has the predicate $\neg(b = 1 \wedge i \geq 1)$ and together with $b = 0$ it becomes more complex. However, the version with the optimization has the predicate $b \neq 0$ and together with $b = 0$ one of the split regions gets the region predicate **false**. Therefore all ingoing edges to that region can be removed, which results in Figure 3.14a. Notice that with the optimization, DASH$_{int}$ has completed the analysis. The error region is unreachable. The version without the optimization keeps trying to rule out the error region, but as Figure 3.14b shows, it tries more and more iterations of the loop, each of them being infeasible.

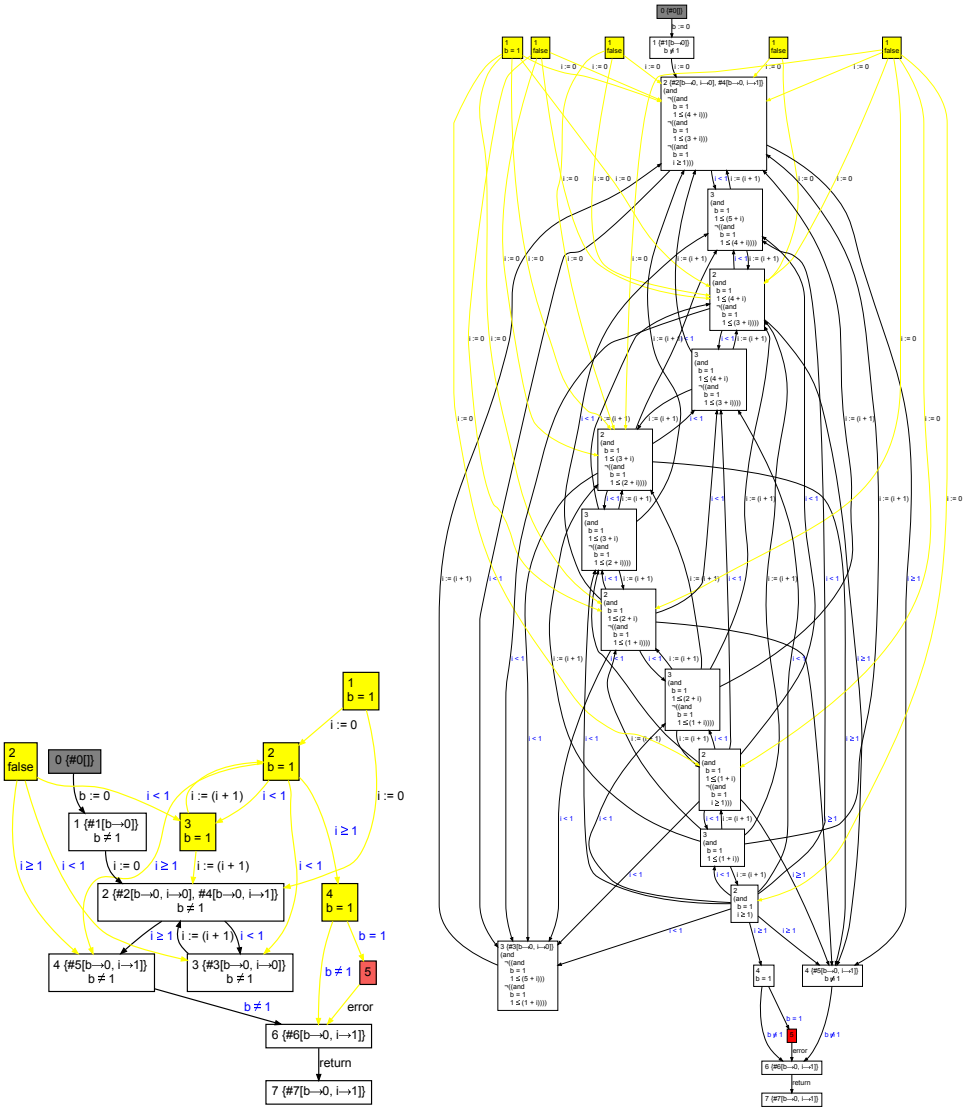
We were a bit surprised that such a simple example could not be handled by DASH without the loop optimization, which was mentioned as a side note in their evaluation section. We were especially surprised since SYNERGY, the algorithm that DASH builds on, boasts that it is very effective in generating a test for a very similar example, namely where the **while** condition is changed to $i < 1000$ and where b is a parameter of the test procedure[7]. In that example the error statement can easily be reached (by calling test with $b \mapsto 1$). By altering the example and making the error statement unreachable, we caused DASH to iterate forever.

In the example given in Figure 3.13a, the **while** loop is irrelevant to the b variable, and therefore also to the instrumented safety property being checked. However, if the loop calculates something which is then asserted afterwards, then DASH can still be brought to its knees. The example code in Figure 3.15 makes DASH analyze forever, even with the optimization.

The DASH authors already know this is a problem for the algorithm since they left this problem for future work:

> "We have left the consideration of more thorough generalization techniques for future work."[8]

[7] Page 119, Figure 1 in SYNERGY [A5]
[8] Page 13 in DASH [A2]



(a) With optimization (b) Without optimization

Figure 3.14: (a) shows that with the RefinePred loop optimization, analysis ends contrary to without the optimization, which results in larger and larger graphs such as (b).

```
void test(int a)
{
    int k = 0;
    for (int i = 1; i < a ∧ i < 10; i++)
        k += i;
    if (k == 2) // Not possible
        error;
}
```

Figure 3.15: $DASH_{int}$ never halts when analyzing the test procedure.

We also had trouble interpreting how their optimization should be implemented, as the next section shows.

3.8.9 Problem: How to implement loop optimization

The above section found that the loop optimization is important and that it is needed for even trivial cases. However, we had problems in interpreting how it should be implemented. As a reminder, they wrote:

> "When faced with an if-branch in a program, DASH will perform an inexpensive test to see whether the WP_α of a weaker predicate, one that ignores the branch condition, still satisfies the template described in Figure 9. This can be done by evaluation, and does not require a theorem prover call."[9]

Notice that for DASH_{int} WP_α is equivalent to WP since DASH_{int} do not have pointers and WP_α is used to add aliasing constraints. Thus the "the weaker predicate, one that ignores the branch condition" must in DASH_{int} be the predicate from the region after the frontier since ignoring the branch condition in WP yields the unmodified region predicate. However, there are still two main problems with their paragraph:

- Figure 9 in the DASH article is not very descriptive. It shows a split of a region into two. So the phrase "satisfies the template described in Figure 9" is not very helpful.
- It is unknown what test it is that they perform when they mention "perform an inexpensive test". It is clearly done by evaluation, but it is unknown what is evaluated.

In another section they refer to Figure 9 and define what they call a *template-based refinement*. We believe this is what they are trying to refer to when they refer to Figure 9. The next question is what their "inexpensive test" is. When defining their *template-based refinement* they define what a suitable predicate is and what requirements there are to it. We believe that their "inexpensive test" is to check if the region predicate alone is a suitable predicate. If this is the case, we have no idea why they did not write that. However, personal correspondence with the authors has verified that this is the case:

> "We just check whether the resulting predicate still satisfies the conditions of a suitable predicate. If so, we use it. Otherwise, we fall back on the actual suitable predicate obtained by taking the if-condition into account."[10]

The authors of DASH write that for ρ to be a suitable predicate it is required that *all possible concrete states obtained by executing τ_c up to the frontier must*

[9] Page 12 in DASH [A2]

[10] Personal correspondence by email on the 18th of March 2014.

belong to the region ($S \wedge \neg\rho$). Here ($S \wedge \neg\rho$) is the region with $\neg\rho$ added to it after splitting the region S, which we have called S_{k-1}.

The question is then how they perform an inexpensive test, using evaluation, to see whether the region predicate moves all the possible concrete states obtained by executing τ_c up to the frontier to the region with $\neg\rho$ added.

We can only think of one possible solution, namely that they test that the region predicate moves all *known* states to the region with $\neg\rho$ added. This is how we have implemented the optimization, which is evident in lines 4-8 of RefinePred in Algorithm 3.6. In this case it is not always the case that all the possible concrete states obtained by executing τ_c up to the frontier are moved correctly. We cannot see how we can ensure that all states that can be obtained by a trace τ_c is moved without using a theorem prover call. Thus our solution does not always return a suitable predicate. When we wrote to the authors of DASH about this problem, they replied:

> "In our implementation, we just check whether the concrete state (used to define the frontier etc.) satisfies $\neg\rho$ and that suffices for our purpose."[11]

Thus checking that all *known* concrete states are moved should suffice. The next section describes a situation where the predicate found is not a suitable predicate.

Problem with the initial region not containing any states

When we implemented the above solution, we found cases where requiring that all known states were moved to the correct region was not enough. A special case is when refining the initial region when no concrete tests have yet been executed. In this case, no concrete states exist in the initial region and all states can thus be moved by any predicate. This can be a problem because the region after the frontier might only contain **true** as its predicate. One cannot refine anything with **true**.

The problem is depicted in Figure 3.16a. To create such a graph an **if** statement must be the first statement and the condition must be equivalent to **false**. As can be seen the initial region contains no states and the error region after the frontier contains only the region predicate **true**. The trace is found infeasible. All states, which are none in this case, can be moved by the region predicate **true** from region 2, and thus RefinePred will return it as the refinement predicate. We solved this problem by disallowing the use of the loop optimization on edges going out of the initial region. The initial region is the only region that can have a frontier edge going out of it while not having any states attached to it.

Initially we solved the problem by simply requiring that at least one state had to exist in the region. However, for reasons related to DASH$_{call}$, as described in Section 4.7.1, it was required that suitable predicates were always used to refine the initial region.

[11] Personal correspondence by email on the 5$^{\text{th}}$ of April 2014.

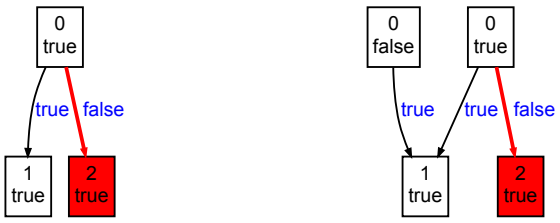


Figure 3.16: (a) shows an infeasible trace where the initial region contains no states. Refining it by splitting the initial region with the refinement predicate ***true*** *results in (b). No progress has been made in (b) compared to (a).*

If we had continued and refined the initial region by splitting it, the region graph in Figure 3.16b would be obtained. Notice that no progress has been achieved in this graph. The same frontier exists and when refining it, the initial region is split resulting in three initial regions. This process will continue infinitely and therefore it simply does not make sense to use **true** as a refinement predicate.

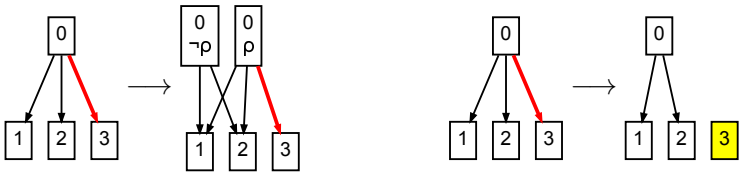
3.8.10 Problem: What happens when splitting the initial region

The RefineGraph pseudocode in Algorithm 3.3 removes edges from the initial region if they are found infeasible. This "optimization" is not mentioned in the DASH article, but is a solution we found ourselves.

The DASH article does not describe what happens when an initial region is refined. If an initial region is handled in the exact same manner as any other region, then the process of refining an initial region is shown in Figure 3.17a. The red frontier at the initial region before refinement is found infeasible and the initial region is refined with the refinement predicate ρ. This results in the graph after the arrow $\longrightarrow$. Notice that there have been no changes made for region 1 and 2, they can still be reached with any state that could reach it before the refinement. This is evident since to reach region 1 or 2, one has to go through one of the initial regions, which means that the initial state must satisfy the predicate $\neg\rho \vee \rho$. But that predicate is equivalent to **true**, and thus no changes have been made for them.

However, notice that the red frontier $(0{:}\rho, 3)$ after refinement is still infeasible. The reason is that it was found infeasible on the left, and since the requirements have only grown, it is still infeasible. Thus when refining the initial region, when it has no incoming edges, then the edge could be removed instead of adding predicates and maintaining multiple initial regions. This is the process depicted in Figure 3.17b. The edge is simply removed. This is the strategy we have implemented.

Initially, we refined the initial region using the first strategy shown in Figure 3.17a, but where only the left region was marked as an initial region. The right one was ignored and thought to be unreachable. For all cases in DASH$_{int}$,



(a) No special case handling

(b) Special case: remove edge

Figure 3.17: Two strategies used to refine the initial region. Region graphs are shown before and after refinement.

the refinement predicate ρ is **false** when refining the initial region. It will always be **false** because RefinePred is disallowed to use the loop optimization on edges going out of the initial region (which contains no predicates), and thus it always uses WP to construct the refinement predicate.

However, in DASH$_{call}$, there are cases where the refinement predicate is not **false**. This is because the initial region can contain a predicate, which is not **true**, when analyzing sub procedures. A concrete example from DASH$_{call}$ can be seen in Figure 4.6, where the initial region is refined with a predicate that is not **false**. Splitting the initial region, as in Figure 3.17a, without marking the right region as an initial region, suddenly makes region 1 and 2 only reachable when the predicate $\neg\rho$ is satisfied, which is clearly a wrong refinement. We therefore needed to maintain either multiple initial regions or, when refining the initial region, remove the frontier edge such that only a single initial region exists. We chose the last option as it generated cleaner graphs.

Chapter 4

Interprocedural analysis with DASH$_{call}$

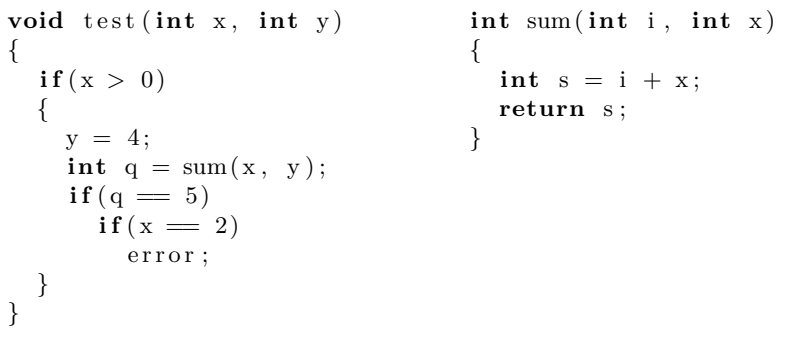
In this chapter we present DASH$_{call}$ that adds interprocedural analysis on top of DASH$_{int}$.

The allowed program statements are extended with assignments of procedure calls $\mathrm{v} := \mathrm{f}(a_0, a_1, \ldots, a_n)$ and **return** e statements. However, the error statement is restricted to the main procedure P. Thus, error statements in sub procedures are not supported primarily due to how FindAbstractErrorPath works. This is explained further in Section 4.1. Procedure calls $\mathrm{f}(a_0, a_1, \ldots, a_n)$ that are not part of an assignment are not supported, since procedures are side effect free, due to the lack of global state, pointers, call by value semantics, and the fact that error statements are not allowed in sub procedures. The procedure call could therefore be removed, without changing the semantics, in a preprocessing step. State changed in a sub procedure must thus be propagated back through a return value. For the rest of the chapter, a *procedure call* will refer to a procedure call that is part of an assignment.

DASH$_{call}$ analyzes a complete program $\mathcal{P}$ where each procedure has its own region graph. The DASH$_{call}$ version of DashLoop takes as input the program $\mathcal{P}$, a set of region graphs $\mathcal{G}$ and the main procedure P. Special care must be taken when DASH$_{call}$ encounters a procedure call as the frontier edge. The main idea is to recursively invoke DashLoop on the invoked procedure, but restraining its input to the context it is called in, and instrumenting it with a safety property that depends on crossing the frontier. We will explain this in detail when describing ExtendFrontier in Section 4.4.

ExecuteSymbolic also needs to be modified such that it can execute a trace over a procedure call edge. At that point it needs to execute a trace through the invoked procedure. We describe ExecuteSymbolic for DASH$_{call}$ in Section 4.3.

The DASH$_{call}$ implementation is able to handle recursive and mutually recursive procedures if the input value that determines the recursion depth is bound to a constant, ex. fib(3) or fib(x), where x is assigned a constant value earlier. However it is not able to handle recursive procedures, where the recursion depth is dependent on a symbolic input value, e.i. x_0. We elaborate on recursive procedures in Section 4.7.3.



```
void test(int x, int y)
{
  if(x > 0)
  {
    y = 4;
    int q = sum(x, y);
    if(q == 5)
      if(x == 2)
        error;
  }
}
```

(a) Caller procedure

```
int sum(int i, int x)
{
  int s = i + x;
  return s;
}
```

(b) Called procedure

Figure 4.1: (a) shows a procedure test which calls the procedure sum in (b). It is used as a running example to describe how DASH$_{call}$ works.

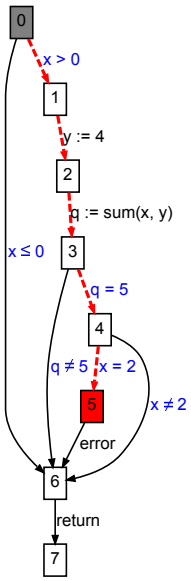
We use a running example in this chapter to illustrate the inner details of how DASH$_{call}$ behaves. The example code is shown in Figure 4.1. The example includes two procedures, test taking variables x and y as parameters, and a sum procedure taking the variables i and x as parameters.

There are a few things to notice about this program. First, it is not possible to reach the error statement, since it requires y to be 4 and z to be 2, but the sum of them should be 5. Second, it should be noted that both procedures take x as a parameter, which could result in naming conflicts during analysis if care is not taken.

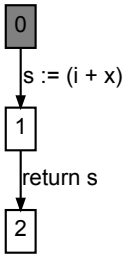
This chapter starts by presenting how FindAbstractErrorPath finds an abstract error path in Section 4.1. Changes must be made to the result returned by ConvertToRegionTraceWithAbstractFrontier, which we describe in Section 4.2. At this point the prerequisites for ExecuteSymbolic have been shown and we describe how ExecuteSymbolic can symbolically execute a trace with procedure calls in Section 4.3. The most important changes in DASH$_{call}$ are made in ExtendFrontier that analyzes a procedure call at the frontier edge, which is described in Section 4.4. Afterwards we describe some changes to RefineGraph in Section 4.5. We then describe a possible instrumentation process that can relax the requirement that only error statements in the main procedure are allowed in Section 4.6. Finally we describe key problems encountered and changes we have made to implement DASH$_{call}$ in Section 4.7.

4.1 FindAbstractErrorPath

The mode of operation of FindAbstractErrorPath in DASH$_{call}$ has not changed compared to the one presented in DASH$_{int}$. It searches a given region graph for a shortest path to an error region. Thus it completely ignores the semantics of procedure call edges, and as such it does not recursively search region graphs of called procedures. The reason for not searching for error regions in sub procedures is that the DASH article has not described how this should be completed. We present a possible solution in Section 4.6 to circumvent this



(a) Path found in the test procedure



(b) Path does not include the sum procedure

Figure 4.2: (a) shows the path found by FindAbstractErrorPath for the first iteration of DASH$_{call}$ while analyzing the code shown in Figure 4.1. Notice that the path does not pass through (b).

limitation if global state is supported. Thus, the FindAbstractErrorPath procedure is not changed. This is emphasized by an example.

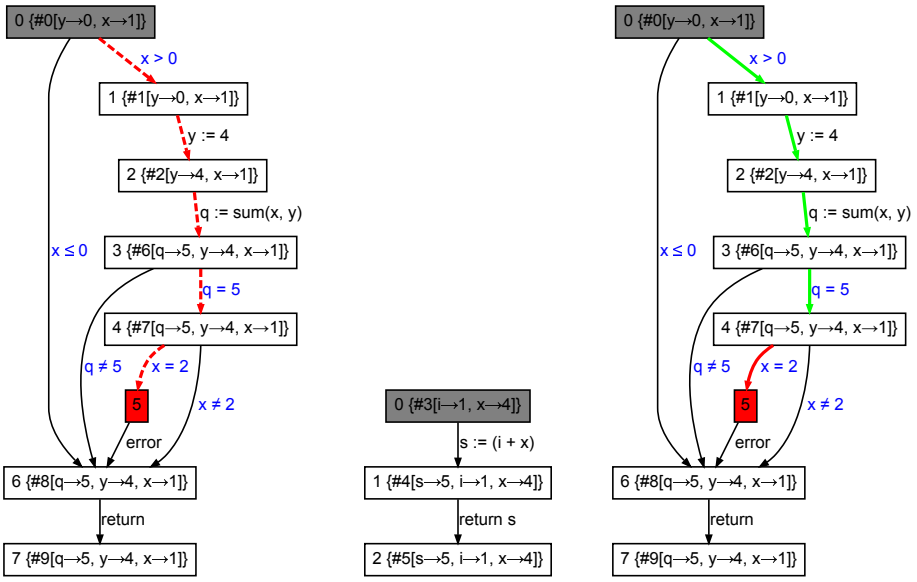
The path seen in Figure 4.2a is the path found when searching for an error path in the first iteration of analyzing the code from Figure 4.1. Thus, the path found is $\langle 0, 1, 2, 3, 4, 5 \rangle$, which specifically does *not* traverse into the sum sub procedure shown in Figure 4.2b.

4.2 ConvertToRegionTraceWithAbstractFrontier

The ConvertToRegionTraceWithAbstractFrontier procedure needs to be slightly changed. It will construct the trace in the exact same manner as in DASH$_{int}$, however, it will also include the states for the part of the trace that follows a concrete execution.

Formally the trace will now have the form $\tau_c = \langle RS_0, \ldots RS_{k-1}, S_k \rangle$ where each $RS_i = \langle S_i, s_i \rangle$ is called a RegionState. Each RegionState includes the region S_i and the state s_i that was found by following the parent relationship from s_{k-1}. The reason that S_k does not include a state is that it is the region after the frontier and therefore no concrete execution has reached it yet.

The states are used in ExecuteSymbolic to figure out how to symbolically execute a procedure call edge when these are encountered in a trace. States s_i are thus only needed when the following statement is a procedure call. This is described in Section 4.3.1, which describes the ExecuteSymbolic procedure for DASH$_{call}$.



(a) Error path found in the test procedure

(b) States in the sum sub procedure

(c) Trace found for the path in (a)

Figure 4.3: ConvertToRegionTraceWithAbstractFrontier converts the path in (a) to the trace in (c). (b) shows the states that were added to the sum procedure when test procedure was executed by an earlier invocation of RunTest. These states are not included in either the path or the trace.

Figure 4.3a shows a path that is to be converted to a trace. The trace found is shown in Figure 4.3c. Region 2 includes a state which is prefixed with #2 and region 3 has a state with the prefix #6. These numbers tells us that 3 regions were visited between these two states, and these are recorded on the graph of the sub procedure seen in Figure 4.3b. It should be noted that states are only linked inside a single procedure. Thus $\mathsf{Parent}(\#6) = \#2$ and not #5 as could have been expected.

Writing out the resulting trace in full, including each of the states, produces the following:

$$\begin{aligned}\tau_c = \langle &\langle 0, \#0[y \mapsto 0, x \mapsto 1]\rangle,\\ &\langle 1, \#1[y \mapsto 0, x \mapsto 1]\rangle,\\ &\langle 2, \#2[y \mapsto 4, x \mapsto 1]\rangle,\\ &\langle 3, \#6[q \mapsto 5, y \mapsto 4, x \mapsto 1]\rangle,\\ &\langle 4, \#7[q \mapsto 5, y \mapsto 4, x \mapsto 1]\rangle,\\ &5\rangle\end{aligned}$$

The next section presents ExecuteSymbolic, the procedure that symbolically executes traces.

4.3 ExecuteSymbolic

The goal of ExecuteSymbolic is to execute a trace τ_c that traverses the procedure P. Compared to DASH$_{int}$ the new requirements are that ExecuteSymbolic handles procedure calls during the trace and additionally returns the symbolic map $\mathcal{S}$ as it was before executing the frontier statement. As a special case, when the frontier edge is a procedure call, ExecuteSymbolic ignores it and leaves it to be handled by ExtendFrontier, which is described in Section 4.4.

ExecuteSymbolic has been split into two parts. The ExecuteSymbolic algorithm, shown in Algorithm 4.1, that executes the trace τ_c, and ExecuteSymbolicSubProcedure, shown in Algorithm 4.2, that symbolically executes sub procedures.

ExecuteSymbolic takes as input a trace τ_c, the procedure P, which the trace runs through, the program $\mathcal{P}$ that contains all the procedures, and finally ExecuteSymbolic takes the complete set of graphs $\mathcal{G}$. As mentioned in Section 4.2 the trace τ_c has been changed from a trace of regions $\langle S_0, \ldots, S_k \rangle$ to a trace of RegionStates $\langle RS_0, \ldots, RS_{k-1}, S_k \rangle$.

To support the added requirements three major changes have been made to the pseudocode of ExecuteSymbolic:

- The ExecuteSymbolic return statement has been changed, such that it also returns the symbolic map $\mathcal{S}$ before executing the frontier.
- Two new cases has been added. These cases handle procedure calls of the form v := f($a_0, \ldots, a_n$) and **return** e statements.
- To return the symbolic map $\mathcal{S}$ as it is seen before the frontier is executed, and because procedure calls are handled differently if they are included before the frontier edge, the execution of the frontier edge has been unrolled from the loop.

ExecuteSymbolic returns the same path constraint ϕ that would be found by the DASH$_{int}$ version of ExecuteSymbolic, when executing a trace that does not include a procedure call. However, ExecuteSymbolic for DASH$_{call}$ also returns the symbolic map $\mathcal{S}$.

Execution of the trace before the frontier

The loop in Algorithm 4.1 behaves the same as in DASH$_{int}$ except that it does not execute the frontier statement, and it additionally handles procedure calls. Edges with **return** e statements can never occur before the frontier edge in a trace τ_c. This is because there are no outgoing edges from regions with incoming **return** edges, since execution stops when a **return** statement is executed. The **return** e case in lines 22-23 has been included in the pseudocode for completeness, but will never be triggered and thus it is simply skipped.

Since procedures only see their parameters and cannot alter global state, handling them symbolically is a matter of constructing the correct parameters for them. Lines 16-21 handles procedure calls of the form v := f($a_0, \ldots, a_n$). The arguments $a_0, \ldots, a_n$ to the procedure call are each symbolically evaluated and

Algorithm 4.1 ExecuteSymbolic($\tau_c = \langle RS_0, \ldots, RS_{k-1}, S_k \rangle, P, \mathcal{P}, \mathcal{G}$)

Returns: $\langle \phi, \mathcal{S} \rangle$, the path constraint ϕ for reaching and crossing the frontier; and the symbolic map $\mathcal{S}$, from before the frontier.

```
1: let ⟨S_0, _⟩ = RS_0
2: let ⟨ρ_0, _⟩ = S_0
3: S := [v ↦ v_0 | v ∈ params(P)]
4: φ := SymbolicEval(ρ_0, S)
5:
6: for i = 0 to k − 2 do
7:     let ⟨_, s_i⟩ = RS_i
8:     let ⟨S_{i+1}, _⟩ = RS_{i+1}
9:     let ⟨ρ_{i+1}, _⟩ = S_{i+1}
10:    op := Op(S_i, S_{i+1})
11:    match op
12:        case(v := e):
13:            S := S[v ↦ SymbolicEval(e, S)]
14:        case(assume c):
15:            φ := φ ∧ SymbolicEval(c, S)
16:        case(v := f(a_0, ..., a_n)):
17:            P' := LookupProcedure(f, P)
18:            args := ⟨SymbolicEval(a_i, S) | ∀a_i⟩
19:            ⟨φ', e'⟩ := ExecuteSymbolicSubProcedure(P', args, s_i, P, G)]
20:            φ := φ ∧ φ'
21:            S := S[v ↦ e']
22:        case(return e):
23:            skip
24:    φ := φ ∧ SymbolicEval(ρ_{i+1}, S)
25: end for
26:
27: let ⟨S_{k−1}, _⟩ = RS_{k−1}
28: let ⟨ρ_k, _⟩ = S_k
29: op := Op(S_{k−1}, S_k)
30: match op
31:    case(v := e):
32:        S' := S[v ↦ SymbolicEval(e, S)]
33:        φ := φ ∧ SymbolicEval(ρ_k, S')
34:    case(assume c):
35:        φ := φ ∧ SymbolicEval(c, S)
36:        φ := φ ∧ SymbolicEval(ρ_k, S)
37:    case(v := f(a_0, ..., a_n)):
38:        skip
39:    case(return e):
40:        φ := φ ∧ SymbolicEval(ρ_k, S)
41: return ⟨φ, S⟩
```

added to a list $args$ in line 18, which functions as the symbolic values of the parameters. Thus if a procedure takes as input x, then the symbolic expression of x, which could be $x_0 + y_0/3$, will be in $args$. The symbolic expressions consist of expressions including only constants or the initial symbolic values generated in line 3.

A call is made to ExecuteSymbolicSubProcedure in line 19, which is responsible for executing the sub procedure symbolically. It is important that ExecuteSymbolicSubProcedure follows the same execution that was followed in the trace τ_c. Therefore, it is among others given the state s_i from the trace τ_c, which is the state that existed prior to executing the procedure call. The region graph for the called procedure P' might contain many concrete executions from prior invocations of RunTest and ExecuteSymbolicSubProcedure uses s_i to find the concrete execution that was generated by the execution that τ_c follows.

The call to ExecuteSymbolicSubProcedure returns an induced path constraint ϕ' from the procedure call and the symbolic expression e' that represents the computation performed by the called procedure. The path constraint ϕ' is added to ϕ in line 20 while v is assigned the symbolic return value e' in line 21. Execution of sub procedures, which is performed by ExecuteSymbolicSubProcedure, is described in Section 4.3.1.

Execution of the frontier

There are multiple changes to how the frontier statement is being symbolically executed. We need to return the symbolic map $\mathcal{S}$ as it was before executing the frontier statement. This is a requirement enforced by ExtendFrontier, which uses it when handling procedure calls at the frontier edge. ExecuteSymbolic still needs to handle assignments and **assume** c statements, but it also needs to handle **return** e statements as part of the frontier.

If the frontier statement is a procedure call $\mathrm{v} := \mathrm{f}(a_0, \ldots, a_n)$ then ExecuteSymbolic skips it in lines 37-38. Procedure calls at the frontier will be handled by ExtendFrontier described in Section 4.4. The reason is that ExecuteSymbolic does not have a trace that it can follow in the called procedure, and it therefore has a large number of possible execution paths that it could select for symbolic execution in the called procedure. Choosing one path might make the combined path constraint feasible while choosing another path might make it infeasible.

In essence the question given is a reachability question: Can a path be found in the procedure f, under the symbolic state $\mathcal{S}$ and with the path constraint ϕ, such that a state s is generated that satisfies the region predicate after the frontier. This is exactly the kind of questions that DASH is constructed to answer and ExtendFrontier exploits this, as we shall see in Section 4.4. Thus ExecuteSymbolic skips procedure calls at the frontier edge and leaves the work to be completed by ExtendFrontier.

What is left to be handled are **assume** c, assignments and **return** e statements. Symbolically executing the region predicate has been moved into each of the cases since the case that handles procedure calls does not symbolically execute the region predicate. All **assume** c statements are handled exactly as if they had

occurred before the frontier.

Assignments are handled in lines 31-33. The assignment operation needs special handling since it changes the state of the symbolic map. This state change is needed when evaluating the region after the frontier. However we need to return the state of the symbolic map before the frontier. The temporary symbolic map $\mathcal{S}'$ is therefore introduced and holds the symbolic map $\mathcal{S}$ with state applied by the assignment, and it is then used when evaluating the region predicate after the frontier. The symbolic map $\mathcal{S}$ remains unchanged.

The last kind of statements that can occur is the **return** e statement. The **return** e statement does not change the symbolic state, nor the path constraint, therefore only the region predicate needs to be symbolically executed, which is completed in line 40.

The next section describes ExecuteSymbolicSubProcedure, which executes called sub procedures during the trace τ_c.

4.3.1 ExecuteSymbolicSubProcedure

The pseudocode for ExecuteSymbolicSubProcedure is shown in Algorithm 4.2. The goal of this procedure is to symbolically execute a called sub procedure. It returns the generated path constraint ϕ and the symbolic return value e. The following input is given to ExecuteSymbolicSubProcedure:

- The called procedure P, which should be symbolically executed.
- The symbolically evaluated arguments in $args$, which represents the parameters of P in terms of the initial symbolic variables generated in ExecuteSymbolic.
- The state s_{caller} before the procedure call that is used to find the concrete execution to follow in P.
- Additionally ExecuteSymbolicSubProcedure is given the program $\mathcal{P}$ and the set of graphs $\mathcal{G}$, which are used for looking up procedures and graphs.

ExecuteSymbolicSubProcedure functions in nearly the same way as ExecuteSymbolic. As in ExecuteSymbolic there is a loop that executes the trace, and the cases for assignments, assumes and procedure calls are nearly identical. However, there are some differences:

- The initial symbolic variables for the parameters of P is given in $args$. This is shown in line 1 which sets the initial symbolic variables to the ones given in $args$, and thus does not create new symbolic variables v_0.
- In ExecuteSymbolic a trace τ_c is given and followed. In ExecuteSymbolicSubProcedure the initial region of the called procedure and the concrete execution to follow needs to be found, this is completed in lines 3-4. Additionally the trace needs to be explicitly maintained, which is fulfilled in lines 8-9 and 25-26.

Algorithm 4.2 ExecuteSymbolicSubProcedure($P, args = \langle b_0, \ldots, b_n \rangle, s_{caller}, \mathcal{P}, \mathcal{G}$)

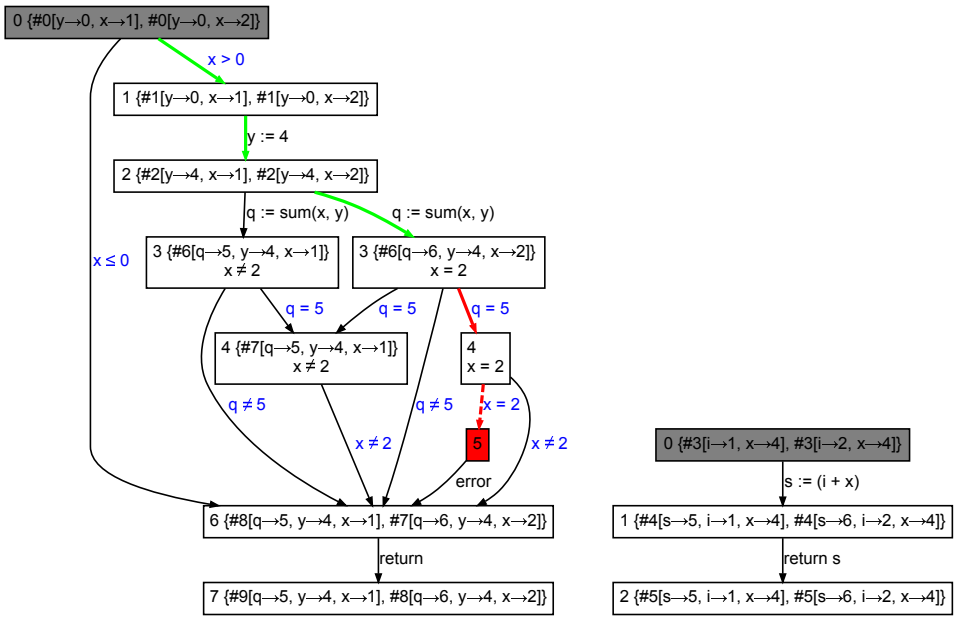
Returns: $\langle \phi, e \rangle$, ϕ is the induced path constraint and e is the symbolic return value of the procedure invocation.

```
 1: S := [v_i ↦ b_i | ∀v_i ∈ params(P)]
 2: G := LookupRegionGraph(P, G)
 3: S_prev := InitialRegion(G)
 4: s_prev := FindCalleeStateFromCallersState(s_caller, S_prev)
 5: φ := true
 6:
 7: loop
 8:     s_next := Child(s_prev)
 9:     S_next := FindRegionWithState(Children(S_prev), s_next)
10:     op := Op(S_prev, S_next)
11:     match op
12:         case(v := e):
13:             S := S[v ↦ SymbolicEval(e, S)]
14:         case(assume c):
15:             φ := φ ∧ SymbolicEval(c, S)
16:         case(v := f(a_0, ..., a_n)):
17:             P' := LookupProcedure(f, P)
18:             args' := ⟨SymbolicEval(a_i, S) | ∀a_i⟩
19:             ⟨φ', e'⟩ := ExecuteSymbolicSubProcedure(P', args', s_prev, P, G)]
20:             φ := φ ∧ φ'
21:             S := S[v ↦ e']
22:         case(return e):
23:             e_result := SymbolicEval(e, S)
24:             return ⟨φ, e_result⟩
25:     s_prev := s_next
26:     S_prev := S_next
27: end loop
```

- When a **return** e statement is encountered it ends the symbolic execution of the sub procedure P. ExecuteSymbolicSubProcedure then returns the path constraint ϕ and the symbolic expression that represents the result of following the induced trace in P.
- Region predicates are not executed since the regions of subgraphs do not contain predicates. Only the graph that ExecuteSymbolic works on can be refined, and it is therefore the only graph that can have predicates on the regions.

The construction of the initial symbolic map is performed in line 1. This is completed by mapping the formal parameters to their corresponding arguments provided in $args$. This ensures that any symbolically evaluated expressions will only contain expressions with constants or initial symbolic variables v_0 generated in ExecuteSymbolic.

To start the execution, ExecuteSymbolicSubProcedure needs to find the initial state in the concrete execution that it should follow in the sub procedure. The concrete execution that should be followed is the one that was generated by calling P when in the state s_{caller}. Finding the initial state is accomplished by first finding the initial region of P, in line 3, with a call to InitialRegion. ExecuteSymbolicSubProcedure then needs to find the state in the initial region



(a) The main procedure test (b) The called procedure sum

Figure 4.4: (a) shows a trace that needs to be symbolically executed, which contains a call before the frontier to the sub procedure in (b).

that was generated by s_{caller}. This is accomplished in line 4 with a call to FindCalleeStateFromCallersState. Each of our states contain both a test id, which is unique for each RunTest invocation, and it contains the number of steps taken in the execution. Given that s_{caller} has a test id we know that the initial state must have the same test id. Also, if the state s_{caller} has taken n steps in the execution, then the initial state must have taken exactly $n + 1$ steps. Given these two pieces of information, we can find the initial state.

The initial region is assigned to S_{prev}, while the region that ExecuteSymbolicSubProcedure is about to transition to is called S_{next}. The same convention is used for the concrete states s_{prev} and s_{next}. We use the child relationship on s_{prev} to find s_{next}. The region S_{next} that contains s_{next} can be found by going through all the child regions of S_{prev} and find the region that contains s_{next}. This is exactly what FindRegionWithState does. The regions S_{prev} and S_{next} can then be used to find the operation op to be executed. After executing the operation op, s_{prev} and S_{prev} is set to s_{next} and S_{next} to prepare for the next iteration of the loop.

This concludes the explanation of ExecuteSymbolic and ExecuteSymbolicSubProcedure. In the next section an example of them is given.

4.3.2 Example of ExecuteSymbolic

As an example of how ExecuteSymbolic functions in DASH$_{call}$, we will symbolically execute the trace found in Figure 4.4a.

In Figure 4.4a we can see the trace $\langle 0, 1, 2, 3{:}x{=}2, 4{:}x{=}2\rangle$ following the green

edges down and including the red frontier. The green part of the trace is found by following the concrete states where $x \mapsto 2$, which is included in the trace τ_c.

Setup and symbolically evaluating the initial region

At startup the symbolic map $\mathcal{S}_{test}$ for the test procedure is set to $[x \mapsto x_0, y \mapsto y_0]$. Symbolically evaluating the region predicate of the initial region sets ϕ_{test} to **true** resulting in the state of ExecuteSymbolic becoming:

$$\begin{aligned} \phi_{test} &\longrightarrow \textbf{true} \\ \mathcal{S}_{test} &\longrightarrow [x \mapsto x_0, y \mapsto y_0] \end{aligned}$$

Symbolically executing **assume** $x > 0$ **on edge** $(0, 1)$

There are no changes to how this and the next step is symbolically executed compared to how DASH_{int} would execute them. The step $(0, 1)$ executes the **assume** $x > 0$ statement and symbolically evaluates the region predicate in region 1, which is **true**. The **assume** $x > 0$ statement adds $x_0 > 0$ to ϕ_{test}. The symbolic map is left unchanged. The region predicate for region 1 is **true** and therefore nothing new was added to ϕ_{test} while executing it. After this step the state of ExecuteSymbolic is:

$$\begin{aligned} \phi_{test} &\longrightarrow x_0 > 0 \\ \mathcal{S}_{test} &\longrightarrow [x \mapsto x_0, y \mapsto y_0] \end{aligned}$$

Symbolically executing `y := 4` **on edge** $(1, 2)$

The second edge $(1, 2)$ with $y := 4$, updates the mapping for y in the symbolic map $\mathcal{S}_{test}$. Since the region predicate is **true** ϕ_{test} is not affected. This leads to the ExecuteSymbolic state:

$$\begin{aligned} \phi_{test} &\longrightarrow x_0 > 0 \\ \mathcal{S}_{test} &\longrightarrow [x \mapsto x_0, y \mapsto 4] \end{aligned}$$

Symbolically executing the `q := sum(x, y)` **call on edge** $(2, 3{:}x{=}2)$

This step symbolically executes the operation `q := sum(x, y)`. First we find the procedure description P_{sum} for sum, in line 17, by a call to LookupProcedure. Then the symbolic argument list $args$ is constructed. The variables x and y are used as arguments to sum and symbolically evaluating them results in $args = \langle x_0, 4 \rangle$.

ExecuteSymbolicSubProcedure is called in line 19 which executes the sub procedure. It takes as arguments the called procedure P_{sum}, $args$ and additionally the current concrete state, which is $s_2 = \#2[y \mapsto 4, x \mapsto 2]$, the program $\mathcal{P}$ and all graphs $\mathcal{G}$.

Symbolically executing sum(x, y) with ExecuteSymbolicSubProcedure

ExecuteSymbolicSubProcedure starts by setting up its own symbolic map $\mathcal{S}_{sum}$ from the symbolic argument list $args$, which then becomes $[i \mapsto x_0, x \mapsto 4]$. It then finds the graph G_{sum} for the current procedure P_{sum} such that it can find the initial region and store it in S_{prev}.

The initial region is region 0 in Figure 4.4b. We need to find the state that follows the state s_2 given as s_{caller}. In region 0 there are two states, namely $\#3[i \mapsto 1, x \mapsto 4]$ and $\#3[i \mapsto 2, x \mapsto 4]$. The states additionally have a test id attached. Both states have the same number of execution steps, namely 3. However, only one state has the same test id as s_{caller} and in this case it is the state $\#3[i \mapsto 2, x \mapsto 4]$. This state is assigned to the variable s_{prev}.

Transitioning between regions in sum is trivial since there is only a single outgoing edge from each region. However, in the general case, there can multiple outgoing edges because of conditionals, and ExecuteSymbolicSubProcedure needs to figure out which edge to follow. This is accomplished by finding the next state s_{next} by following the child relationship on s_{prev}. For the first iteration, s_{next} is the state $\#4[s \mapsto 6, i \mapsto 2, x \mapsto 4]$. Using the state s_{next} the region S_{next} is found, which is region 1.

The edge between region 0 and region 1 has the operation $s := (i+x)$ and the symbolic map $\mathcal{S}_{sum}$ is therefore updated to $[s \mapsto x_0{+}4, i \mapsto x_0, x \mapsto 4]$ and ϕ is left unchanged. Afterwards s_{prev} is set to the state $s_{next} = \#4[s \mapsto 6, i \mapsto 2, x \mapsto 4]$ and S_{prev} is set to S_{next}, which is region 1.

The next state s_{next} is found, and in this case it is $\#5[s \mapsto 6, i \mapsto 2, x \mapsto 4]$. The region that the state is contained in, region 2, is assigned to S_{next}. The operation on the edge $(1, 2)$ is the return statement **return** s. Symbolically evaluating this yields the result $e_{result} = x_0 + 4$, which is returned from ExecuteSymbolicSubProcedure together with the path constraint **true**.

Handling return from the procedure call q := sum(x, y)

When ExecuteSymbolicSubProcedure returns, ExecuteSymbolic stores the path constraint in ϕ' and the result in e'. The path constraint is added to ϕ, but since ϕ' is **true** then ϕ is left unchanged. The assigned variable q is updated to the value e' in the symbolic map $\mathcal{S}_{test}$, such that it maps to the symbolic expression $x_0 + 4$ returned by ExecuteSymbolicSubProcedure. Finally, the region predicate from region 3, which is $x = 2$, is symbolically evaluated and added to ϕ resulting in the state of ExecuteSymbolic becoming:

$$\begin{aligned} \phi_{test} &\longrightarrow x_0 > 0 \wedge x_0 = 2 \\ \mathcal{S}_{test} &\longrightarrow [x \mapsto x_0, y \mapsto 4, q \mapsto x_0 + 4] \end{aligned}$$

Symbolically executing assume $q = 5$ at the frontier edge $(3{:}x{=}2, 4{:}x{=}2)$

ExecuteSymbolic exits the loop, since it is about to execute the frontier edge $(3{:}x{=}2, 4{:}x{=}2)$. This edge has the operation **assume** $q = 5$ and the region after it contains the predicate $x = 2$. Evaluating both of these predicates, adding them to ϕ, and removing duplicate terms results in the path constraint

$\phi = x_0 > 0 \wedge x_0 = 2 \wedge x_0 + 4 = 5$. The result returned from ExecuteSymbolic is thus:

$$\begin{aligned} \phi &\longrightarrow x_0 > 0 \wedge x_0 = 2 \wedge x_0 + 4 = 5 \\ \mathcal{S} &\longrightarrow [x \mapsto x_0, y \mapsto 4, q \mapsto x_0 + 4] \end{aligned}$$

This path constraint will be found unsatisfiable by ExtendFrontier, due to the constraints $x_0 = 2 \wedge x_0 + 4 = 5$.

The next section explains ExtendFrontier and how procedure calls are handled when they are encountered at the frontier edge.

4.4 ExtendFrontier

ExtendFrontier for DASH$_{call}$ behaves the same as in DASH$_{int}$ for all traces except when the frontier is a procedure call. In this case, ExtendFrontier performs additional work that requires extra information passed from DashLoop. As in DASH$_{int}$, ExtendFrontier takes the trace τ_c and the procedure currently being analyzed P. Additionally, it takes the region graphs for all procedures $\mathcal{G}$ and the full program $\mathcal{P}$ as arguments. The DASH$_{call}$ variant of ExtendFrontier is shown in Algorithm 4.3.

If the frontier edge (S_{k-1}, S_k) is a procedure call $\mathrm{v} := \mathrm{f}(a_0, \ldots, a_n)$, then the job of ExtendFrontier is to check if it is possible to reach a state s by calling f such that the state s satisfies the region predicate from S_k. By encoding this problem on the called procedure P' in a clever way, we can use DashLoop to answer the problem. We wish to invoke DashLoop on the called procedure P' such that:

- If DashLoop reports that an error could be reached with test input t, then we are able to run a test that crosses the frontier procedure call and get a state s that satisfies the region predicate in S_k.
- If DashLoop reports that no errors could be reached with a graph G as proof, then a refinement predicate ρ, which eliminates the trace τ_c in the caller procedure, can be obtained by inspecting the proof G.

This can be achieved by running DashLoop on a slightly altered version of $P's$ region graph. The job of ExtendFrontier is now to:

- Compute the input constraint ϕ_{ic} for the analysis of P', such that only values that can be generated by following the trace τ_c in P can be used when analyzing P'.
- Encode the region predicate of S_k into an exit constraint ϕ_{ec} such that if analysis of P' reaches a region with ϕ_{ec}, then S_k can be reached in the caller procedure P.
- Use ϕ_{ic} and ϕ_{ec} to alter P's initial region graph G'.
- Invoke DashLoop on the altered region graph G'.

Algorithm 4.3 ExtendFrontier($\tau_c = \langle RS_0, \ldots, RS_{k-1}, S_k \rangle, P, \mathcal{G}, \mathcal{P}$)

Returns:
$\langle t, \textbf{true} \rangle$, if the frontier can be extended; or
$\langle \text{UNSAT}, \rho \rangle$, if the frontier cannot be extended

```
1: let ⟨S_{k-1}, _⟩ = RS_{k-1}
2: ⟨φ, S⟩ := ExecuteSymbolic(τ_c, P, P, G)
3: op := Op(S_{k-1}, S_k)
4: if op matches v := f(a_0, ..., a_n) then
5:     let ⟨ρ_k, _⟩ = S_k
6:     P' := Lookup(f, P)
7:     π := CreateVariableRenamer(locals(P) ∪ {v, v_0 | ∀v ∈ params(P)})
8:     φ_ic := (⋀_{v_i ∈ params(P')} v_i = π(a_i)) ∧ (⋀_{(w ↦ e) ∈ S} π(w = e)) ∧ π(φ)
9:     φ_ec := π(ρ_k[@r/v])
10:    G' := ReconstructGraphsAndInsertConstraints(G, φ_ic, φ_ec, P')
11:    ⟨r, z⟩ = DashLoop(G', P, P')
12:    if r = FAIL then
13:        t := π^{-1}(z \ {v_0 | ∀v ∈ params(P')}) \ (locals(P) ∪ params(P))
14:        ρ := true
15:    else
16:        t := UNSAT
17:        ρ := π^{-1}((⋁_{ρ_i ∈ InitialRefines(z)} ρ_i)[a_0/v_0, ..., a_n/v_n] | v_i ∈ params(P'))
18:    end if
19: else
20:    t := IsSAT(φ, P)
21:    if t = UNSAT then
22:        ρ := RefinePred(τ_c)
23:    else
24:        ρ := true
25:    end if
26: end if
27: return ⟨t, ρ⟩
```

- When DashLoop returns, either extract the test input needed to call procedure P such that the frontier is crossed, or construct a refinement predicate to refine the region graph G, making τ_c infeasible in P.

Furthermore, ExtendFrontier has to rename variables in the constraints ϕ_{ic} and ϕ_{ec}, otherwise these variables might clash with variables in the called procedure.

The first lines of ExtendFrontier, shown in Algorithm 4.3, should be familiar. ExecuteSymbolic is called in line 2, returning a path constraint ϕ and the symbolic map $\mathcal{S}$ that existed before the frontier. Line 4 checks if the operation performed on the frontier edge is a procedure call. If it is not, ExtendFrontier behaves exactly like in DASH$_{int}$.

If the operation is a procedure call $\text{v} := \text{f}(a_0, \ldots, a_n)$ then ExecuteSymbolic did not symbolically execute it, and it is then the job of ExtendFrontier to check if the frontier edge can be crossed. In line 6, ExtendFrontier starts by looking up the called procedure f in the program $\mathcal{P}$, and finds the procedure P'. The next lines of ExtendFrontier construct the input and exit constraints, invokes DashLoop, and extracts the result of DashLoop. The next sections describe how each of these tasks is fulfilled by ExtendFrontier.

4.4.1 Renaming

In line 7, a variable renamer π is constructed by giving it a set of variables that it should rename, while any other variable should be left unchanged. The variables that needs to be renamed are all those variables that originate from the callers procedure P, such that they do not conflict with any variables in the callee's procedure P'. These are the local variables of P, parameters of P and the initial symbolic values v_0 generated in ExecuteSymbolic for the parameters of P. As an example, lets say that $\mathsf{locals}(P) = \{x, y\}$ and $\mathsf{params}(P) = \{a, b, c\}$. The construction of the renamer, in line 7, then proceeds as:

$$\begin{aligned}
\pi \;&:=\; \mathsf{CreateVariableRenamer}(\mathsf{locals}(P) \cup \{v, v_0 \mid \forall v \in \mathsf{params}(P)\}) \\
&\longrightarrow \mathsf{CreateVariableRenamer}(\{x, y\} \cup \{v, v_0 \mid \forall v \in \{a, b, c\}\}) \\
&\longrightarrow \mathsf{CreateVariableRenamer}(\{x, y\} \cup \{a, a_0, b, b_0, c, c_0\}) \\
&\longrightarrow \mathsf{CreateVariableRenamer}(\{x, y, a, a_0, b, b_0, c, c_0\})
\end{aligned}$$

Thus the variables $\{x, y, a, a_0, b, b_0, c, c_0\}$ should be renamed. This also illustrates that the notation $\{v, v_0 \mid \forall v \in \mathsf{params}(P)\}$ is a set that includes the parameters and their initial symbolic variables that were generated by ExecuteSymbolic.

The renamer uses a unique number and a character '$\downarrow$', which is not allowed in the source code, to rename variables. As such, $\pi(a)$ becomes $1{\downarrow}a$ if a is a variable that should be renamed and 1 is the unique number chosen when π was constructed. Whenever the symbol '$\downarrow$' is seen on a variable in a region graph, it is evident that the variable is external to the procedure. The operation π^{-1} is used to reverse the renaming and as such $\pi^{-1}(1{\downarrow}a)$ becomes the original variable a.

Alternatively we could have created a renamer that renames all occurrences of variables by prepending it with '$\downarrow$' such that $\pi(a)$ becomes $\downarrow a$ and $\pi(\downarrow a)$ becomes $\downarrow\downarrow a$. When reverse renaming, the expression $\pi(\downarrow\downarrow a)$ becomes $\downarrow a$ and variables that have not been renamed, such as a are not reverse renamed. The reason why our pseudocode explicitly lists the variables that should be renamed are twofold: 1) because it seems cleaner to explicitly rename only those variables that absolutely needs to be renamed and 2) because our implementation uses Z3 to perform the renaming that requires us to list the individual variables and their renamed counterparts. The variables that we do not rename, are those that have been renamed once before.

4.4.2 Constructing the input constraint ϕ_{ic}

The input to the called procedure P' must be restricted, such that only test input that is possible by following the trace τ_c in the caller procedure P, is allowed. How this is accomplished is described in this section.

For example, the code for the procedure test shown in Figure 4.1a calls the sum procedure shown in Figure 4.1b. The call made to sum is made with the actual arguments x and y, and sum takes the formal parameters i and x. Since both procedures use x as variables, these will be disambiguated with x_{test} and

x_{sum} in this presentation. Both of the variables x_{test} and y are constrained in test at the point where sum is called. The variable x_{test} must be positive and y must be equal to 4. The analysis of the sum procedure must similarly constrain the variables i and x_{sum}. Since x_{test} is given as i then i must be positive and since y is given for x_{sum} then x_{sum} must be 4.

In general, there are two possible origins of constraints. One origin is the path constraint ϕ generated by ExecuteSymbolic. In the example x_{test} is used for the parameter i, and since the symbolic value x_{test} is constrained to only positive values in the path constraint ϕ, then i must also be constrained to positive values. The second origin for constraints is the symbolic map $\mathcal{S}$. In the example y is set to the constant 4 prior to the sum procedure call. Since y is passed for the variable x_{sum}, then x_{sum} must be equal to 4. This constraint is not captured by the path constraint ϕ, but it is captured by the symbolic map, which at this point contains the mapping $y \mapsto 4$.

Furthermore the exit constraint ϕ_{ec}, constructed in the next section, may contain predicates which use local variables of the caller procedure, but where those variables are not passed as inputs to the callee. The constraints on these variables must also be added to the input constraint such that ϕ_{ec} is correctly bound to them. Thus, the input constraint constructed in line 8 uses the formula:

$$\phi_{ic} := \left(\bigwedge_{v_i \in \mathsf{params}(P')} v_i = \pi(a_i) \right) \wedge \left(\bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) \right) \wedge \pi(\phi)$$

First, the formal parameters are bound to the actual arguments. Second, the symbolic map $\mathcal{S}$ is added where each mapping is seen as an equality predicate and finally the path constraint ϕ is appended. Renaming is used on all the constraints except on the formal parameters of the called procedure P'.

Example of computing the input constraint

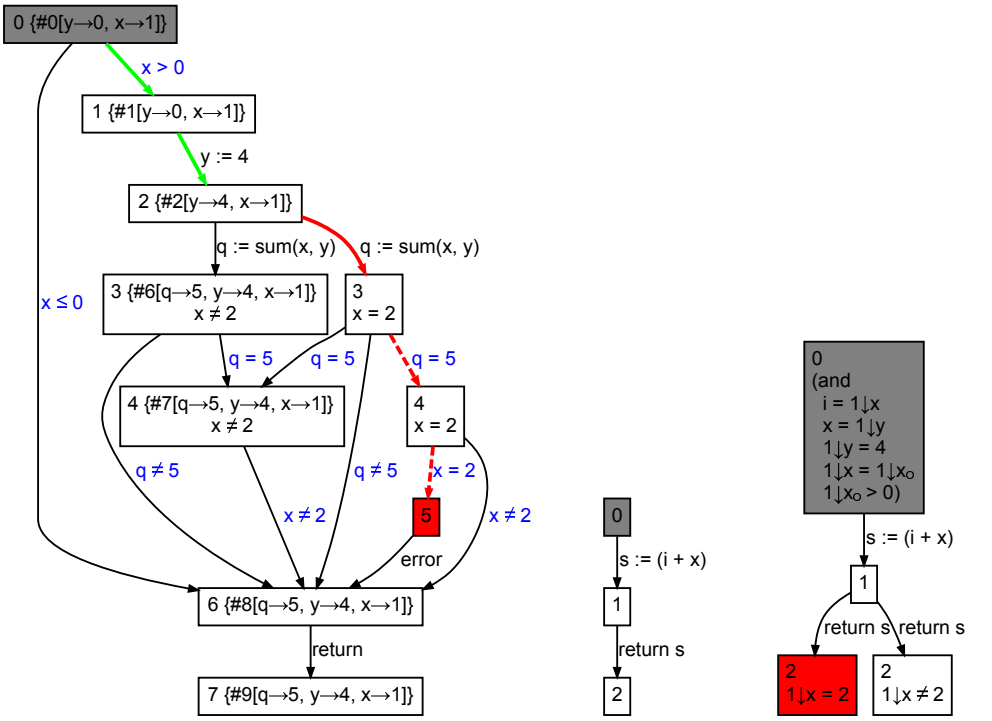
This section presents an example of computing the input constraint. Figure 4.5a shows the region graph for the test procedure listed in Figure 4.1a, after a number of iterations have been performed by DASH_{call}. The trace τ_c shown in the region graph has a frontier edge that is a procedure call to sum. ExtendFrontier constructs the region graph shown in Figure 4.5c and invokes DashLoop on it.

The initial region for sum in Figure 4.5c shows the added input constraint. First the formal parameters are bound to the actual arguments from test. In this case, x_{test} is bound to i and y is bound to x_{sum}. The computation becomes:

$$\begin{aligned} \bigwedge_{v_i \in \mathsf{params}(P')} v_i = \pi(a_i) \quad &\longrightarrow \quad i = \pi(x) \wedge x = \pi(y) \\ &\longrightarrow \quad i = 1{\downarrow}x \wedge x = 1{\downarrow}y \end{aligned}$$

which are the first two equalities shown in the initial region of Figure 4.5c.

Next the symbolic map is added. At the point of the procedure call, the symbolic map $\mathcal{S}$ contains the mappings $[x \mapsto x_0, y \mapsto 4]$. Computing the symbolic



(a) Region graph for the test procedure

(b) Intial region graph for sum

(c) Region graph with input and exit constraints for sum

Figure 4.5: (a) shows a region graph for the test *procedure after a number of iterations of* $DASH_{call}$*. The current frontier is a procedure call to* sum*. The initial region graph for* sum *is shown in (b) and the graph generated by* ExtendFrontier *for the call to* sum *is shown in (c).*

map part of ϕ_{ic} results in:

$$\begin{aligned} \bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) &\longrightarrow \pi(x = x_0) \wedge \pi(y = 4) \\ &\longrightarrow 1{\downarrow}x = 1{\downarrow}x_0 \wedge 1{\downarrow}y = 4 \end{aligned}$$

The result is included as the third and fourth equalities of the initial region in Figure 4.5c. The last step is to add the path constraint ϕ. In this example, the path constraint is $x_0 > 0$, which originates from the **assume** $x > 0$ statement between region 0 and 1 in Figure 4.5a. The computation $\pi(\phi)$ becomes:

$$\begin{aligned} \pi(\phi) &\longrightarrow \pi(x_0 > 0) \\ &\longrightarrow 1{\downarrow}x_0 > 0 \end{aligned}$$

The path constraint is the last predicate shown in the initial region of Figure 4.5c. We have now computed the input constraint ϕ_{ic}, which is the

conjunction of the computed constraints:

$$\begin{aligned}\phi_{ic} &\longrightarrow \left(\bigwedge_{v_i \in \mathsf{params}(P')} v_i = \pi(a_i)\right) \wedge \left(\bigwedge_{m \mapsto e \in \mathcal{S}} \pi(m = e)\right) \wedge \pi(\phi) \\ &\longrightarrow \left(i = 1{\downarrow}x \wedge x = 1{\downarrow}y\right) \wedge \left(1{\downarrow}x = 1{\downarrow}x_0 \wedge 1{\downarrow}y = 4\right) \wedge 1{\downarrow}x_0 > 0\end{aligned}$$

This is the constraint shown on the initial region in Figure 4.5c. Let us verify that the above constraint capture that i must be positive, and that x_{sum} must be equal to 4.

The variable x_{sum} is forced to be 4. This is evident when the two relevant constraints, one from the binding of arguments to parameters and one from the symbolic map, are shown together:

$$x = 1{\downarrow}y \wedge 1{\downarrow}y = 4 \quad \longrightarrow \quad x = 4$$

The requirement that i must be positive is also captured by the input constraint. Three constraints force i to be positive. One from the binding of arguments to parameters, one from the symbolic map, and finally the single path constraint. When shown together, it is easily verified that i must be positive:

$$i = 1{\downarrow}x \wedge 1{\downarrow}x = 1{\downarrow}x_0 \wedge 1{\downarrow}x_0 > 0 \quad \longrightarrow \quad i > 0$$

Thus, the requirements for i and x_{sum} are passed on when analyzing the sub procedure. The next section describes how the exit constraint ϕ_{ec} is generated.

4.4.3 Constructing the exit constraint ϕ_{ec}

ExtendFrontier needs to construct the exit constraint ϕ_{ec} for the procedure call $\mathrm{v} := \mathrm{f}(a_0, \ldots, a_n)$, such that if ϕ_{ec} can be satisfied at a **return** statement in P', then we can cross the frontier and reach S_k in P. This is accomplished in line 9 of Algorithm 4.3 with the pseudocode $\pi(\rho_k[@r/\mathrm{v}])$. Thus, we take the region predicate of S_k, the region after the frontier, and replaces all occurrences of v with a result placeholder variable $@r$. The placeholder variable $@r$ is used by ReconstructGraphsAndInsertConstraints and is explained in Section 4.4.4. The last step is to rename the variables that come from the callers context P by using π.

In Figure 4.5a, when following the trace, the region after the frontier is region 3. Therefore, in ExtendFrontier, ρ_k is the region predicate of region 3, which is $x = 2$. When calculating the exit constraint for the sum procedure shown in Figure 4.5c, we substitute all occurrences of q with $@r$ since q is the variable being assigned with the return value of the procedure call. The calculation is thus:

$$\begin{aligned}\pi(\rho_k[@r/\mathrm{v}]) &\longrightarrow \pi((x = 2)[@r/q]) \\ &\longrightarrow \pi(x = 2) \\ &\longrightarrow 1{\downarrow}x = 2\end{aligned}$$

Since there are no occurrences of q in ρ_k the predicate is unchanged by the substitution. The exit constraint ϕ_{ec} cannot directly be seen in the region graph of sum yet. One of the responsibilities of ReconstructGraphsAndInsertConstraints is to insert ϕ_{ec} at the correct places and it will alter the predicate slightly. ReconstructGraphsAndInsertConstraints is described in the next section.

4.4.4 ReconstructGraphsAndInsertConstraints

The ReconstructGraphsAndInsertConstraints procedure has multiple responsibilities:

- Generate a fresh set of region graphs $\mathcal{G}'$ used for the analysis of the called procedure P'.
- Insert the input constraint ϕ_{ic} in the initial region of G', the region graph for the called procedure P'.
- Split all return regions in G' in two using ϕ_{ec}, and marking one of them as an error region.

The reason that fresh region graphs are needed for the analysis of P', is that the graphs used in the current analysis may contain concrete executions that are impossible under the input constraint ϕ_{ic} of P'. Therefore DashLoop needs to start from a fresh region graph where these states have been removed. DashLoop may also refine the region graph G' when analyzing the called procedure P'. Such refinements are only relevant for the analysis of the called procedure and are irrelevant for the analysis of any other procedures, and must therefore be discarded when DashLoop returns.

Inserting the input constraint is easy, as it is simply added on the initial region of G'. Inserting the exit constraint involves a bit more work. Each **return** e region in G' is split into two, which we denote as S_{ec} and $S_{\neg ec}$. The exit constraint ϕ_{ec} is transformed by substituting the result placeholder $@r$ with the **return** expression e. The region S_{ec} has the predicate $\phi_{ec}[\mathrm{e}/@r]$ added while $S_{\neg ec}$ gets the predicate $\neg\phi_{ec}[\mathrm{e}/@r]$. Additionally the region S_{ec} is marked as an error region. Thus if the error region is reached, then conceptually the region predicate ρ_k, from the region after the frontier, is satisfied.

The exit constraint calculated for the sum procedure call in Figure 4.5a was $1{\downarrow}x = 2$. The initial region graph, where the input and exit constraints have not been added yet, is shown in Figure 4.5b. The altered region graph, with input and exit constraints added, can be seen in Figure 4.5c. It can be seen that the region after the **return** s statement has been split into two. The exit constraint that is inserted for the **return** s statement is:

$$\begin{aligned} \phi_{ec}[\mathrm{e}/@r] &\longrightarrow (1{\downarrow}x = 2)[\mathrm{s}/@r] \\ &\longrightarrow 1{\downarrow}x = 2 \end{aligned}$$

The region where $1{\downarrow}x = 2$ has been added as a predicate is marked as an error region, whereas the region with the negated version $1{\downarrow}x \neq 2$ is a not an error region. This can be verified by examining Figure 4.5c.

Alternatively, had the assigned variable q instead been x, such that the procedure call had been x = sum(x, y), then the exit constraint ϕ_{ec} computed would have been $@r = 2$. Computing the predicate for the **return** s statement for this constraint becomes:

$$\begin{aligned} \phi_{ec}[\mathrm{e}/@r] &\longrightarrow (@r = 2)[\mathrm{s}/@r] \\ &\longrightarrow \mathrm{s} = 2 \end{aligned}$$

Thus, for the **return** s statement, every occurrence of x in ρ_k is replaced with s, through the intermediate variable $@r$. This happens since it is the value of s that is assigned to x if **return** s is reached in P'.

When the input and the exit constraint has been inserted in G', then it is ready for DashLoop to analyze. The complete set of region graphs $\mathcal{G}'$ is returned from ReconstructGraphsAndInsertConstraints, which contains G'. The call to DashLoop is made in line 11. When DashLoop returns, the result needs to be extracted, which is explained in the next section.

4.4.5 Extracting results from interprocedural analysis

ExtendFrontier invokes DashLoop on line 11 in Algorithm 4.3 with an instrumented safety property added to the region graph of the called procedure P'. The safety property was constructed such that if DashLoop reports FAIL, then ExtendFrontier is able to return a test input that crosses the frontier edge. If DashLoop returns PASS, ExtendFrontier can construct a refinement predicate from the returned graph.

First we show how to extract the test input to the caller procedure P from the test input reported by DashLoop, which reaches an error region in procedure P'. Afterwards we show how to compute a refinement predicate from the region graph that was given as a proof by DashLoop, which shows that the error regions were unreachable in P'.

Extracting test input

At line 13 in Algorithm 4.3 the test input for the caller procedure P is extracted. The variable z contains the test input that reaches an error region in the sub procedure P'. The test input z contains, as seen from the view of P':

- Initial symbolic variables v_0 for all parameters of P'.
- Values for the variables mentioned in the input constrain ϕ_{ic}:
  - Parameters and local variables for the caller procedure P.
  - Initial symbolic variables for parameters for the caller procedure P.
  - Variables mentioned in the input constraint for P (if any).

The values needed to run a test for P are the initial symbolic variables for the parameters of P and all the variables mentioned in the input constraint for

P. Thus, the input t for P is extracted from the test input z to P' with the pseudocode:

$$t := \pi^{-1}\Big(z \setminus \{v_0 \mid \forall v \in \mathsf{params}(P')\}\Big) \setminus \Big(\mathsf{locals}(P) \cup \mathsf{params}(P)\Big)$$

First the initial symbolic variables for the parameters for the called procedure P' are removed from z. These are variables the values that needs to be given to P', such that it ends up in an error region, but they are irrelevant for calling P. The result is then reverse renamed with π^{-1} and the locals and parameters for the caller procedure P is removed. These variables might occur in z because they may have been mentioned in the input or exit constraints for P'. What is left behind is the initial symbolic values v_0 for the parameters of P generated by ExecuteSymbolic together with any external variables that might have been given in an input constraint when analyzing P as a sub procedure.

We present an example to make extraction of the test input clearer. A call was made to sum from the test procedure in Figure 4.5a. The call to DashLoop is able to analyze the region graph in Figure 4.5c and come up with a test input z that reaches the error region:

$$[1{\downarrow}x_0 \mapsto 2,\ 1{\downarrow}y \mapsto 4,\ 1{\downarrow}x \mapsto 2,\ i_0 \mapsto 4,\ x_0 \mapsto 4]$$

First ExtendFrontier removes all initial symbolic variables for the parameters of the called procedure P' from z:

$$\begin{aligned}
z \setminus \{v_0 \mid \forall v \in \mathsf{params}(P')\} &\longrightarrow z \setminus \{v_0 \mid \forall v \in \{i, x\}\} \\
&\longrightarrow z \setminus \{i_0, x_0\} \\
&\longrightarrow [1{\downarrow}x_0 \mapsto 2,\ 1{\downarrow}y \mapsto 4,\ 1{\downarrow}x \mapsto 2, \\
&\qquad i_0 \mapsto 4,\ x_0 \mapsto 4] \setminus \{i_0, x_0\} \\
&\longrightarrow [1{\downarrow}x_0 \mapsto 2,\ 1{\downarrow}y \mapsto 4,\ 1{\downarrow}x \mapsto 2]
\end{aligned}$$

Parameters and local variables of P' are never mentioned in the returned test input and we therefore do not need to remove them.

The next step is to reverse rename variables using π^{-1}:

$$\begin{aligned}
\pi^{-1}\Big(z \setminus \{v_0 \mid \forall v \in \mathsf{params}(P')\}\Big) &\longrightarrow \pi^{-1}\Big([1{\downarrow}x_0 \mapsto 2,\ 1{\downarrow}y \mapsto 4,\ 1{\downarrow}x \mapsto 2]\Big) \\
&\longrightarrow [x_0 \mapsto 2,\ y \mapsto 4,\ x \mapsto 2]
\end{aligned}$$

Finally all local variables and parameters for the caller procedure P are removed. In this case there are no locals. However, local variables will be present if 1) a local variable is used in the call to a procedure or 2) if the local variable is mentioned in ρ_k and is not the variable being assigned with the result of the procedure call. There are two parameters to test and these are x and y. Removing these, the final test input t is found:

$$\begin{aligned}
t &\longrightarrow \pi^{-1}\Big(z \setminus \{v_0 \mid \forall v \in \mathsf{params}(P')\}\Big) \setminus \Big(\mathsf{locals}(P) \cup \mathsf{params}(P)\Big) \\
&\longrightarrow [x_0 \mapsto 2,\ y \mapsto 4,\ x \mapsto 2] \setminus \Big(\emptyset \cup \{x, y\}\Big) \\
&\longrightarrow [x_0 \mapsto 2]
\end{aligned}$$

Thus, the test input returned from ExtendFrontier dictates that the initial value for x must be 2. The y variable is not constrained and when running a test it is given a default value of 0. By inspecting the region graphs in Figure 4.5, it can be confirmed that running a test with $x \mapsto 2$ will reach the region after the frontier.

In this case DashLoop was able to reach an error region in the called procedure P'. The next section presents how ExtendFrontier extracts a refinement predicate if DashLoop shows that the error regions in P' are unreachable.

Computing the refinement predicate

If DashLoop is not able to find a test input that reaches an error region in a sub procedure P', then it returns a graph z as a proof that the error regions were unreachable. The goal is then to compute a suitable refinement predicate ρ from the graph z, such that the trace τ_c is eliminated in the caller procedure P.

Remember that DashLoop was given a graph where input and exit constraints where added, such that reaching an error region in P' would correspond to τ_c being able to cross the frontier. At this point, DashLoop has refined the graph, such that no error regions are reachable in P' and the goal is to compute a suitable refinement predicate that shows that the trace τ_c is infeasible in P.

For a refinement predicate ρ to be suitable with respect to τ_c in P, it must not be possible to reach S_k from S_{k-1} where $\neg\rho$ is added. Furthermore, all states that can be obtained by executing τ_c up to the frontier must belong to the region with $\neg\rho$ added.

The key observation is to discover that when the initial region, in the graph given to DashLoop, is refined, then the used refinement predicate ρ_i is a predicate that must be satisfied for a trace in P' to reach an error region. Thus, if the predicate $\neg\rho_i$ is satisfied, then a path to an error region becomes unreachable, and thus some path to S_k in P is also unreachable. Therefore all the predicates ρ_i used to refine the initial region are interesting. If a path to an error region is removed prior to reaching the initial region, then the predicates used there are irrelevant to the trace τ_c since even if the input constraint was set to **true**, then the error region could still not be reached. This is evident since the input constraint cannot make a region predicate **false**. Thus, it seems that we can construct a refinement predicate for eliminating τ_c in P by using the predicates ρ_i used to refine the initial region in the region graph z returned by DashLoop.

We now need to find the suitable predicate ρ, such that if $\neg\rho$ is satisfied in region S_{k-1}, then region S_k cannot be reached in P and all states that can be obtained by executing τ_c in P belongs to the region with $\neg\rho$ added. Because RefinePred does not use the loop optimization on the initial region, as described in Section 3.6.3, all the ρ_i predicates are suitable refinement predicates for traces in P'.

If we assume that the names of the parameters and arguments are the same, then we can ignore argument/parameter substitution and renaming. Then, formally we have:

- Each $\neg\rho_i$ eliminates a path to an error region. Therefore $\neg\rho_0 \wedge \ldots \wedge \neg\rho_n$

eliminates all paths to all error regions in the sub procedure, and thereby eliminates all paths to region S_k in the calling procedure P.

- Because ρ_i is used to refine the initial region in P', and ρ_i is a suitable predicate, then, by the definition of a suitable predicate, all states that can be satisfied by the input constraint on the initial region must all satisfy $\neg\rho_i$. The conjunction of the predicates $\neg\rho_0 \wedge \ldots \wedge \neg\rho_n$ must also be satisfied by the states that satisfy the input constraint. Since the input constraint models all the states that can be obtained by τ_c in P, then all the states that can be obtained by τ_c must satisfy $\neg\rho_0 \wedge \ldots \wedge \neg\rho_n$.

The conjunction $\neg\rho_0 \wedge \ldots \wedge \neg\rho_n$ must then collectively remove all paths down to the error regions, and all states that can be obtained by the input constraint must satisfy the predicate. Thus we have found that $\neg\rho$, the negated refinement predicate for τ_c in P, is $\neg\rho_0 \wedge \ldots \wedge \neg\rho_n$. To find the suitable refinement predicate ρ for P, we only have to negate the constraint:

$$\begin{aligned}
\rho &\longrightarrow \neg(\neg\rho) \\
&\longrightarrow \neg(\neg\rho_0 \wedge \ldots \wedge \neg\rho_n) \\
&\longrightarrow \rho_0 \vee \ldots \vee \rho_n \\
&\longrightarrow \bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i
\end{aligned}$$

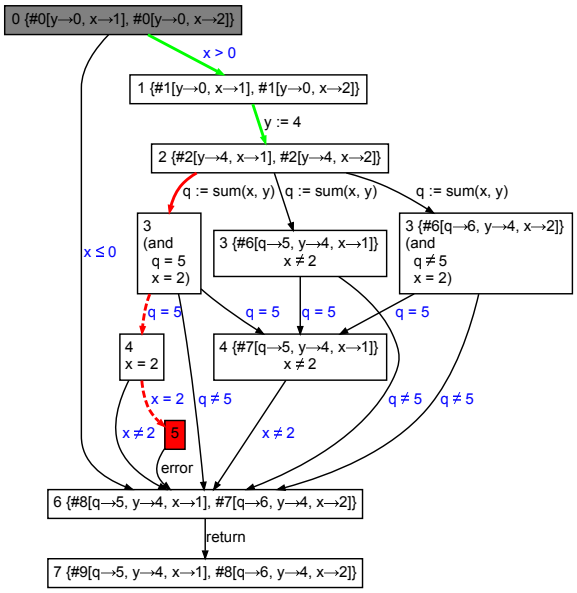
Notice that the predicates used to refine the initial region, which are those returned by $\mathsf{InitialRefines}(z)$, are not directly found in z, and have to be maintained separately.

If we remove the assumption that argument and parameter names are equal, we have to take argument/parameter substitution and renaming into account. Argument/parameter substitution is reversed by replacing all symbolic variable names with their corresponding argument. To reverse the renaming, π^{-1} is used. Computing the refinement predicate then becomes:

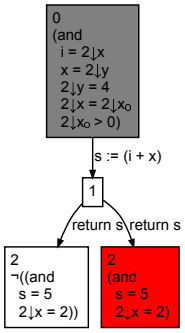
$$\rho := \pi^{-1}\left(\left(\bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i\right)\Big[a_0/v_0, \ldots, a_n/v_n\Big] \mid v_i \in \mathsf{params}(P')\right)$$

which is the computation performed in line 17 of Algorithm 4.3. Notice that, if the initial region is never refined in the region graph z, then all paths to the error regions have been pruned before reaching the initial region. In that case, the computation above yields **false**, which is the expected behavior since then no concrete execution of τ_c in P can reach region S_k.

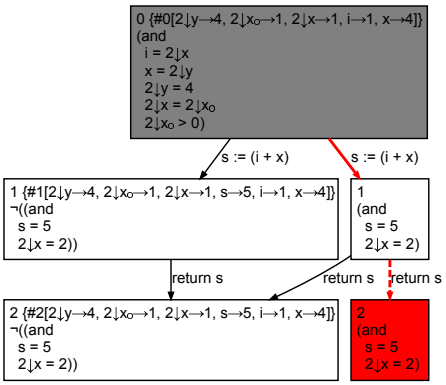
We will use the procedure call to sum in Figure 4.6a, which shows a later iteration of DASH$_{call}$, as an example to illustrate how the refinement predicate ρ is computed. The initial region graph constructed for sum is shown in Figure 4.6b, and it is impossible to reach the error region in it. This is because the input variable x is required to be 4, because of the input constraint, and the i variable is required to be 2, because of the exit constraint. However, the sum of these



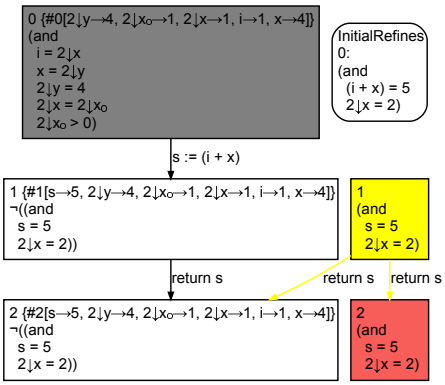
(a) Procedure call with two constraints in ρ_k which is the predicate on the region after the frontier edge.



(b) ϕ_{ic} and ϕ_{ec} added to the initial region graph.



(c) The only iteration where the initial region is refined.



(d) Frontier edge from (c) has been removed.

Figure 4.6: (a) shows an interesting procedure call in regards to computing a refinement predicate. The region graphs (b), (c) and (d) shows parts of the refinement process completed by DashLoop when invoked by ExtendFrontier.

two variables stored in s must be 5 if the error region is to be reached, but the sum is always 6. Therefore it is impossible to reach the error region.

The refinement predicate returned from ExtendFrontier is based on the predicates that were used to refine the initial region. Figure 4.6c shows the only trace that results in refinement of the initial region. The trace is infeasible since the path constraint ϕ after symbolic execution contains $i_0 + x_0 = 5 \wedge x_0 = 4 \wedge i_0 = 2$, which is unsatisfiable. The refinement predicate used to refine the initial region is computed by using the weakest precondition for the assignment

s := i+x on the frontier edge and the predicate from the region after the frontier:

$$\mathsf{WP}(\text{s := i+x}, s{=}5 \wedge 2{\downarrow}x{=}2) \quad \longrightarrow \quad i{+}x{=}5 \wedge 2{\downarrow}x{=}2$$

Refining the initial region with the above predicate results in Figure 4.6d. Notice that, as previously described in Section 3.3.5, when refining the initial region over some edge then that edge is simply removed. An extra requirement is added to RefineGraph, such that if it refines the initial region with a predicate, then that predicate is stored in a list on the graph for later retrieval. This is described in Section 4.5. In the example, the list of predicates that are used to refine the initial region can be seen in the round box with the name "InitialRefines" in Figure 4.6d.

When DashLoop returns with the resulting graph in z, all the predicates ρ_i used to refine the initial region is returned by InitialRefines. The predicates are combined in a disjunction, and in the example this yields:

$$\begin{aligned}
\bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i \quad &\longrightarrow \quad \bigvee_{\rho_i \in \{(i{+}x{=}5 \wedge 2{\downarrow}x{=}2)\}} \rho_i \\
&\longrightarrow \quad \bigvee i{+}x{=}5 \wedge 2{\downarrow}x{=}2 \\
&\longrightarrow \quad i{+}x{=}5 \wedge 2{\downarrow}x{=}2
\end{aligned}$$

Again, these predicates are not directly found in the region graph, and have to be maintained separately by RefineGraph. In this example the initial region was refined only once, but had it been refined multiple times, then InitialRefines would have returned all the predicates used to refine the initial region.

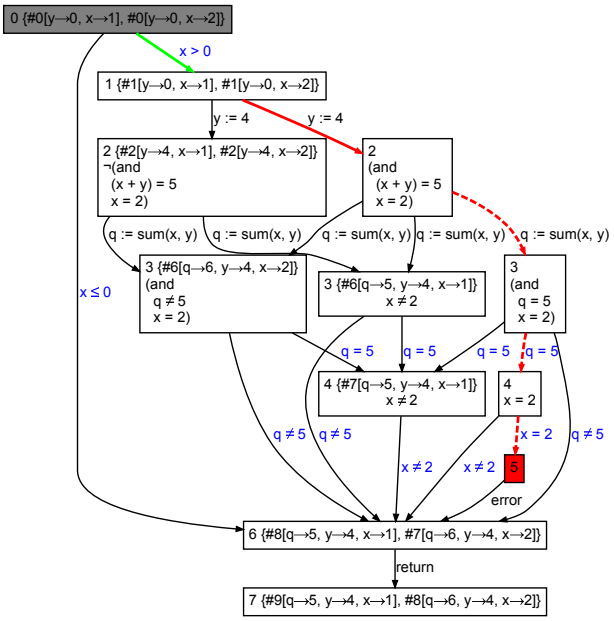
The next step is to substitute all parameters of the called procedure P', with the actual values given in the call. For the call, the variable x is given as the parameter i and the variable y is given for the x parameter. The resulting computation is:

$$\begin{aligned}
&\left(\bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i\right)\Big[a_0/v_0, \ldots, a_n/v_n\Big] \;\mid v_i \in \mathsf{params}(P') \\
&\longrightarrow \quad \Big(i{+}x{=}5 \wedge 2{\downarrow}x{=}2\Big)\Big[a_0/v_0, \ldots, a_n/v_n\Big] \;\mid v_i \in \{i, x\} \\
&\longrightarrow \quad \Big(i{+}x{=}5 \wedge 2{\downarrow}x{=}2\Big)\Big[x/i, y/x\Big] \\
&\longrightarrow \quad x{+}y{=}5 \wedge 2{\downarrow}x{=}2
\end{aligned}$$

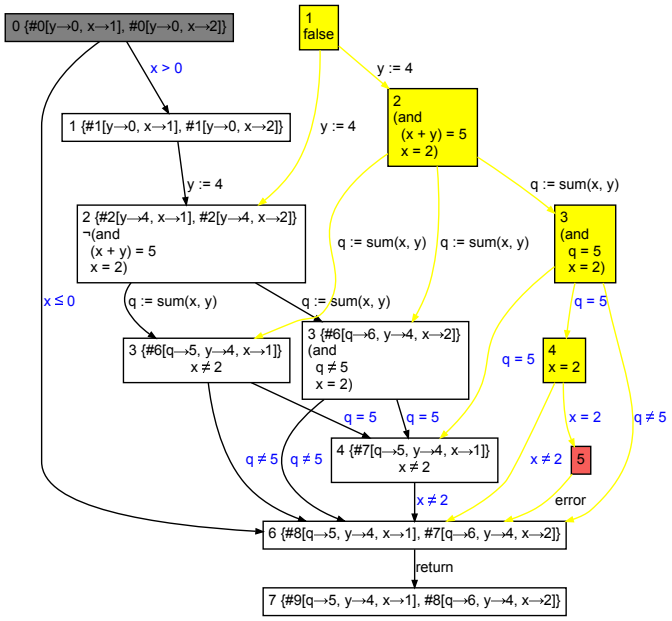
The last step is to reverse rename the variables using π^{-1}:

$$\begin{aligned}
\rho := \pi^{-1} &\left(\left(\bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i\right)\Big[a_0/v_0, \ldots, a_n/v_n\Big] \;\mid v_i \in \mathsf{params}(P')\right) \\
&\longrightarrow \quad \pi^{-1}\Big(x{+}y{=}5 \wedge 2{\downarrow}x{=}2\Big) \\
&\longrightarrow \quad x{+}y{=}5 \wedge x{=}2
\end{aligned}$$

This is the refinement predicate ρ returned by ExtendFrontier. The idea is that with this predicate, the current trace τ_c will be eliminated since it



(a) The refined region graph resulting from finding that the procedure call in Figure 4.6a was infeasible.



(b) The graph found by the next iteration of DashLoop when the trace in (a) is found infeasible.

Figure 4.7: Shows the last two refinements of the region graph in Figure 4.6a.

was unable to cross the frontier. The refinement predicate has been added in Figure 4.7a.

The next iteration of DASH$_{call}$ will try the trace seen in Figure 4.7a, which is infeasible. The refinement predicate is computed as in DASH$_{int}$, since the frontier is not a procedure call. Thus WP is used over the assignment y := 4 with the postcondition $x{+}y{=}5 \wedge x{=}2$:

$$\begin{aligned} \mathsf{WP}(\text{y} := 4, x{+}y{=}5 \wedge x{=}2) &\longrightarrow x{+}4{=}5 \wedge x{=}2 \\ &\longrightarrow \textbf{false} \end{aligned}$$

As can be seen, the refinement predicate is equivalent to **false**. When the refinement predicate is used to split region 1, the region graph shown in Figure 4.7b is created. The error region is unreachable and the analysis stops with DashLoop returning PASS. This concludes the ExtendFrontier presentation.

4.5 RefineGraph

RefineGraph is subject to two changes. First the predicates used to split the initial region needs to be recorded. In Algorithm 4.4 the procedure AddInitialRegionSplitPredicate is used to store those predicates. It has been placed together with the optimization for refining the initial region as described in Section 3.3.5.

The second change is to distribute the states in S_{k-1} into S^{*}_{k-1} and S^{**}_{k-1}, since ρ is not as strong as it was in DASH$_{int}$. In DASH$_{int}$ we knew that ρ would always exclude all the known states in S_{k-1}. In DASH$_{call}$ we only know that it is a suitable predicate, which implies that it will only exclude the states associated with the current trace τ_c. States that are not associated with τ_c, but still resides in S_{k-1} may be able to satisfy ρ and would therefore be placed in S^{**}_{k-1}. Remember that there are no requirements that say if ρ is satisfied, then S_k is reached. However, what is required is that if ρ is *not* satisfied, then the error region cannot be reached. The states that are added to S^{**}_{k-1} still does not reach S_k, since then the frontier would not have been at the edge (S_{k-1}, S_k).

4.6 Finding error statements in sub procedures

As mentioned in the start of this chapter, DASH$_{call}$ only finds the error statements in the top level procedure of a multi-procedure program. The DASH article specifically mentions:

> "We will assume without loss of generality that the property ϕ that we wish to check is only associated with the main procedure P_0 in the program $\mathcal{P}$."[1]

The DASH algorithm, as described in the DASH article, supports pointers into the heap and as such global state. Using global state, the limitation that error statements can only occur in the main procedure can be avoided with a clever instrumentation. We believe that the original DASH authors thought of

[1] Page 11 in DASH [A2]

Algorithm 4.4 RefineGraph($\rho, \tau_c = \langle RS_0, \ldots, RS_{k-1}, S_k \rangle, G = \langle \Sigma_\simeq, \rightarrow_\simeq \rangle$)

Returns: $\langle \Sigma_\simeq, \rightarrow_\simeq \rangle$, the refined region graph.

let $\langle S_{k-1}, \_ \rangle = RS_{k-1}$
let $\langle \rho_{k-1}, states \rangle = S_{k-1}$

if $k = 1$ **then**
    AddInitialRegionSplitPredicate(ρ)
    return $\langle \Sigma_\simeq, \rightarrow_\simeq \setminus (S_{k-1}, S_k) \rangle$ ▷ Since k = 1, this is the same as removing (S_0, S_1)
end if

$\Sigma^*_\simeq := \Sigma_\simeq \setminus \{S_{k-1}\}$ ▷ Remove S_{k-1}
$\rightarrow^*_\simeq := \rightarrow_\simeq \setminus \{(S, S_{k-1}) \mid S \in \mathsf{Parents}(S_{k-1})\}$
$\rightarrow^*_\simeq := \rightarrow^*_\simeq \setminus \{(S_{k-1}, S) \mid S \in \mathsf{Children}(S_{k-1})\}$

$\rho^*_{k-1} := \mathsf{Simplify}(\rho_{k-1} \wedge \neg\rho)$
$\rho^{**}_{k-1} := \mathsf{Simplify}(\rho_{k-1} \wedge \rho)$
for $s \in states$ **do** ▷ Distribute states between S^*_{k-1} and S^{**}_{k-1}
    if $\mathsf{Eval}(\rho^*_{k-1}, s) =$ **true then**
        $states^* := states^* \cup \{s\}$
    else
        $states^{**} := states^{**} \cup \{s\}$
    end if
end for

$S^*_{k-1} := \langle \rho^*_{k-1}, states^* \rangle$
$\Sigma^*_\simeq := \Sigma^*_\simeq \cup \{S^*_{k-1}\}$ ▷ Insert S^*_{k-1}
$\rightarrow^*_\simeq := \rightarrow^*_\simeq \cup \{(S, S^*_{k-1}) \mid S \in \mathsf{Parents}(S_{k-1})\}$
$\rightarrow^*_\simeq := \rightarrow^*_\simeq \cup \{(S^*_{k-1}, S) \mid S \in \mathsf{Children}(S_{k-1})\}$

$S^{**}_{k-1} := \langle \rho^{**}_{k-1}, states^{**} \rangle$
$\Sigma^*_\simeq := \Sigma^*_\simeq \cup \{S^{**}_{k-1}\}$
$\rightarrow^*_\simeq := \rightarrow^*_\simeq \cup \{(S^{**}_{k-1}, S) \mid S \in \mathsf{Children}(S_{k-1})\}$
if $\mathsf{IsSAT}(\rho^{**}_{k-1}) =$ UNSAT **then** ▷ Add incoming edges if ρ^{**}_{k-1} is satisfiable
    $\rightarrow^*_\simeq := \rightarrow^*_\simeq \cup \{(S, S^{**}_{k-1}) \mid S \in \mathsf{Parents}(S_{k-1})\}$
end if

$\rightarrow^*_\simeq := \rightarrow^*_\simeq \setminus \{(S^*_{k-1}, S_k)\}$ ▷ Remove frontier edge from S^*_{k-1}
return $\langle \Sigma^*_\simeq, \rightarrow^*_\simeq \rangle$

this instrumentation when they wrote *without loss of generality*. Essentially all calls to error in sub procedures are replaced with an assignment ERROR = 1 to a global variable ERROR. The main procedure is then instrumented to check if ERROR has been set to 1. If this is the case, then the main procedure executes the error statement to signal the failure.

The instrumentation process is illustrated in Figure 4.8. Figure 4.8a shows two procedures where main is the main procedure and sub is a sub procedure called by main. Notice that sub uses the error statement, which DASH does not support. Figure 4.8b shows the same code, but instrumented to use a global variable ERROR to signal to the main procedure when sub has reached an error. As such it does not matter that only error statements in the main procedure are searched for by DASH, at least not if global state is supported. However, since DASH$_{call}$ does not support global state, we cannot use this instrumentation process for programs analyzed by DASH$_{call}$.

```
int main(int x, int y)
{
  return sub(x, y);
}

int sub(int a, int b)
{
  if(a * 4 > b)
    error;
  return a + b;
}
```

(a) Before instrumentation

```
GLOBAL int ERROR = 0;

int main(int x, int y)
{
  int r = sub(x, y);
  if(ERROR == 1)
    error;
  return r;
}

int sub(int a, int b)
{
  if(a * 4 > b)
    ERROR = 1;
  return a + b;
}
```

(b) After instrumentation

Figure 4.8: (a) shows two procedures where the sub procedure uses an error statement. DASH only supports error statements in the main procedure. Using global state error statements in sub procedures can be avoided as is seen in (b).

4.7 Challenges and modifications

This section summarizes the challenges and modifications we have made to the DASH algorithm, when handling programs that support integer variables and procedure calls.

4.7.1 Infinite refinement when the frontier is a procedure call

We have had a number of problems when incorporating the loop optimization from DASH$_{int}$ while adding procedure calls. We found that some of our test cases went into an infinite refine loop, when refining over a procedure call. The problems we describe here arose before we had introduced the optimization that removes the frontier edge when refining the initial region, and it was also before we disallowed the use of the loop optimization on the initial region.

We present an example where infinite refinement occurs over a procedure call edge. The two procedures in the example are shown in Figure 4.9a and Figure 4.9b. The test procedure is the main procedure. The problem is that the same edge is continuously refined with the same input and exit constraints. This results in multiple new initial regions that have the region predicate **false** and are therefore unsatisfiable.

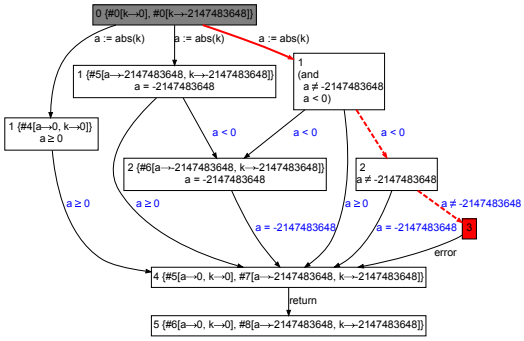
The trace in Figure 4.9c has the procedure call to abs as the frontier. DashLoop is invoked and because no input values can satisfy the exit constraint, we expect to find the refinement predicate to be, or equivalent to, **false**. No values that abs can take will satisfy the exit constraint, since as seen earlier, the only way abs can return a negative value, is by passing it $-2{,}147{,}483{,}648$ as input. However this value is specifically disallowed in the exit constraint and therefore the exit constraint $a < 0 \wedge a \neq -2{,}147{,}483{,}648$ is impossible to satisfy. The

```
void test(int k)
{
    int a = abs(k);
    if(a < 0)
        if(a ≠ −2147483648)
            error;
}
```

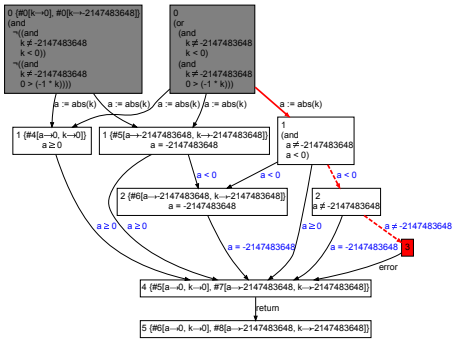
(a) test procedure

```
int abs(int z)
{
    if (z < 0)
        return −z;
    return z;
}
```

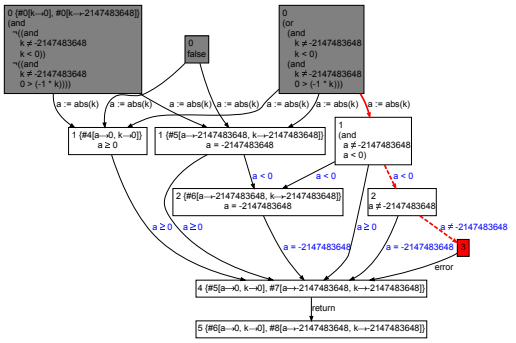
(b) abs procedure



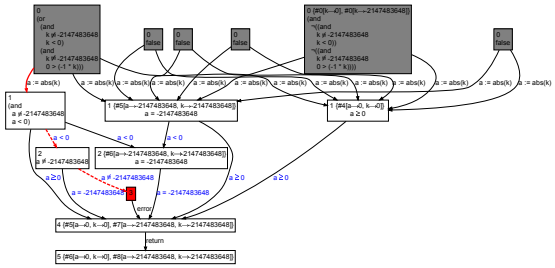
(c) About to analyze sub procedure



(d) About to analyze sub procedure



(e) First **false** region arises



(f) More **false** regions arises

Figure 4.9: Shows the problem of infinite loop refinement. Code is shown in (a) and (b). In (c) we are about to refine a procedure call edge. (d) shows the graph after refinement and together with (e) shows the infinite loop forming, due to the same input and exit constraints. (f) shows the graph after a couple of refinements and the infinite loop is in progress.

suitable refinement predicate returned should therefore be **false**, but as we can see in Figure 4.9d the found refinement predicate returned is actually $(k \neq -2{,}147{,}483{,}648 \wedge k < 0) \vee (k \neq -2{,}147{,}483{,}648 \wedge k > 0)$, which we name ρ^*.

In the next iteration of DASH the procedure call is still the frontier, but now additional input constraints are given when analyzing abs. However, DashLoop finds the same refinement predicate ρ^*, with the result of splitting the initial region in two. One with the predicate $\rho^* \wedge \neg\rho^*$ and one with the predicate $\rho^* \wedge \rho^*$. The first predicate is equivalent to **false**, with the second being identical

to the original predicate, and thus no progress is achieved. This is shown in Figure 4.9e. The refinement predicate returned is not suitable, which seems to be the root cause of the infinite refinement problem. The infinite loop has begun, where additional initial regions with the predicate **false** are added, as can be seen in Figure 4.9f.

We tried a number of techniques, which either did not work or it was unknown why they worked, to solve the problem:

- Completely disabling the loop optimization solved the problem, which also served to confirm our belief that the problem arose because we found and used a refinement predicate that was not suitable. However, without the loop optimization, some of the test cases for DASH$_{int}$ failed.
- At this point in time, our RefinePred implementation was allowed to use the loop optimization, even if there were no states in a region that was moved. We tried to add the requirement that at least one state should be moved, as argued for earlier in Section 3.8.9. This fixed some of the tests that failed, but others, like the example presented in this section, still failed.
- Another solution we came up with, was the idea that an edge going out of the initial region could simply be removed if the SAT solver found it infeasible. This solved the problem for all our test cases. However we were unable to explain why this solution should solve the problem, and we were hesitant to use it.

We found the final solution while arguing that the predicate computed by ExtendFrontier in Section 4.4.5, when DashLoop found the error regions to be unreachable in a sub procedure, was actually a suitable refinement predicate. The argument only worked if we assumed that all the refinement predicates that were used to refine the initial region in the sub procedure were suitable predicates. If the predicates were suitable, then we could show that the computed predicate would be suitable for the original procedure. This resulted in changing RefinePred such that the loop optimization could never be used when it was the initial region that was being refined. We introduced it in DASH$_{int}$, in Section 3.6, since it also fixes the problem in Section 3.8.8. In this way, the infinite refinement problem for the example presented in this section was solved. The refinement predicate returned was forced to be suitable, and therefore progress was achieved.

Our implementation still uses the idea of removing outgoing edges from the initial region when they are refined. However, we have tested that if we allow multiple initial regions, the solution where the loop optimization is disallowed on initial regions still works for all our test cases. Therefore, removing edges when refining the initial region is only an optimization that simplifies the presentation in this thesis, such that the graphs presented are smaller.

4.7.2 Lack of path constraint in the input constraint

We have had major problems adding interprocedural analysis to DASH$_{int}$. This section describes the key problems with invoking DashLoop from inside Ex-

tendFrontier. The DASH article uses approximately one page on describing modifications of ExtendFrontier and some additional helper procedures. However, we have had to modify code in many places to get DASH_{int} to handle interprocedural analysis. One of the biggest problems we have had was how to construct the input and exit constraints given to DashLoop, when invoking it from inside ExtendFrontier.

The original pseudocode for ExtendFrontier that can handle interprocedural analysis is shown in Algorithm 4.5. Section 4.7.4 describes that they take a trace that traverses only the analyzed procedure and constructs a full trace that traverses sub procedures using a call to GetWholeAbstractTrace.

Algorithm 4.5 $\mathsf{ExtendFrontier}_{original}(\tau_o, F, P)$ for interprocedural analysis

Returns:
$\langle t,$ **true**$\rangle$, if the frontier can be extended; or
$\langle$UNSAT$, \rho\rangle$, if the frontier cannot be extended

```
 1: τ_w = ⟨S_0, S_1, ..., S_n⟩ := GetWholeAbstractTrace(τ_o, F)
 2: (k − 1, k) := Frontier(τ_w)
 3: ⟨φ_1, S, φ_2⟩ := ExecuteSymbolic(τ_w, P)
 4: if Edge(S_{k−1}, S_k) ∈ CallReturn(E) then
 5:     let ⟨Σ, σ^I, →⟩ = GetProc(Edge(S_{k−1}, S_k))
 6:     φ := InputContraints(S)
 7:     φ' := S_k[e/x]
 8:     ⟨r, m⟩ := DASH(⟨Σ, σ^I ∧ φ, →⟩, ¬φ')
 9:     if r = FAIL then
10:         t := m
11:         ρ := true
12:     else
13:         ρ := ComputeRefinePred(m)
14:         t := UNSAT
15:     end if
16: else
17:     t := IsSAT(φ_1, S, φ_2, P)
18:     if t = UNSAT then
19:         ρ := RefinePred(S, τ_w)
20:     else
21:         ρ := true
22:     end if
23: end if
24: return ⟨t, ρ⟩
```

They construct the input constraint in line 6 with a call to InputConstraints giving the symbolic map $\mathcal{S}$ as the only argument. They write the following about the input constraint in the article:

> "The predicate ϕ corresponds to the constraints on Q's input variables which are computed directly from the symbolic memory $\mathcal{S}$ (by the auxiliary function InputConstraints at line 7), [...]."[2]

A small typo in their quote exists, since InputConstraints is actually called in line 6. What is more severe is that they write that the input constraint

[2] Page 11 in DASH [A2]

```
void testabs(int x, int y)
{
  if(y ≠ 0)
  {
    x = abs(x);
    if(x < 0)
      error;
  }
}
```

(a)

```
void testabs2(int x)
{
  if(x ≠ −2147483648)
  {
    x = abs(x);
    if(x < 0)
      error;
  }
}
```

(b)

```
int abs(int a)
{
  if (a < 0)
    return −a;
  return a;
}
```

(c)

Figure 4.10: (a) shows the testabs procedure which calls the abs procedure in (c) whenever $y \neq 0$. (b) shows the testabs2 procedure which has a precondition on x that it must not be the smallest negative value. Analyzing these procedures with the constraints calculated by InputConstraints produces meaningless solutions.

for the called procedure Q can be computed directly from what we call the symbolic map $\mathcal{S}$. As we know by now, the symbolic map contains mappings of the form $v \mapsto e$ where e can be a constant or a symbolic expression. Thus the constraint they generate must be in the form $p_1 = \mathsf{SymbolicEval}(a_1, \mathcal{S}) \wedge \ldots \wedge p_n = \mathsf{SymbolicEval}(a_n, \mathcal{S})$ where p_i are the parameter names and a_i is the argument given for p_i. The path constraint is missing from this input constraint. There is no way to include the path constraint from the trace τ_c by using only the symbolic map $\mathcal{S}$, since $\mathcal{S}$ only contains a snapshot of the program state at region S_{k-1}. The path constraint is located in ϕ_1 and is *not* given to InputConstraints.

Our first implementation followed their pseudocode and explanations. However, we found cases where this led to problems. We had two distinct problems:

1) When DashLoop returns with a test input for the sub procedure, then that test input might not include all the variables needed to call the callee procedure. This happens when a variable is used in **if** expression but not given as an argument to the sub procedure. An example is given in Figure 4.10a. The y variable cannot be zero if abs is to be called. However, when analyzing abs the path constraint is not added to the input constraint and the result of the analysis will not even include a value for y. If the default value of zero is used, the execution will not follow τ_c nor cross the frontier.

One can solve this problem by using an IsSAT call. The problem given to IsSAT is simply ϕ_1 and where all the arguments are assigned the solutions in t. Thus if x has to be 4, then the extra constraint $x = 4$ must be added to ϕ_1. However, this solution does not solve the next problem we discovered.

2) If some variables are constrained by the path constraint prior to a procedure call, such as in an **if** condition, then the procedure analysis might come up with an input parameter that conflicts with the path constraint, since it is not given when analyzing the called procedure. An example is given in Figure 4.10b. Here the testabs2 procedure calls the abs procedure with the argument x. At the point of the call x can be any value except for the smallest possible value $-2{,}147{,}483{,}648$. This value is the only value that allows abs to return a negative value. Therefore it is not possible to reach the error statement in the testabs2

```
void foo()
{
  int y = 4;
  int x = zero();
  if(x == y)
    error;
}
```

(a)

```
int zero()
{
  return 0;
}
```

(b)

Figure 4.11: Example where including both the path constraint and the arguments from the symbolic map is not enough for correct analysis. The whole symbolic map must be included in the input constraint to zero.

procedure. However, when analyzing the abs procedure, the requirement given is that it must return a negative value, such that the error statement can be reached. The result returned is that abs should be called with −2,147,483,648, as was discovered in the DASH$_{int}$ complete example from Section 3.7. However, the testabs2 procedure is unable to call abs with that specific value, since the path constraint prohibits it.

These two examples shows that constructing the input constraint solely from the symbolic map $\mathcal{S}$ cannot be correct. The solution to the above problems was to add the path constraint together with the arguments from the symbolic map. However, there are cases where the exit constraint contains variables that are not given in either the path constraint or the arguments to the called procedure.

The two procedures in Figure 4.11 shows that it is not enough to only add the symbolically evaluated arguments from the symbolic map to the input constraint ϕ_{ic} when analyzing the called procedure. The foo procedure can reach the error statement if x is equal y. The procedure starts by assign 4 to y and thereafter zero is called, which returns zero. Thus x is always 0 and y always 4 which makes the error statement unreachable.

Analyzing foo would proceed by executing a concrete test, which does not reach the error statement. The trace found in the next iteration has the frontier edge going into the error region. The trace is found infeasible. The next iteration has a trace where the frontier edge is over the zero procedure call. In this case the exit constraint is $x = y$, but y is not part of either the path constraint or the arguments given to the zero procedure. If a default value of zero is assigned to y since it is not constrained, then the analysis reports that the trace is feasible, which it is clearly not. The problem is that the analysis of the sub procedure cannot see that y must be 4.

Problems can also arise when a test is executed with RunTest and a variable in the exit constraint is not mentioned in either the path constraint or in arguments. IsSAT then does not find a value for the variable, which causes RunTest to be unable to evaluate a region predicate that contains the variable.

The conclusion is that DASH has to dump the whole symbolic map into the input constraint together with the path constraint. This is also how we presented the computation of the input constraint in Section 4.4.2.

We have later learned, by personal correspondence, that the DASH imple-

```
void bound(int x)
{
  int f = fib(3);
  if(f == x)
    error;

}
```

(a)

```
void free(int x)
{
  int f = fib(x);
  if(f == 1)
    if(x == 0)
      error;
}
```

(b)

```
int fib(int i)
{
  if (i == 0) return 0;
  if (i ≤ 2) return 1;
  return fib(i - 1) +
         fib(i - 2);
}
```

(c)

Figure 4.12: The bound procedure in (a) calls the recursive procedure fib given in (c) with a bound constant. DASH quickly finds a test for bound that reaches the error. The free procedure in (b) calls fib with a free unconstrained variable, which makes DASH enter an infinite recursive analysis.

mentation actually asserts the path constraint to Z3 prior to analyzing a sub procedure. In this way, they achieve the same effect as if the input constraint contained the path constraint. However, we do not know how they add constraints from the symbolic map.

4.7.3 Problem: Handling of recursive procedures

As was mentioned in the introduction to this chapter, DASH$_{call}$ is unable to handle certain types of recursive invocations. The DASH article contains the following paragraph describing how it handles recursive procedures:

> "If a procedure needs to be recursively invoked in order to reach an error condition, DASH itself will be recursively invoked, substituting appropriate values for concrete parameters, so that symbolic execution will eventually "bottom out", in the base case of the recursion. On the other hand, if the recursive execution of a procedure is not directly related to the error, the algorithm will generate test cases that pass right though the recursive invocations, at which point the call will be on the near side of the frontier."[3]

It thus seems like the DASH algorithm has no problems with recursive procedures and does not need any kind of special handling for it. For the example given in Figure 4.12a DASH quickly terminates finding the input $x \mapsto 2$, which is the value that fib(3) returns. Thus, the error region is quickly reached.

However, we have found examples where DashLoop is recursively invoked infinitely, i.e. no automatic "bottoming out" in any base cases.

The free procedure given in Figure 4.12b results in infinite analysis. The error statement cannot be reached since fib(0) is zero (x needs to be zero to satisfy the **if** condition). However, since x is not bound to any specific value, DASH keeps recursively invoking DashLoop, never bottoming out. We have not found any solution to this problem.

[3] Page 12 in DASH [A2]

4.7.4 GetWholeAbstractTrace: confusing

The ExtendFrontier procedure presented in the DASH article, reproduced as Algorithm 4.5, shows that the procedure GetWholeAbstractTrace is used. The authors describe the functionality as:

> "ExtendFrontier first calls the auxiliary function GetWholeAbstract-Trace (line 1). GetWholeAbstractTrace takes an ordered abstract error trace $\tau = \langle S_0, S_1, \ldots, S_n \rangle$ and a forest F as input, and returns an "expanded" whole abstract error trace τ_w. Essentially, τ_w is the abstract trace τ with all call-return edges up to its frontier replaced with the abstract trace traversed in the called function (and this works in a recursive manner), so that it is really a trace of every abstract program point through which the test passed. If $\mathsf{Edge}(S_i, S_{i+1})$ is a call-return edge that occurs before the frontier, GetWholeAbstractTrace runs a test t (obtained from the concrete witness in S_i) on the called procedure $\mathsf{GetProc}(e)$ and replaces $\mathsf{Edge}(S_i, S_{i+1})$ with the sequence of regions corresponding to the test t."[4]

Thus they create a trace that passes through every program point in both the main procedure and through the called sub procedures. There are multiple problems to this approach:

- The trace τ_w is a list of regions where all call-return edges (S_i, S_{i+1}) have been replaced with $S_{t1}, \ldots, S_{tn}$ originating from the test t. It is not given which operation exists on the edges (S_i, S_{t1}) and (S_{tn}, S_{i+1}).
- A call return edge (S_i, S_{i+1}) cannot be replaced without explicit handling of arguments and parameters. If x is passed to a procedure as the parameter y then there must be an assignment of some form to convey that x and y are connected. The same problem exists for **return** statements.
- What happens when there are two procedures that both takes x as a parameter. They do not touch upon renaming of variables while describing GetWholeAbstractTrace.
- As we have understood the DASH algorithm, they maintain a forest F for each procedure in the program. They write "As in the single procedure case, we maintain a forest F and an abstraction $P_{\simeq}$ for every procedure P in the program."[5] They should therefore be able to lookup the correct path in one of the forests and thereby not need to execute a test t to figure out the path taken by τ in the sub procedure.

They also mention that only ExtendFrontier needs to be altered for interprocedural analysis to function. Given the above listed problems we cannot see how ExecuteSymbolic should be able to execute τ_w without modifications. Also, as should have been evident from our presentation, we have concrete states

[4] Page 11 in DASH [A2]
[5] Page 11 in DASH [A2]

attached to the region graph of sub procedures such that following the trace τ was a simple task for us.

Our pseudocode for ExecuteSymbolic, shown in Algorithm 4.1, handles: 1) following the trace, 2) arguments/parameters, 3) return values and finally it uses a symbolic map for each sub procedure such that renaming is not needed. Our first implementation actually did perform renaming, but when we wrote down the pseudocode for it, we found that renaming could be avoided and recoded our implementation.

All in all, it seems that GetWholeAbstractTrace is a procedure that is used to compact the presentation of the DASH interprocedural algorithm. The real DASH implementation probably does something similar to what we do.

4.7.5 Consequences of having states on regions

The DASH authors maintains a forest F for every procedure, as is evident in the quote below:

> "As in the single procedure case, we maintain a forest F and an abstraction $P_{\simeq}$ for every procedure P in the program."[6]

A major difference between our implementation and their pseudocode is that our version keeps the abstraction $P_{\simeq}$ and the forest F together in the region graph data structure. They invoke DashLoop with an empty forest and a fresh abstraction. Thus, when we need to invoke DashLoop all the concrete states must be removed from the region graph, such that the sub procedure gets a fresh start.

4.7.6 ComputeRefinePred: confusions about ρ_i and $\neg$

In the DASH article, the procedure ComputeRefinePred is responsible for computing the refinement predicate for P in ExtendFrontier when DashLoop has proved that no error regions are reachable in the sub procedure P'. Our computation for ComputeRefinePred differs from the one presented in the DASH article. We have used:

$$\rho := \bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i$$

given that we ignore reverse renaming and argument/parameter substitution. We have argued in Section 4.4.5 for why this is a suitable predicate that can be used to eliminate τ_c in P. In the article the authors use the following definition:

> "Specifically, ComputeRefinePred(m) is defined as follows.
>
> $$\mathsf{ComputeRefinePred}(m) := \neg \bigvee \rho_i$$
>
> where each ρ_i is a predicate in the proof m used to split the initial region $\sigma^I \wedge \phi$. It can be shown that ComputeRefinePred returns a suitable predicate."[7]

[6] Page 11 in DASH [A2]
[7] Page 12 in DASH [A2]

Immediately it can be seen that we have removed a negation ($\neg$) from the definition. Additionally we have discovered, by personal correspondence, that the ρ_i they use in their definition are not actually the predicates returned by RefinePred, but are the predicates that would be attached to the initial region. Those predicates are the negated versions of those returned by RefinePred. Thus, their ρ_i, which we call $\rho_{i_{their}}$, is actually the negated version of our ρ_i's, which we can call $\rho_{i_{our}}$. We can express the difference by the equation $\rho_{i_{their}} = \neg\rho_{i_{our}}$. We assumed that ρ_i referred to the suitable predicate returned by RefinePred, since ρ has been used as such in the previous sections of the article. If we insert their definition of ρ_i in ComputeRefinePred we get:

$$\begin{aligned}\mathsf{ComputeRefinePred}(m) &:= \neg \bigvee \rho_{i_{their}} \\ &:= \neg \bigvee \neg\rho_{i_{our}} \\ &:= \bigwedge \rho_{i_{our}}\end{aligned}$$

This still does not match our definition. They use a conjunction where we use a disjunction. What is worse, is that this predicate is not a suitable predicate for τ_c in the caller procedure P. Remember, that for a predicate to be suitable with respect to τ_c, the negated version must disallow any state to reach the region after the frontier. This is a requirement since then we are allowed to remove an edge when splitting S_{k-1} into two. However, their negated version:

$$\neg \bigwedge \rho_{i_{our}} \quad \longrightarrow \quad \bigvee \neg\rho_{i_{our}}$$

basically means that if a single $\rho_{i_{our}}$ is not satisfied, it is not possible to reach S_k. However, it might be possible to reach one of the error regions in P' as long as one of the $\rho_{i_{our}}$'s is satisfied. Therefore, their predicate is not a suitable refinement predicate. This makes a difference when the initial region in P' has been refined multiple times, such that there is more than one ρ_i. It also makes a difference when the initial region in P' has not been refined at all. Then their predicate becomes **true**, since it is neutral element for conjunction, but with **true**, progress is not achieved.

We do not believe that they use the above formula. There must be some mismatch in the communication between us and the authors of DASH, either in the article or by personal correspondence.

Chapter 5

Implementation details when implementing DASH$_{call}$ in Java for analyzing a subset of Java

This chapter describes the implementation details of DASH$_{call}$. We have named the implementation DASH for Java, or in short form DASH4j. DASH4j takes as input a small Java program and analyses it as DASH$_{call}$ would. Our implementation code is structured to resemble the pseudocode presented for DASH$_{call}$. There are some practical obstacles that need to be solved before all the requirements for the DASH$_{call}$ algorithm are met:

- Java code needs to be loaded. There is a difference between what DASH supports and what Java supports. For example, DASH does not support expressions with side effects, but Java does with the syntax i++.
- A SAT solver is needed for the IsSAT procedure.
- A region graph must be constructed for DASH analysis.
- RunTest must be able to execute a test and collect states for the region graph.

The supported Java features are described in Section 5.1. Desugaring of Java code is described in Section 5.2 whereas construction of region graphs is presented in Section 5.3. We use instrumentation when concretely executing a test, which we describe in Section 5.4. Z3 is used as the SAT solver and integrating with it is described in Section 5.5. We elaborate on handling the Java Standard Library in Section 5.6. Finally, in Section 5.7 we describe how the source code for DASH4j can be obtained and we provide an overview of the tests we have written.

5.1 Supported Java features

In this section we describe the features in Java that are supported by the DASH4j implementation of DASH$_{call}$.

DASH4j supports static methods with integer arithmetic, parameters and local variables. Inside methods, assignments and any of the normal control flow constructs are allowed: **if**, **while**, **do while** and **for**, since they can be mapped to the **if** c **goto** l construct that DASH$_{int}$ supports.

DASH$_{call}$, as described in Chapter 4, introduced procedure calls. In Java this is mapped to static method calls. The result of a method call must be assigned to a variable, as required by DASH$_{call}$. The reason for using static methods, as opposed to using instance methods, is that instance methods always take a hidden parameter before the actual parameters. The hidden parameter is the **this** reference to the object instance that the method is called on. It is visible during analysis, but could have been ignored, and we did this initially, but static methods maps cleaner to the DASH concept of a procedure.

Other features of Java are not currently supported. Thus features such as doubles, long, exceptions, objects and reflection are not supported. Significant subsets of Java features are discussed as future extensions in Chapter 6.

5.2 Loading Java code

There are two choices for the input to DASH4j: either Java source code or bytecode. When Java source code is compiled it generates **.class** files containing bytecode. Operating on any of these two choices is not optimal since:

- Java source code
  - Has irrelevant syntactic sugar. Examples are generics, **for**, **while**, foreach-loops, inner classes, anonymous classes, varargs, String + operator, implicit toString(), implicit String.valueOf(...), etc. Each construct would have to be handled.
  - Allows unlimited nested expressions: q=(a+b)+(c−v()).
  - Source code may not be available for compiled classes.
- Java bytecode
  - Has at least 202 bytecodes[1] that needs to be handled individually.
  - Uses a stack based model, where it can be difficult to associate stack entries with variables.
  - Very low level.

Neither of these maps well to the language constructs used by DASH. Instead we use a tool called Soot[2]. It converts Java bytecode into a simple intermediate language called Jimple. Jimple is very similar to the language that DASH supports:

- All **if**, **while**, **do while** and **for** is converted to simplified form, with **if** c **goto** l and **goto** l statements.

[1] http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-7.html
[2] http://www.sable.mcgill.ca/soot/

```
static int add
 (int a,int b,int c)
{
  return a + b + c;
}
```

(a) Java

```
static int add
 (int,int,int)
   iload_0
   iload_1
   iadd
   iload_2
   iadd
   ireturn
```

(b) Bytecode

```
static int add
 (int,int,int)
{
  int a, b, c, temp$0,
   temp$1, temp$2;

  a := @parameter0: int;
  b := @parameter1: int;
  c := @parameter2: int;
  temp$0 = a;
  temp$1 = temp$0 + b;
  temp$2 = temp$1 + c;
  return temp$2;
}
```

(c) Jimple

Figure 5.1: The code listed in (a) shows the Java source code of a static method add that takes three integers and returns their sum. (b) shows add when compiled to bytecode. (c) shows add when converted to a simple 3-address representation called Jimple.

- Expressions are represented using 3-address representation. For example, the expression q=(a+b)+c will be replaced by z=a+b; q=z+c.
- If method calls are located inside expressions, then they are extracted and their results are assigned to an intermediate variable.
- Arguments to method calls are either constants or local variables. If this is not the case, the argument expression is converted to 3-address representation and the result is used as the argument.

In essence, Jimple is Java but where all syntactic sugar has been removed. Jimple was designed for ease of use for algorithms such as DASH.

An example Java program, the bytecode and its Jimple counterpart can be seen in Figure 5.1. It should be obvious that detecting various operations from bytecode are cumbersome and non-trivial, Soot simplifies this considerably.

We therefore use the Soot tool to load Java bytecode as Jimple. It should be noted that our implementation uses a special static procedure error() that functions as the DASH error statement. We could have used **assert** statements, but the representation in Java bytecode and Jimple is more complicated. It depends on a static field to check whether **assert** statements are enabled and an exception is thrown if an assertion is violated. Instead we have chosen to represent the error statement by a call to a static error() method. The next section describes how a region graph is constructed from Jimple source.

5.3 Construction of a region graph from Jimple

Given a program in Jimple, we need to construct a region graph for each procedure in the program. The Soot tool supports creating a control flow graph

(CFG) for a Jimple program. As mentioned earlier, the initial region graph is equivalent to the CFG. However, the CFG that Soot creates has some important differences compared to the region graph:

- Nodes in the Soot CFG contain statements whereas nodes in the region graph represent equivalence classes.
- Edges in the CFG models control flow whereas the edges in the region graph models possible state changes by executing statements.
- The conditional **if** c **goto** l statement must be converted to two **assume** statements. One edge that points to the destination l and contains an **assume** c statement. The other edge represents falling through to the next statement with an **assume** ¬c statement.
- The CFG contains plain **goto** l statements. These must be removed by connecting incoming edges to the **goto** l statements directly to the destination designated by l.
- There are no error regions in the CFG. These needs to be detected and marked in the region graph.

DASH4j transforms the Soot CFG to a region graph. The transformation is performed by creating a new graph. In general, whenever there is a node in the original CFG there is a node in the region graph. The statement inside the node in the CFG is moved to the edges below. The individual transformations are illustrated in Figure 5.2.

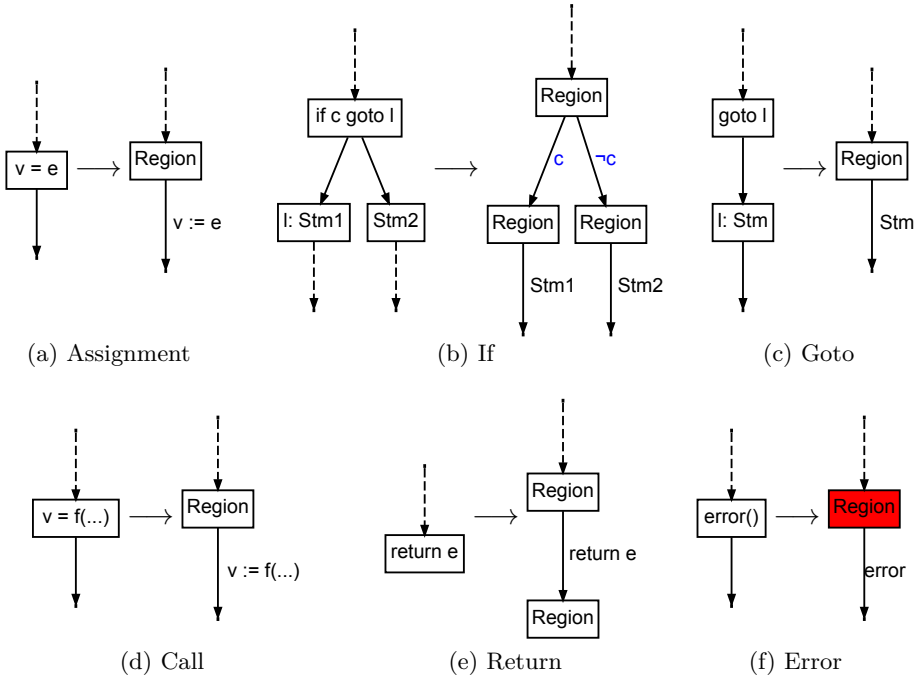
The general rule holds for assignments v := e, as seen in Figure 5.2a, and for assignments with method calls v := f(...) seen in Figure 5.2d. The statements are simply moved to the edges below.

The **if** c **goto** l statements needs more work. The node has 2 outgoing edges. One that corresponds to the **goto** l statement and another that falls through to the next statement. We cannot move the **if** c **goto** l down onto the edges, instead we place **assume** statements on the two edges. The edge corresponding to the **goto** l statement, has an **assume** c statement added. The edge that falls through to the next statement is given the statement **assume** ¬c. The translation is depicted in Figure 5.2b.

The plain **goto** l nodes are handled by removing them altogether. All incoming edges to the **goto** l are connected to the destination l. This works in a recursive manner, such that if a **goto** chain is present it is compacted to a single edge. This process is shown in Figure 5.2c.

The **return** nodes are converted as in Figure 5.2e. Jimple allows multiple exit points and the **return** nodes are the leaf nodes in the CFG. They do not have any outgoing edges. Therefore it is not possible to move the statement down to the edge below. For **return** nodes, we construct a new region below it, with a connecting edge containing the **return** statement.

Finally, the error() call needs to be handled. The transformation is nearly identical to assignments. The error() call is moved to the edge below and converted to an error statement. The region before the error statement is marked



(a) Assignment (b) If (c) Goto

(d) Call (e) Return (f) Error

Figure 5.2: Individual transformations of a control flow graph based on Jimple statements into a region graph with DASH statements.

as an error region. We cannot mark the region after the error statement as an error region. The reason is that this region may have incoming edges from other regions where the edges are not labeled with an error statement. Those regions are not going into an error condition, and therefore we cannot mark it as an error region. The transformation of error() regions is shown in Figure 5.2f.

The observant reader might have noticed that, as in Figure 5.1c, Jimple includes a section in the beginning where the parameters of a method are assigned to local variables. Those assignments are also found in the CFG that Soot produces. These assignments are ignored when generating the region graph. DASH4j stores the mapping of parameter indexes to variable names separately, analogues to how DASH keeps this information in P.

The region graph can now be constructed and DASH4j can start analyzing it. The next section describes how RunTest performs concrete execution.

5.4 Concrete execution – Implementing RunTest

RunTest is a part of the DASH algorithm that does not come with any pseudocode. This section describes how we have implemented it.

RunTest is given a test input t and is required to execute a concrete test where the state of the program needs to be recorded at every program point that corresponds to a region. We have chosen to implement RunTest by executing an instrumented version of the program. Before running the DASH algorithm, an instrumented version of the program is created and stored on disk. When

RunTest is called it loads the instrumented class in an isolated context, initializes it, and executes it concretely.

A special class called InstrumentationHelper is used throughout the instrumented program. The InstrumentationHelper is described in the next section.

5.4.1 The InstrumentationHelper

The InstrumentationHelper keeps track of:

- The currently reached region S_r, which is initially set to the initial region.
- The current state $\mathcal{C}$ of the program. This is much like the symbolic map $\mathcal{S}$, but $\mathcal{C}$ only contains concrete values such as $x \mapsto 4$. The concrete state is seeded with all variables that are external to P.
- A stack of method states, which essentially models a stack frame. A method state includes the region it has currently reached and the state $\mathcal{C}$ for the method. Whenever a method is called a new method state is pushed on top of the stack.

We instrument the original Jimple program by adding calls to the InstrumentationHelper, either before or after a Jimple statement. The state of the InstrumentationHelper is altered by calling these methods:

- updateVar("a", a): updates $\mathcal{C}$ such that the name "a" now has the concrete value a. This is equivalent to $\mathcal{C} := \mathcal{C}[\text{"}a\text{"} \mapsto a]$.
- advanceToNextRegion(): advances the region pointer S_r to the next region. There can be multiple outgoing edges of S_r and therefore a number of regions are candidates to be the next one. advanceToNextRegion() finds the next region by evaluating the region predicate of each region using the current state $\mathcal{C}$. When the edge going to a region contains an **assume** c statement, then the assumption is checked as well. Since all the equivalence classes for regions in DASH are disjoint, and because the **assume** c statements are disjoint as well, there can only be one transition that satisfies $\mathcal{C}$. Thus, when evaluating region predicates and possible **assume** statements, only one evaluates to **true** under $\mathcal{C}$.
- saveStateOnGraph(): places a copy $\mathcal{C}_{copy}$ of $\mathcal{C}$ on the region pointed to by S_r. It also connects $\mathcal{C}_{copy}$ with the parent state such that the parent-child relationship is constructed between the states.
- pushNewMethodState() and popNewMethodState() methods push and pop method states respectively. pushNewMethodState() saves S_r and $\mathcal{C}$ and adds it to the method stack. popNewMethodState() pops the topmost method element and loads S_r and $\mathcal{C}$ from it.

The next section describes how these methods are used to instrument a Jimple program, such that states are saved on the correct regions.

```
helper.callWithAssign();
v = cl_dI.f(a_0,...,a_n);
helper.callWithAssignEnd("v", v);
```

Figure 5.3: Compact instrumentation used for method calls

5.4.2 Instrumenting a class

The instrumentation process is depicted in Figure 5.4. It shows how each Jimple instruction is instrumented. As can be seen, many of the Jimple instructions needs to be instrumented with multiple method calls. As inserting these method calls are cumbersome in practice, our implementation actually only adds a single method call to each instrumentation, except for method calls which require two. We have "unpacked" these method calls to more clearly show what is being performed. Figure 5.3 shows our actual instrumentation for method calls.

To avoid class name clashes we change the name of the instrumented class by appending it with "_dI", which stands for DASH Instrumented. It is thereafter written to the disk where it can be loaded during RunTest and executed on the JVM.

The next sections present the individual instrumentations.

Instrumentation 1 – Add static helper field

I_1 shows that a test class is instrumented by adding a static field _dH, such that when the instrumented class is executed, it can get a reference to an injected InstrumentationHelper instance. The field name _dH stands for DASH Helper. The InstrumentationHelper is initialized before concrete execution begins. It has a reference to all graphs such that it can place states on them and it knows which method that is called initially by RunTest. Instead of instrumenting each class with a static field, we could have had a static field in another class that all instrumented classes could fetch. However, it is bad coding practice to have global state and it would also not be possible to execute two tests concurrently. By having RunTest explicitly set the InstrumentationHelper field on the instrumented class we avoid the global state, since classes are loaded in an isolated context.

Instrumentation 2 – Initialization of method call

The goal of I_2 is to fetch the InstrumentationHelper into a local variable, since method calls in Jimple must be on local variables. It also saves the initial state onto the region graph. The parameter values are loaded with calls to updateVar and saved to the initial region by saveStateOnGraph. It seems counterintuitive to load the parameters when RunTest was given them in the first place. However, when method calls are made from inside the instrumented class, the initial state of the called method must also be saved. These parameters are not given to RunTest and we have chosen to load them in the instrumented class, since then all parameters are loaded automatically.

#	Before instrumentation	After instrumentation

I_1	`public class TestClass`
	`{`
	`  ...`
	`}`
		`public class TestClass_dI`
		`{`
		`  InstrumentationHelper _dH;`
		`  ...`
		`}`

I_2	`public static void test(...)`
	`{`
	`  int a, b, ...;`
	`  a := parameter0;`
	`  b := parameter1;`
	`  ...`
	`}`
		`public static void test(...)`
		`{`
		`  int a, b, ...;`
		`  InstrumentationHelper helper;`
		`  a := @parameter0;`
		`  b := @parameter1;`
		`  ...`
		`  helper = _dH;`
		`  helper.updateVar("a", a);`
		`  helper.updateVar("b", b);`
		`  ...`
		`  helper.saveStateOnGraph();`
		`  ...`
		`}`

I_3	`  v = e;`	`  v = e;`
		`  helper.updateVar("v", v);`
		`  helper.advanceToNextRegion();`
		`  helper.saveStateOnGraph();`

I_4	`l1: if c goto l2;`	`l1: helper.advanceToNextRegion();`
		`  helper.saveStateOnGraph();`
		`  if c goto l2;`

I_5	`  goto l;`	`  goto l; //not instrumented`

I_6	`  error();`	`  error();`
		`  helper.advanceToNextRegion();`
		`  helper.saveStateOnGraph();`

I_7	`l: v = cl.f(`$a_0,\ldots,a_n$`);`	`l: helper.pushNewMethodState();`
		`  v = cl_dI.f(`$a_0,\ldots,a_n$`);`
		`  helper.popMethodState();`
		`  helper.updateVar("v", v);`
		`  helper.advanceToNextRegion();`
		`  helper.saveStateOnGraph();`

I_8	`l: return e;`	`l: helper.advanceToNextRegion();`
		`  helper.saveStateOnGraph();`
		`  return e;`

Figure 5.4: Instrumentation performed by DASH4j for each Jimple instruction executed inside a class.

A special requirement of Jimple is that parameters are loaded into local variables, using special identity statements, before any other code can be executed. Therefore we cannot interleave the loading of parameters with updating them using updateVar. This is why helper and the parameters are loaded after the identity statements. The initial state has the parent pointer set to **null**.

Instrumentation 3 – Handling assignments

I_3 handles assignment statements in the form v = e. The instrumentation is placed after the assignment operation has been performed. This allows us to simply read the value of v, without having to evaluate e. An assignment changes the state of the program and it could affect which region we were to transition to.

Therefore first updateVar is called to update the value stored in $\mathcal{C}$ and afterwards the next region is found using advanceToNextRegion. The state is then saved on the found region using saveStateOnGraph.

Instrumentation 4 – Conditional branching

The instrumentation I_4 handles conditional statements of the form **if** c **goto** l2. These statements are represented as two **assume** edges in region graph, as described in Section 5.3.

The instrumentation code is placed before the **if** statement. With assignments, all edges going of the region must contain the same assign statement. With conditionals, there are two possible statements, namely **assume** c and **assume** ¬c. Notice that since **assume** statements does not change the state in $\mathcal{C}$, we only need to find the next region and save a copy of the state on it.

The method advanceToNextRegion is called to find the next region. For all statements other than **if**, advanceToNextRegion only needs to check if the region predicate is satisfied by $\mathcal{C}$. For the **if** statements the **assume** statements on the edges must be checked as well. Thus, the predicate that must be satisfied for a transition to be possible is $\rho_{r+1} \wedge$ c where ρ_{r+1} is region predicate from the candidate region and c is from the **assume** statement leading there. Again, only a single transition is possible because the regions are disjoint under each distinct **assume** condition. When the next region is found, the state is saved with a call to saveStateOnGraph.

There is one important detail that must be handled when inserting instrumentation code *before* an instruction. If there is a label l1 at the instrumented instruction, it has to be moved such that the instrumentation code is executed as well. This is seen in I_4 where the label l1 has been moved from **if** c **goto** l2 to helper.advanceToNextRegion(). If not done correctly, the region graph and the concrete execution could go "out of sync".

Instrumentation 5 – Unconditional branching

DASH does not support unconditional **goto** statements and these were removed from the region graph constructed in Section 5.3, but their effect was modeled with edges in the graph. We therefore do not need to advance to the next region, nor save state on the region, with the result that no instrumentation is added for **goto** l statements, as can be seen in I_5.

Instrumentation 6 – Error call

The error() call, which works as the error statement in DASH4j, is instrumented in I_6. The instrumentation code is placed after the error call for convenience, since then labels do not need to be updated. The next region is found and the state is saved.

Here, instrumentation could have been added such that an error flag was set, and the implementation of IsErrorRegionReached could check this flag. This would constitute a small optimization.

Instrumentation 7 – Method invocations

I_7 shows the instrumentation for method invocations. The goal of this instrumentation is to:

- Transform the call such that an instrumented method is called, i.e. the call is made to the instrumented class and not the original.
- Save the current method state, i.e. S_r and $\mathcal{C}$, on the method stack.
- Set S_r and $\mathcal{C}$ to the correct values for the called method.
- When the called method returns, restore the method state S_r and $\mathcal{C}$ as it was before the method call.
- Update the assigned variable v in $\mathcal{C}$ with the return value of the method invocation.
- Save the state on the correct region after the call edge in the region graph.

To call the instrumented class, the call is modified from calling $\texttt{cl.f}(a_0, \ldots, a_n)$ to call $\texttt{cl\_dI.f}(a_0, \ldots, a_n)$ where c is the original class name. Notice that _dI has been appended to the class name, such that the instrumented version of the class is called.

Saving the current method state and creating the new one is accomplished by a call to pushNewMethodState. The old method state is saved on a stack and a new method state is created for the called method cl.f. The region graph for cl.f is found by looking up cl.f in $\mathcal{G}$, which was one of the arguments given to RunTest. The internal pointer S_r is set to the initial region and the state map $\mathcal{C}$ is initially empty. The parameters for the invocation are automatically loaded as described I_2. This concludes the setup for the method call.

When the method call returns the old method state is restored by calling popNewMethodState. The rest of the instrumentation is identical to the instrumentation made for assignments. The state $\mathcal{C}$ is updated such that it reflects the new value of v, the next region is found and the state is saved.

As was the case for the **if** instrumentation, we need to move any labels that would reach the method call up to the start of the instrumentation code. Thus the label l is moved such that pushNewMethodState is called if there is code that jumps to l.

Instrumentation 8 – Return statement

The last instruction to instrument is the **return** e statement. The goals are to record the state of the method and save it on the region after the **return** edge on the graph. When executing a **return** e statement, the current method returns and thus no statements after the **return** statement is executed. Therefore the instrumentation is placed before the **return** statement. Again, any incoming edges that hit the **return** statement must be moved to the start of instrumentation code.

5.4.3 Loading and running an instrumented program

We have shown how a program is instrumented. The instrumented program is saved to disk by converting the instrumented Jimple program back to bytecode using Soot. When RunTest is called, the instrumented program needs to be loaded and executed.

Loading is performed by using an URLClassLoader, as it can load a class from a specific location, i.e. the directory where DASH4j saves instrumented classes. DASH4j creates a new URLClassLoader for each RunTest invocation. Notice that classes loaded by one URLClassLoader shares their set of static variables. Loading the same classes with another URLClassLoader thus create a new set of static variables. This had made a difference if global static variables were supported by DASH4j, as discussed in Section 6.1.3, because then all static variables would have had to be found and reset.

When the test classes are loaded, the InstrumentationHelper is constructed and injected into the instrumented classes. DASH4j therefore instantiates the Class objects for the instrumented classes, such that the static field _dH can be set. A Class object is also used to invoke the tested method in the instrumented class.

When the test method returns, all the region graphs have been modified by the added instrumentation code and thus nothing else needs to be done, and RunTest can return to DashLoop. The next section describes how DASH4j integrates with Z3.

5.5 Integrating with Z3

DASH4j uses the Z3 theorem prover[3] to solve the constraints generated by ExecuteSymbolic, simplify constraints and evaluate expressions.

Since DASH4j is an implementation of DASH$_{call}$ in the context of Java, 32 bit signed integers must be supported. They are modeled by using Z3 BitVectors of length 32. A BitVector in Z3 is a fixed sized collection of bits. The bit vectors are "raw" in the sense that they contain no type information, thus they do not know if they are representing a signed or unsigned integer. Instead the operations performed on BitVectors are typed. For example, mkBVSDiv(bv1 , bv2) creates a signed division operation of bv1 and bv2. This is similar to the instruction set of an x86 CPU, where there are no type information associated with memory locations and the instructions come in signed and unsigned editions when needed. In Java, all 32 bit integers are signed and as such DASH4j uses the signed operations.

Initially, DASH4j stored the constraints for both regions and ExecuteSymbolic in its own internal format. The internal format was converted into Z3's format when IsSAT was called. Every expression was immutable such that expressions could be shared. DASH4j also had a custom made simplifier, which was a simple bottom-up rewriter. However, we ran into memory usage problems, which we believe were the result of:

[3] Z3 can be found at `http://z3.codeplex.com/`

- During symbolic execution each region predicate would be symbolically evaluated, which resulted in new expressions being created.
- Our custom written simplifier did not handle De Morgan's law in any smart way, and could easily end up duplicating many expressions. De Morgan's law is about pushing ¬'s down into subexpressions. These expressions would then be duplicated by our simplifier.
- Converting the constraints into Z3's format duplicates all the constraints and would not allow Z3 to perform any optimization during the construction of the constraints.

Using Z3's internal format and simplifier solved the memory problems. We believe Z3 has a significantly smarter internal storage than our own naive approach. Z3's ctx-solver-simplify was also significantly better than our own bottom up simplifier, when comparing the size of the expression afterwards. Our custom simplifier was also easily outperformed by Z3's simple bottom up rewriter called Simplify.

The dependency on Z3 also comes with a price. All interaction with Z3 uses a Z3 context object. It must be passed to all routines which interact with Z3, which clutters the implementation. What is worse, is that replacing Z3 with a different SAT solver would require a major rewrite of DASH4j. At this moment DASH4j is tightly coupled to Z3.

During development we found two bugs in Z3. One bug was that ctx-solver-simplify returned a non-equivalent constraint, i.e. it simplified incorrectly. Due to this bug, one of our assertions in DASH4j failed and we tracked it down to ctx-solver-simplify's wrong simplification. The bug was reported, confirmed and fixed[4]. The second bug was that when Z3 solved constraints with only two variables, it would sometimes return a result with three variables. The extra variable was named k!0. This bug was also reported and fixed[5]. The Z3 developers were very quick to respond and fix the reported bugs.

The last release version of Z3 was released 11th of November 2012[6]. This version does not have a Java API. Therefore DASH4j uses a version of Z3 compiled from their unstable branch. This branch contains scripts that generate a Java API that calls the Z3 implementation using Java Native Interfaces[7]. The first bug was only present in their unstable branch whereas the second was also found in their latest release.

5.6 Handling the Java Standard Library

In general, DASH4j supports calls into the Java Standard Library, but there is a series of requirements that must hold. When a method calls a static method inside another class, DASH4j instruments all the methods in the called class. DASH4j fails with exceptions if it finds Jimple code that it does not handle,

[4] https://z3.codeplex.com/workitem/94
[5] https://z3.codeplex.com/workitem/97
[6] http://z3.codeplex.com/releases
[7] http://leodemoura.github.io/blog/2012/12/10/z3-for-java.html

such as doubles, native method calls or instance methods. Thus, the class that is called into must only contain methods with integer arithmetic. If such a class exists in the Java Standard Library, then DASH4j is able to instrument and analyze it. However, we have not been able to find such classes in the Java Standard Library. Even the java.lang.Math class calls native methods. For example, sin(**double** a) calls StrictMath.sin(a), which is a native method.

Even if DASH handled doubles, integers, objects, constructors and exceptions, then DASH would have problems with the Java Standard Library. A significant amount of the Java Standard Library depends on native methods, like the file system classes, where mocking the semantics are required. The device drivers that Microsoft used DASH to analyze also calls into operating system functions which they had to model [B10]. Modeling the operating system calls was a significant challenge for them, and we do not believe it is any easier to model the Java Standard Library. We have not done any investigation on how we could model the Java Standard Library.

5.7 DASH4j implementation

We have had lots of problems implementing the DASH algorithm. We believe that the authors of DASH have first written an implementation of the algorithm, and then later wanted to write about it. In converting the algorithm from source code to article format, some details have gone missing. For example, the input constraint computed in ExtendFrontier did not include the path constraint in the article, instead, their implementation asserted it to Z3, which we mentioned in Section 4.7.2. In such cases it would have been beneficial to be able to inspect their source code, as the source code contains all the details. We hope our descriptions are clearer, but we also release our source code, such that others can browse through our implementation in case of missing details.

Our implementation is therefore open source and is located on a GitHub repository[8]. The GitHub repository includes directions on how to build and run the project. The source code is structured such that it resembles the pseudocode described herein as much as possible.

We have a total of 320 tests cases. 19 of them are ignored due to the implementation not being able to analyze them (recursive procedures, cannot find loop invariant, etc.). 255 of the tests are example programs that are analyzed by DASH4j. The rest are tests up against Z3, unit tests of program code and some tests to test nondeterministic branching that we have also implemented.

We have written a small tool that checks log output from the implementation and shows running times from different parts of the program. The output is given in Figure 5.5. The total runtime is 38 seconds for all our tests running on an laptop with a i7-3610QM Intel processor and 8GB of ram. Notice that the total runtime in Figure 5.5 is 35 seconds, which is due to not every part of DASH4j being traced. It can be seen that a significant amount of time is spent using ctx-solver-simplify inside RefineGraph. FindAbstractErrorPath is called 1391 times, which shows the number of iterations performed by DashLoop. Given

[8]Public repository: `https://github.com/foens/dash`

Procedure	Time ms	Avg	Min	Max	#calls
refineGraph	18698	28	0	2342	664
ctx-solver-simplify	18006	17	0	1173	1008
SATSolver	6675	6	4	323	1073
IsSAT	3911	3	1	320	1073
addPredicates	2185	2	1	4	1073
createSatisfiableResult	26	0	0	1	468
loadClass	3025	10	0	1525	293
forceResolve	1411	4	0	897	293
retrieveActiveBody	940	3	0	136	293
loadNecessaryClasses	399	1	0	398	293
runTest	2845	5	0	589	550
ctx-solver-simplify	1688	2	0	55	669
InstrumentationBuilder	1537	5	2	78	293
ExecuteSymbolic	754	0	0	95	1214
computeRefinePred	287	4	0	134	59
RefinePred	197	0	0	56	605
ToConcreteTraceWithAbstractFront...	61	0	0	13	1214
FindAbstractErrorPath	31	0	0	1	1391
isErrorRegionReached	0	0	0	0	550
extractTestInputForOuterProcedure	0	0	0	0	82
Total	35798	4	0	2342	8657

Figure 5.5: Table showing runtime information gathered from the log output of DASH4j.

that the total runtime is 38 seconds, then on average each of the DashLoop iterations take 27 milliseconds.

Chapter 6

Future Work

This chapter presents future work we would have liked to perform if time had allowed us to. First, the chapter presents various ways to extend the set of language features currently supported by DASH4j in Section 6.1. Section 6.2 describes various structural and performance improvements we would like to investigate. Finally, Section 6.3 describes tests we would like to perform against Microsoft's DASH implementation.

6.1 Extending the set of supported language features

The following extensions of language features require that ExecuteSymbolic can generate the correct constraints and that RunTest is able to generate the concrete input from the solved constraints. For example, if objects are supported, then somehow RunTest must be able to construct the correct objects from the test input t. It could therefore be valuable to investigate how other tools, like jCUTE [B11] that also needs to construct constraints and objects for their tests, handles this.

6.1.1 Pointer support with DASH$_{heap}$

We would like to support the C-like pointers, supported by DASH, in Java. The idea with DASH$_{heap}$ is to support simple objects with only one public integer field. Then the references to simple objects would function as if they were pointers to integers.

Implementing DASH$_{heap}$ would introduce the problem of aliasing, which would require us to implement WP$_\alpha$. The procedure WP$_\alpha$ differs from WP only in that it uses smart tricks to avoid an exponential blowup that could be caused by aliasing.

It would also require us to alter ExecuteSymbolic such that memory can be shared over method calls, when a pointer is passed to a sub procedure. We know that Z3 supports arrays with unbounded length, and we believe we can use this array to model the heap.

In essence, DASH$_{heap}$ corresponds to the full version of DASH described in the DASH article.

6.1.2 Supporting objects with multiple fields in DASH$_{struct}$

In the programming language C, structures are supported by using the struct keyword. These are simply a collection of fields. Supporting structs and pointers in the context of Java, would allow us to support objects with multiple public fields. These fields could point to other objects or contain integer values.

If we were able to support these C-like structures, then we could generate Java memory graphs representing linked lists, binary search trees or other unbounded memory structures. Creating constraints and concretely constructing the memory graphs requires us to extend DASH beyond what is described in the DASH article.

Where DASH$_{heap}$ could be implemented with pointers into an infinite large Z3 array, where each pointer points to exactly one integer, then we believe that struct could be implemented by pointing to the first field in an object, and where the remaining fields would be present sequentially after the first field. In this way, when obj.a is accessed, the symbolic value is found where obj is stored. Accessing obj.b would access the symbolic value found at address obj+1 in the Z3 array.

6.1.3 Supporting static fields with DASH$_{globals}$

DASH$_{globals}$ would add support for C-like global variables, which are variables that can be accessed and modified from any procedure.

In Java global variables would correspond to static fields, both private and public. The public static fields are accessible from any method and the private ones can only be accessed inside the declaring class, due to compile time checking. As described in Section 4.6, supporting the error statement in sub procedures can be achieved be instrumenting the program to use static fields or global variables.

Assuming that DASH$_{heap}$ has been implemented, then global variables could be modeled by calling the main procedure with all the global variables as additional arguments. When a procedure calls a sub procedure, all the variables are passed on. In this way the variables would function as global variables.

6.1.4 DASH$_{arrays}$

There are two types of arrays we could support in DASH$_{arrays}$, arrays that are located on the stack and arrays that are located on the heap. The later would build on DASH$_{heap}$.

Arrays could be modeled by using the Z3 array type, however these arrays are unbounded. Many languages, like Java, have a length property associated with the array. The length property would also be relevant to check out of bounds errors.

6.1.5 Supporting boolean, short and other types in DASH$_{types}$

Introduction of types other than 32 bit integers can be modeled in Z3 by simply creating bit vectors of the desired length. However conversions between bit

vectors of different lengths are required since shorts can be assigned to integers and integers to shorts (by casting).

At the bytecode level, Java already inserts casts when assigning a larger number to a smaller one. However, Java does not have expressions for converting a smaller number to a larger one. We would have to detect these and perform the equivalent operation in Z3.

6.1.6 General object support with DASH$_{object}$

The goal of DASH$_{object}$ would be to introduce proper objects with private variables and instance methods.

One way to represent objects is as they are represented on runtime – as procedures that take the instance as an argument. Thus we could represent the state of the object as a struct and pass it to called instance methods.

We believe the hard part is constructing only valid objects. Private fields can only be set by calling constructors or mutator methods. We are unsure how we can figure out which sequence of calls that is to be made to construct an object with some specific field values.

For example, a method may require an ArrayList with some specific state before triggering an error condition. The requirement is now for DASH4j to instantiate an ArrayList and call the correct sequence of methods with the correct arguments to get it into the correct state. We do not know how to solve such a problem, but we may be able to find answers in tools with similar requirements.

6.2 Improvements to code structure or performance

We have focused on correctness of the supported features when implementing DASH4j. We have found several areas where there is room for improvements or optimizations. These are discussed in the next sections.

6.2.1 Limit the execution steps performed after the frontier

In our current implementation, when RunTest is called, it keeps executing the program until the executed method returns. However, for correctness, we only require that at least the frontier edge is executed when RunTest is invoked. There are cases where one might end up in a long loop which is irrelevant to the safety property and which hurts performance.

An example is given in Figure 6.1. Initially, the default value of x, which is 0, is used. The result is that the execution enters an infinite loop. In this case it would be beneficial to limit the number of concrete execution steps that are executed after the frontier, to avoid that the analysis gets stuck running a test.

6.2.2 States linked across procedures and reference to region

During concrete execution by RunTest, the states are only linked inside a single procedure. Thus, when a procedure is invoked, the state before and after the procedure call are linked together, without the intermediate states that

```
void test(int x)
{
  while(x == 0) { /* empty */ }
  error;
}
```

Figure 6.1: The program will hit the error statement if x is different from zero.

were created in the called procedure. When performing symbolic execution, ExecuteSymbolic have to search through the initial region of a called procedure to find the state it should continue from. To save some time, the state before the procedure call could contain an extra reference to the state inside the sub procedure. Then ExecuteSymbolic would not need to search the initial region for the correct state.

Another optimization that could make a difference, was that each state could keep a reference to the region that they are contained. In ExecuteSymbolicSubProcedure, with pseudocode given in Algorithm 4.2, a concrete execution is symbolically executed by following the child relationship of the states in the concrete execution. However, the region that a state is placed in needs to be found, which is done by FindRegionWithState(Children(S_{prev}), s_{next}). It searches through all child regions of S_{prev} to find the region that contains s_{next}. If states kept a reference to the region they were contained in, then this operation would be a constant lookup operation instead of a search through all child regions of S_{prev} and checking each state against s_{next}.

6.2.3 Improve storage space of states

The current implementation of RunTest in DASH4j stores the state of all variables on each region. This is redundant since at most one variable will change and that only happens if the edge is an assignment.

6.2.4 Investigate simulation vs instrumentation for RunTest

DASH4j uses instrumentation to implement the RunTest functionality. We would like to investigate what impacts it would have to switch to a simulator instead. Much of the code for ExecuteSymbolic could be used for the simulator, as each expression is already supported. The new requirements would be to save state on regions, execute the code without following a specific trace and to use concrete values for the initial symbolic values. We believe we could stub these requirements out into interfaces such that ExecuteSymbolic did not know if it was following a trace or executing a program concretely.

There are some considerations to be made if such a switch is completed. For example, concrete execution performed by RunTest might get slower, since a simulator is used instead of using the production grade and optimized JVM. One benefit we would certainly loose, was that we now have two separate implementations for RunTest and ExecuteSymbolic. These separate implementations in a sense verifies that each of them works, since if they get "out of sync" then all sorts of assertions should fail. When we join them together, errors in

our implementation are harder to catch since symbolic execution and concrete execution would share a significant amount of code.

6.3 Testing against the DASH implementation

Given extra time we would have liked to test our program instances found while implementing DASH$_{int}$ and DASH$_{call}$ against the DASH implementation made by Microsoft. Their implementation is not open source, however, it is possible to add YOGI into their *Static Driver Verifier Research Platform* (SDVRP), and YOGI embeds the DASH implementation[1]. We would then create C-drivers that mimic the problems we have found with DASH and see how their implementation behaves. We would for example like to see how they handle the recursive procedures that DASH4j keeps running on infinitely.

We know that this is not a simple task to complete, since the SDVRP kit and YOGI have numerous dependencies and we lack experience in writing device drivers.

[1]Instructions for installing YOGI are given on `https://research.microsoft.com/en-us/projects/yogi/readme.txt`

Chapter 7

Conclusion

We have implemented DASH in Java both for analyzing single procedure integer programs and for analyzing procedural integer programs. We have named the implementation DASH4j. The programs DASH4j analyzes are written in a limited subset of Java where the Java Standard Library and features such as objects, exceptions and reflection are excluded. We have found several implementation problems and even errors in the description of DASH as presented by the DASH article [A2]. DASH4j is able to find subtle errors like integer overflow and can prove many programs as correct.

We have presented the DASH algorithm in depth, including many minor and some major details that are significant for the DASH implementation. Examples include optimization of predicates, missing input constraints when analyzing called procedures, and descriptions of how a specific loop optimization is implemented. Additionally, we have presented and analyzed problem instances that were encountered and shown that the solutions presented make large differences for the DASH algorithm. We believe that if others are to implement DASH, or build on it, it will be significantly easier to follow our description that includes solutions to problems not disclosed by the authors of DASH. We have also published our source code[1] such that others can see how DASH4j is implemented. It is published under a very permissive license[2].

We have found that the DASH algorithm can be oblivious to **if**s, **while** or **for** loops when they are irrelevant for the safety property being analyzed, because the concrete tests generated simply traverse them. However, we have found that when a result calculated inside a loop is relevant for a safety property, then the refinement process used in DASH can run infinitely. This is due to the fact that the weakest precondition technique is unsuitable for finding loop invariants. The authors of DASH have mentioned a loop optimization technique that helps, but does not directly solve the problem. Also, recursive procedures can be hard for DASH to analyze and sometimes DASH ends up recursively invoking itself until memory is exhausted.

The strategy to implement DASH to uncover details not described in the

[1] Public repository: `https://github.com/foens/dash`

[2] Creative Commons Attribution 4.0 International License: `http://creativecommons.org/licenses/by/4.0/`

article has shown to be a good strategy. We have found a number of problems and details not covered in the article where we have had to find solutions ourselves. For example, we found that the description of the input constraint in ExecuteSymbolic had the path constraint missing. We believe it would have been near impossible to find many of such problems if we had not implemented DASH. It has thus helped us in writing pseudocode that incorporates solutions to the problems.

Writing pseudocode that incorporated the uncovered details of DASH has also helped us in improving the implementation. For example, we had initially performed renaming while executing traces in ExecuteSymbolic, but when we wrote down the pseudocode, we found that renaming was not required. We therefore rewrote ExecuteSymbolic, which produced a cleaner implementation and better pseudocode. The same applies to how we performed renaming in ExtendFrontier, where we found multiple cases where renaming was not required.

This kind of self-reinforcing cycle, between the pseudocode and implementation, was not anticipated but has improved both pseudocode and implementation considerably.

We have found that our methodology of writing test cases have given us numerous examples that show that alterations to the DASH algorithm were needed. This is evident throughout our thesis, given that we nearly always present an example program that shows the problem being described. Having example programs has helped us immensely in discovering faults and solutions throughout the implementation process.

We have described ideas, which we believe might help in extending DASH such that it can handle objects as used in Java. We believe that support for exceptions, types and objects can be implemented. However, we do not believe that reflection can be supported in any significant portion, and we know that many static analysis tools disregard reflection altogether.

Finally, we have found that to use the DASH algorithm on any significant portion of Java requires a great deal of modeling of the Java Standard Library. Modeling of methods for which code is missing is a cumbersome task, one of which the DASH authors also called one of the most challenging issues encountered [B10]. For the DASH authors, it involved calls to the operating system, whereas for Java, it will involve the Java Standard Library and native method calls implemented by the Java Virtual Machine.

Primary Bibliography

[A1] Thomas Ball and Sriram K. Rajamani. "Automatically Validating Temporal Safety Properties of Interfaces." In: *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*. SPIN '01. Toronto, Ontario, Canada: Springer-Verlag New York, Inc., 2001, pp. 103–122.

[A2] Nels E Beckman, Aditya V Nori, Sriram K Rajamani, Robert J Simmons, Sai Deep Tetali, and Aditya V Thakur. "Proofs from tests." In: *Software Engineering, IEEE Transactions on* 36.4 (2010), pp. 495–508.

[A3] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "The Software Model Checker Blast: Applications to Software Engineering." In: *International Journal on Software Tools for Technology Transfer* 9.5 (Oct. 2007), pp. 505–525.

[A4] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In: *SIGPLAN Not.* 40.6 (June 2005), pp. 213–223.

[A5] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. "SYNERGY: a new algorithm for property checking." In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. SIGSOFT '06/FSE-14. Portland, Oregon, USA: ACM, 2006, pp. 117–127.

[A6] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C." In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 263–272.

Secondary Bibliography

[B7] Thomas Ball and Sriram K Rajamani. *SLIC: A specification language for interface checking (of C)*. Tech. rep. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[B8] Henrik B. Christensen. *Flexible, Reliable Software: Using Patterns and Agile Development*. 1st. Chapman & Hall/CRC, 2010.

[B9] J.C. Martin. "Introduction to Languages and the Theory of Computation." In: McGraw-Hill, 2003, pp. 419–421.

[B10] Aditya V. Nori and Sriram K. Rajamani. "An Empirical Study of Optimizations in YOGI." In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: ACM, 2010, pp. 355–364.

[B11] Koushik Sen and Gul Agha. "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools." In: *Computer Aided Verification*. Springer. 2006, pp. 419–423.