# Techniques for Combining Testing and Verification for Efficient Assertion Checking in Sequential Programs

**Prasanth Chatharasi**

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Bachelor of Technology

भारतीय प्रौद्योगिकी संस्थान हैदराबाद

Indian Institute of Technology Hyderabad

Department of Computer Science and Engineering

May 2011

# Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

(Signature)

(**Prasanth Chatharasi**)

(Roll No.)

# Approval Sheet

This Thesis entitled Techniques for Combining Testing and Verification for Efficient Assertion Checking in Sequential Programs by **Prasanth Chatharasi** is approved for the degree of Bachelor of Technology from IIT Hyderabad

(**Dr. Aditya Nori**) Adviser
Microsoft Research, Bangalore
Adjunct Faculty,
Dept. of Computer Science Eng
IITH

(**Dr. M.V. Panduranga Rao**) Co-Adviser
Dept. of Computer Science Eng
IITH

# Acknowledgements

I would like to express my deepest respect and sincere gratitude to my adviser **Dr. Aditya Nori**, Microsoft Research Bangalore, for his constant guidance and encouragement at all stages of my work. I am fortunate to have had technical discussions with him, from which I have benefited enormously. I sincerely thank him for all those valuable long hours he has spent with me.

I am extremely grateful to my co-adviser **Dr. M. V. Panduranga Rao**, for his constant encouragement and support all through my work. I am thankful to him for all the invaluable advice on both technical and nontechnical matters. The discussions with him have been a source of motivation and energy for me to persist in the field of research.

I would like to show my greatest appreciation to my well wisher **Dr. Kesav Nori**. I cannot say thank you enough for his tremendous support and help. I felt motivated and encouraged every time I met him.

I am extremely thankful to all faculty members of the Department of Computer Science and Engineering for sharing their views and giving valuable suggestions during the discussion of my work in department reviews.

I thank our director **Prof. U. B. Desai** for his friendly administrative support in getting all our requirements done as quickly as possible.

I thank all my classmates and research scholars for their friendly support who made the stay at this institute enjoyable, we shared joy and knowledge. I thank all my friends at IIT Hyderabad for the same. Especially, I would like to thank my room-mate Shanthanu Deshpande for his encouragement in ups and downs.

I deeply express my loving thanks to my mother and father for their encouragement, care and love. I express my heartfelt appreciation and gratitude to my dear sister Pavani and brother Sreenivas for their esteemed support.

Finally, I thank everyone who helped me directly or indirectly during my stay at IIT Hyderabad.

# Dedication

This work is dedicated to My grandfather, Ch. Rama Rao

# Abstract

The property checking problem is to check if a program satisfies a specified safety property. Two well known approaches for property checking are testing and verification. Testing finds out bugs in a program easily but it is difficult to prove the absence of bugs whereas verification is easy to find the proof for correctness rather than finding bugs in the program. Each approach has its strengths and weaknesses.

In this thesis, we summarize recent contributions in the area of testing and verification for the property checking problem. We also discuss some concepts prerequisite to techniques in testing and verification. SMASH, a first 3-valued compositional may-must analysis algorithm, combines testing and verification in a novel way for proving safety properties of a program [P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali, Compositional may-must program analysis: Unleashing the power of alternation, in *POPL 10: Principles of Programming Languages.* ACM Press, 2010]. SMASH algorithm is not guaranteed to terminate for recursive programs where data types are unbounded or if dynamic allocation is allowed. But, SMASH algorithm may not terminate on some class of recursive programs where data types are bounded and dynamic allocation is not allowed. We elaborate the problem with examples and conclude with an idea to solve the problem.

# Contents

# List of Algorithms

# List of Figures

# List of Symbols

| | |
|---|---|
| $\epsilon$ | Null |
| $\langle \Sigma, \sigma^I, \rightarrow \rangle$ | Concrete program P |
| $\langle \Sigma_{\simeq}, \sigma^I_{\simeq}, \rightarrow_{\simeq} \rangle$ | Abstract program P |
| $\phi$ | Constraints |
| $\rho$ | Predicate |
| $\rightarrow$ | Transitive Relation |
| $\rightarrow_{\simeq}$ | Transitive Relation on equivalence classes of $\simeq$ |
| $\Sigma$ | Set of states |
| $\sigma^I$ | Set of initial states |
| $\sigma^I \simeq$ | Set of equivalence classes that contain initial states |
| $\Sigma_{\simeq}$ | Set of equivalence classes of $\simeq$ in $\Sigma$ |
| $\xrightarrow{*}$ | reflexive-transitive closure of $\rightarrow$ |
| $\tau$ | Abstract trace |
| $\tau_{err}$ | Abstract error trace ( $S_0, S_1, ...$ ) |
| $\varphi$ | Set of bad states |
| $\varphi_{\simeq}$ | Set of equivalence classes that contain bad states |
| $S_k \rightarrow S_{k+1}$ | Frontier edge |
| A | Abstraction of a program P |
| F | Forest containing set of concrete paths of a program P |
| S | symbolic memory map obtained by performing symbolic execution |
| $\mathrm{S}_0$ | A state $\in \Sigma_{\simeq}$ |
| $\mathrm{s}_0$ | A state $\in \Sigma$ |
| $\mathrm{S}_x$ | Symbolic name for variable x |
| t | Concrete error path |

# Chapter 1

# Introduction

Today, checking the correctness of a software has become the primary goal of software industries for quality assurance. Usually, checking the correctness of software nearly accounts for 50% of the cost of software development [1]. Correctness is one of the properties associated with a program. There are several types in properties of a program. *Safety properties* assert that program does not exhibit bad behaviour. An example for the safety property is that the program never goes into an error state. *Liveness properties* show that program does something useful. Currently, we limit our discussion to proving safety properties of a program. *Property checking* or *assertion checking* is checking if the property is satisfied by all possible executions.

Two broad approaches to property checking are *program testing* and *program verification*. Program testing tries to find inputs to demonstrate violations of the property. According to Edsger W. Dijkstra [2], "Program testing is a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of it's absence". After executing number of tests, we are not sure about the existence of another input that violates the property. Program verification tries to construct a formal proof that shows that program always satisfies the property in all of its possible executions. Some times, program verification leads to non termination in computing the over approximation of the possible program executions. Generally, user intervention is needed to terminate the computation and it leads to less adoption in industries.

Program analysis is the process of analysing the behaviour of programs. Program analysis tools automatically analyze the program and proves certain properties. There has been a lot of attention received in last few decades for these tools to check the properties [3]. These tools can be categorized using their approach towards proving properties. These categories are program testing and program verification.

Tools based on program testing search for inputs that violates property by executing the program in one form or the other. There are several techniques in program testing to find such inputs. At one extreme, program is executed on chosen concrete inputs to check the violation of properties. These concrete inputs are chosen by random generators, test case generators [4] or manually. All these methods have their advantages and limitations. At another extreme, program is executed symbolically [5] to characterize all possible paths in the program. Then, each and every path is examined using automated constraint solver [6] for violation of property. Since constraint solver has limitations, this method doesn't guarantee the complete coverage. As a result, many intermediate solutions are pursued to check the properties such as directed testing [7]. In this method, the program is executed on a random concrete input and collects the constraints along the path using symbolic execution [5]. Then, this information is subsequently used to find another test that drives the program execution into another path. This process is repeated until it finds an input that violates the property. This idea has been implemented in a tool called Dynamic Automatic Random Testing (DART) [7]. Later, this has been enhanced to support pointers and implemented in a tool called Concolic Unit Test Engine (CUTE) [8]. Then, this has been extended by encoding test results and implemented in a tool called Sub Modular Automated Random Testing (SMART) [9]. But, it is very difficult using these tools to say that there are no inputs that violates the property. It can be done if complete coverage is ensured by a test case generator and there are no generation of false witnesses.

Tools based on program verification search for a proof to show that program always satisfies the property. There are several techniques in program verification. At one extreme, deductive verification [10] is used to construct formal proof. According to Floyd [11], "Inductive Invariant technique is the right way of using induction for proving properties of imperative programs". In deductive verification, initially an inductive invariant is to be proved on the program. It means that the invariant holds in the initial state of the program and if it holds in some state then it continues to hold in every successor state as well. Thus, if the inductive invariant implies the desired property, then the proof is complete. The hardest part of proof is finding the inductive invariant. Recently, some methods are proposed to iteratively refine the invariants to get the suitable inductive invariant for proof. At other extreme, model checking [12] is used to automatically verify properties of programs. In this classical model checking, reachable state space is constructed for finite state programs and is checked for truth of property. In order to support infinite state programs, an abstract program containing finite number of executions is derived from the infinite state program. If there is a bug in the original program, then there must be a corresponding bug in the abstract program. But if the model checker finds a bug in the abstract program, it will not necessarily indicate a bug in the original program. If such a situation occurs, then the abstract program is refined to eliminate this spurious counter example. This method has been implemented in a tool called SLAM [13] that uses predicate abstraction and automatic partition refinement. Later, BLAST [14] has been employed with counter example driven automatic abstraction refinement to construct an abstract model that is model-checked for safety properties. But it is very difficult using these tools to report some input that violate the property. It may be done as only a by-product of expensive, failed search proof.

Recently, a lot of importance has been given to combining testing and verification for proving properties of program. In Synergy [15], directed testing and abstraction refinement based verification algorithms are combined in a novel way to prove safety properties of program. Later, it has been enhanced in DASH [16] by supporting the pointers and functions. Then, it has been extended further in SMASH [17] by encoding the analysis results for better performance.

These tools all combine techniques from static program analysis (symbolic execution), dynamic analysis (testing and run time instrumentation), model checking (systematic state-space exploration), and automated constraint solving. Static program analysis helps in constructing formal proofs for program verification and dynamic program analysis helps in finding a test input for programs testing.

## 1.1 Contribution and Scope of work

The contribution of this work is as follows.

1. An exposition of fundamental concepts like symbolic execution and test generation tools like Dynamic Automatic Random Testing (DART)[7], Concolic Unit Test Engine (CUTE) [8] and Sub Modular Automated Random Testing (SMART) [9] that are prerequisite for research in program analysis.

2. Survey of recent advances in combining testing and verification for property checking problem and explanation of tools like Synergy [15], DASH [16], and SMASH [17] that combines testing and verification in a novel way.

3. SMASH [17] is not guaranteed to terminate for recursive programs where data types are unbounded or if dynamic allocation is allowed. But, SMASH may not terminate on some class of recursive programs where data types are bounded and dynamic allocation is not allowed. We illustrate the problem using examples and conclude with an idea to solve the problem.

4. Illustrations of some of above tools using new examples.

The present work remains in the scope of proving properties of sequential programs. The goal of this work is to put the existing series of algorithms in a sequence for a better understanding of current research in program analysis. So, we have described the algorithms in a more readable form and ignored details of theoretical analysis of algorithms and practical implementations. We consider only side effect free expressions in the program for property or assertion checking. Please see the cited works for details.

## 1.2 Organization of thesis

The contents of thesis are organized as follows.

In **Chapter 2**, we explain the fundamentals like assertions, symbolic execution and automated constraint solver or theorem prover.

In **Chapter 3**, we explain test case generation in detail. We begin with random test case generation and followed by static and dynamic test case generations along with their advantages and limitations. Then, we explain the tools DART, CUTE and SMART that uses the dynamic test case generation. We conclude with advantages and limitations of above tools.

In **Chapter 4**, we describe Synergy algorithm, that combines testing and verification in a novel way. We begin with motivation towards the idea of Synergy algorithm. Then, we give an introduction to ideas in Synergy algorithm. Then, we describe the algorithm in a readable manner. Then, we give the illustration of Synergy algorithm over three kinds of examples, which results in either proof or test input or non terminating. Finally, we conclude with the limitation of the algorithm.

In **Chapter 5**, we describe the DASH algorithm, an extension of Synergy algorithm. We begin with limitations of Synergy algorithms and introduce the ideas towards uplifting limitations. Then, we describe the algorithm using the same notations followed in earlier chapter. Then, we give illustrations of DASH algorithm over examples. Then, we conclude with limitations of DASH algorithm.

In **Chapter 6**, we begin with motivation towards SMASH algorithm. Then, we describe the notion of summaries that are encoded analysis results. Then, we describe $\neg$ *May* and *Must* summaries for answering queries over a function. Then, we give some illustrations towards the alternation of those summaries for better performance.

In **Chapter 7**, we describe the limitations of SMASH algorithm through examples. SMASH would not be guaranteed to terminate over recursive programs which have infinite data type domains or dynamic memory allocations [17]. In this chapter, we first provide an example of such program. Secondly, it was not known whether SMASH can be terminated on all programs having finite data type domains and not having dynamic memory allocations. We then show an example where SMASH fails on a program with above two properties. Finally, we conclude this chapter by showing a way to handle such cases with an example.

In **Chapter 8**, we describe an idea to solve the problem. We conclude the chapter by illustrating the solution using the idea.

# Chapter 2

# Background

In this chapter, we explain concept of assertions, notion of symbolic execution and idea of automated constraint solver or theorem prover. It forms the background in understanding the algorithms for property checking.

## 2.1 Assertions

Assertion is a predicate that asserts about values which the relevant variables will take at that particular point of time in execution.

```
x := 5;
{x > 0} -- Assertion
x := x + 1
{x > 1} -- Assertion
```

In the above example, $x > 0$ and $x > 1$ are assertions. *Precondition* of a program is an assertion that specifies the relationship between variables before execution of the program. *Post condition* of a program is an assertion that describes the intent of that program. Let Q, P and R be the symbolic notations for the program, precondition, and postcondition respectively. We use notation $P\{Q\}R$ to say that if the assertion P is true before initiation of program Q, then the assertion R will be true on its completion. Now, we explain the axioms and rules of inferences associated with *programming* statements.

**Axiom of an assignment statement:** Let assignment statement be $x = f$, where x is an identifier for a simple variable and f is an expression possibly containing x. The axiomatic schema for the above statement is $P \ \{x = f\} \ P_1$, where P is obtained from a given $P_1$ by substituting f for all occurrences of x. A new P will be obtained for every assertion substituted for $P_1$. The above schema describes infinite set of axioms which share common pattern.

**Rules of consequence:** The rule of inference takes the form " $\vdash$X and $\vdash$Y then $\vdash$Z ". It means, if assertions of form X and Y have been proved as theorems, then Z is also proved as theorem. Some of the rules of consequence are listed below.

$$\text{If } P\{Q\}R \text{ and } R \Rightarrow S \text{ then } P\{Q\}S$$
$$\text{If } P\{Q\}R \text{ and } S \Rightarrow P \text{ then } S\{Q\}R$$

**Rule of composition:** Let Q1 and Q2 be two sequences of statements that are executed one after another. The inference rule associated with composition is as follows.

$$\text{If } P\{Q1\}R \text{ and } R\{Q2\}S \text{ then } P\{Q1; Q2\}S.$$

**Rule of iteration:** The general notation of looping statements is " While B do S ". Suppose P is an assertion, which is always true on completion of S, provided that it is true on initiation also. Formally,

$$\text{If } P \wedge B\{S\}P \text{ then } \{\text{while B do S}\}\neg B \wedge P.$$

Now, we discuss the method of deductive reasoning to prove certain properties of a program. An important property of program is partial correctness (proving correctness without termination)

**Deductive reasoning:** It involves application of rules of inference to sets of valid axioms for proving properties of program. Let Q, P, and R be a program, precondition, and post condition respectively. Then, deducing P{Q}R leads to satisfying necessary condition for partial correctness of program Q. Consider the following program to illustrate the deductive reasoning.

```
int foo ( int x ) {
    y = x + 2 ;
    return y;
}
```

Assume that, the input to the above program is always greater than or equal to 0. We wish to prove the correctness of above program, the value of y should be greater than or equal to 2. The precondition for above program is "x $\geq$ 0" and postcondition is "y $\geq$ 2". To prove the partial correctness of above program, we need to deduce "x $\geq$ 0 {foo} y $\geq$ 2"

$$\begin{array}{rl}
\text{Using Axiom of assignment} & \vdash \quad \{x+2 \geq 2\}\ y = x+2\ \{y \geq 2\} \\
 & \vdash \quad x \geq 0 \implies x+2 \geq 2 \\
\text{Using rule of consequence} & \vdash \quad \{x \geq 0\}\ \{y = x+2;\}\ \{y \geq 2\} \\
\text{Using rule of composition} & \vdash \quad \{x \geq 0\}\ \{y = x+2;\ \text{return y; }\}\ \{y \geq 2\}
\end{array}$$

Finally, " x $\geq$ 0 {foo} y $\geq$ 2 " is deduced. Hence, the partial correctness of above program is proved using deductive reasoning. However, proving properties of programs, containing million lines of code, is difficult using the deductive verification approach.

## 2.2 Symbolic execution

In this section, we discuss about symbolic execution. Then, we illustrate it using an example. Finally, we conclude this section with limitations.

**Description:** Symbolic execution is equivalent to executing the program "symbolically" by giving symbols as inputs to program. It symbolically explores the possible behaviours of a program and characterizes them using path conditions. Path condition is the accumulator of properties from branch conditional statements along the path. If program execution follows a path with given inputs, then those inputs should satisfy the path condition associated with that path. Now, every path is associated with the path condition and final state values of program variables. The results of symbolic execution may be equivalent to a results of same program over large number of test cases.

**Verification:** Symbolic execution can be used to verify certain safety properties of program. In this approach, each path condition and its final state values will be used to check against safety property. If some path condition and its final state values violates the property, then path condition will be given to constraint solver to get feasible inputs which satisfies the constraints. If all the path conditions and its final state values satisfy the property, then it will acts as proof for correctness of that safety property.

**Test generation:** Symbolic execution can be used to generate test cases that may covers all the possible program behaviours of program. It may give complete coverage depending upon the complexity of constraints in branching statements. After executing the symbolic execution, all the path condition are given to constraint solver to generate the minimal number of test cases that covers the most possible behaviours of the program.

**Illustration:** Consider the following program to illustrate the symbolic execution.

```
int function (int x) {
   if ( x == 10 )   abort ; // error
   else             print "hi";
   return 0;
}
```

Let $S_x$ be symbolic name for variable x. There are two possible execution paths in above program, one following *if* branch and another following *else* branch. After symbolic execution, the corresponding path conditions for above paths are $S_x = 10$ and $S_x$ != 10 . Now these constraints are given to constraint solver to generate test cases. The possible test inputs from constraint solver are 10 and 15 (random).

**Limitations** : It cannot ensure complete coverage if the program contains statements involving constraints outside the scope of reasoning of the constraint solver. This limitation is illustrated using the following example [9].

```
void obscure (int x, int y ) {
    if (x == hash(y) ) error ;
}
```

Assume, function *hash* cannot be reasoned about symbolically. Let $S_x$ and $S_y$ be symbolic variables for x and y. After performing symbolic execution, the path conditions for two paths are ( $S_x = hash(S_y)$ ) and ( $S_x \mathrel{!=} hash(S_y)$ ). These constraints are given to constraint solver for feasible test cases. But, *hash* function is out of scope of constraint solver. At this instant, it cannot generate two values for inputs that are guaranteed to satisfy (or violate) the constraint ( $S_x = hash(S_y)$ ).

## 2.3 Theorem prover

Theorem provers or constraint solvers are used to validate the expression and finds the assignment to variables in that expression to make it true. Automated theorem proving is proving mathematical theorem by a computer program. Since validation of expressions under different logic's is undecidable, there exists no perfect tool which can validate any kind of expression. But according to Godel's completeness theorem, validation of expressions in first order predicate calculus is done in polynomial time. We use mostly theorem provers for validating the expressions over first order predicate logic. Automated theorem prover some times called as automated constraint solver since it can automatically solve constraints and gives feasible inputs. There exist many automated constraint solver like Z3 [18] to solve the expressions in first order logic. They gives feasible input if the expression is valid otherwise returns saying infeasible expression.

In this section, we consider a basic interpretation oriented theorem prover over integers [6]. It is a basic theorem prover for validating expressions over integers. This theorem prover was built around powerful system for manipulating and simplifying integer expressions which is called as Formula Simplifying System. This system takes input expressions which are in the form of disjunctive normal form. So, given logical expression is converted into disjunctive normal form and will be fed into Formula system. It will solve the expression and returns the feasible values to variables in the expression.

# Chapter 3

# Test input generation

In this chapter, we begin with definition of test input generation or test case generation. Then, we describe the random test case generation and followed by static and dynamic test case generations along with their advantages and limitations. Then, we explain the tools that employs the dynamic test case generation namely DART, CUTE and SMART. Finally, we conclude this chapter by listing the advantages and limitations of techniques employed in these tools.

**Definition:** Consider a sequential, deterministic program P consisting of a set of program statements S (assignments, tests, loops, etc.). Given an input I, program P computes value v = P(I) (also called output) where v is the output of execution of P on input I. Then the problem of test input generation is defined as follows: Given a statement s $\in$ S of a program P, compute an input I such that the execution of program P on input I reaches statement s.

## 3.1 Random test case generation

In this approach, test inputs are chosen randomly over the domain of potential inputs for a given program. Consider the following program to illustrate the random test case generation.

```
int function (int x) {
   if ( x == 10 )
     error;
   return 0;
}
```

In the above program, assume x is a 32 bit integer. With the above approach, the value of x is randomly chosen over domain of 32 bit integers. Then, chance of following the *then* branch of conditional statement " if (x == 10)" is $\frac{1}{2^{32}}$ and is significantly low. As a result, this approach has very less chance to exercise *then* branch of the conditional statement and reach the *error*. If the conditional statement is " if (x != 10)", then the chance of exercising the *then* branch is very high.

**Limitations:** Since the inputs are chosen randomly over input domain, many set of inputs may follow the same path and leading to the same observable behaviour. The chance of choosing the random input that results in buggy behaviour is very less in this approach.

## 3.2 Static test case generation

In this section, we discuss about static test case generation. It generates the test cases by statically analyzing the given program. It uses most of the techniques from symbolic execution. Symbolic execution has been discussed in chapter 2. Symbolic execution attempts to compute the inputs to drive the program along specific execution paths, without ever executing program. The basic idea of symbolic execution is to explore the tree of all the computations that the program exhibits with all possible value assignments to input parameters. Symbolic execution constructs the path condition for every control path in the program. The path condition characterizes the set of inputs for which program execute along that path. Path condition is a conjunction of constraints on inputs.

**Generating test cases:** In order to generate test cases, program will be executed symbolically and explores possible program behaviours. Symbolic execution generates the path condition for each path. If the path condition is feasible to solve, then constraint solver outputs the feasible test input. Now, each path condition is given to constraint solver to generate inputs that satisfies the path condition. Assuming that the constraint solver used to check the satisfiability of all path constraints is sound and complete, this use of static analysis amounts to a kind of symbolic testing.

**Illustration:** Consider the following program to illustrate the static test case generation.

```
int main ( int x, int y ){
   if ( x > 10 ) {
      if ( y < 20 ) {
         S1; // Statement
      } else {
         error; // Statement
      }
   }
   S2; // Statement
}
```

Assume $S_x$ and $S_y$ are symbolic notations for input variables x and y. In the above program, there are three feasible paths. After performing symbolic execution, the path conditions for paths in above program are ( $S_x > 10$ & $S_y < 20$ ), ( $S_x > 10$ & $S_y \geq 20$) and ( $S_x \leq 10$). These path conditions are given to constraint solver for solving each of them. The feasible test inputs for above program would be (15, 10), (20, 25) and (5, 10). But the program execution on input (20, 25) leads to error statement.

**Limitations** : It cannot ensure complete coverage if the program contains statements involving constraints outside the scope of reasoning of the constraint solver.

## 3.3 Dynamic test case generation

In this section, we discuss about dynamic test case generation, another approach towards generating test cases for a given program. Dynamic test case generation consists of analyzing the program by executing several times. It uses symbolic execution and run time information to compute appropriate inputs to drive along specific execution paths or branches of a program.

**Steps:** Dynamic test generation consists of

1. Executing the program P, starting with some given or random inputs
2. Gathering symbolic constraints on inputs at conditional statements along the execution and
3. Using a constraint solver to infer variants of the previous inputs to steer the programs next execution toward an alternative program branch.

This process is repeated until a specific program statement ( *error* ) is reached. whenever symbolic execution doesnt know how to generate a constraint for a program statement depending on some inputs, that constraint will be simplified using inputs concrete (execution) values at that point of execution.

**Illustration:** Consider the following program to illustrate dynamic test case generation.

```
void obscure (int x, int y ) {
    if (x == hash(y) )
        error; //
}
```

Assume, function *hash* cannot be reasoned about symbolically. Let $S_x$ and $S_y$ be symbolic variables for x and y. After performing symbolic execution, the path conditions for two paths are ( $S_x = hash(S_y)$ ) and ( $S_x$ != $hash(S_y)$ ). These constraints are given to constraint solver for feasible test cases. But, *hash* function is out of scope of constraint solver. At this instant, it cannot generate two values for inputs that are guaranteed to satisfy (or violate) the constraint ( $S_x = hash(S_y)$ ) using symbolic execution. At this time, run time information is used to address the above problem. The following steps illustrate the typical process of generating test inputs using this approach.

1. **First execution**: Let random inputs of x and y be 33 and 42 respectively. Assume, hash(42) is equal to 567. Then, program execution on these inputs doesn't follow *then* branch in above program. The constraints captured during symbolic execution along this path are ( $S_x$ != $hash(S_y)$ ). This constraint is negated to find another input which drive program along different path. So, the constraints to be solved are ( $S_x = hash(S_y)$ ). Since this constraint is out of scope of constraint solver, this constraint is simplified using run time information of concrete values of inputs at that point of execution. In this case, $hash(S_y)$ will be replaced by 567 and the simplified constraint is ( $S_x$ != 567). This is solved by constraint solver and generates the input value for x and y as 42 and 567 respectively.
2. **Second execution**: Now the inputs to above program are 42 and 567 respectively. The program execution execution will follow the *then* branch on these inputs and *error* statement will be reached.

**Advantages:** This approach can easily drive the above program execution through all its feasible program paths where as static test generation is unable to generate test inputs to above example. This approach can alleviate imprecision in symbolic execution by using concrete values and randomization. In practise, imprecision in symbolic execution typically arises in many places, and dynamic test generation can recover from that imprecision. Dynamic test generation can be viewed as extending static test generation with additional run time information. It can use the same symbolic execution engine and use concrete values to simplify constraints outside the scope of the constraint solver.

**Limitations:** Practically, this approach typically cant explore all the feasible paths of large programs in a reasonable amount of time. However, it usually does achieve much better coverage than pure random testing and symbolic testing.

## 3.4 Dynamic Automated Random Testing

In this section, we discuss a tool called Directed Automated Random Testing, or DART. Then, we continue with description of algorithm in DART for test case generation and we give some illustration using examples. Finally, we conclude with advantages and limitations of this tool.

**Introduction** It blends dynamic test generation with model checking techniques with the goal of systematically executing all feasible program paths of a program while detecting some specific statement (*error*). DART addresses does the following things as desribed in [7].

1. “Automated extraction of the interface of a program with its external environment using static source-code parsing
2. Automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in and
3. Dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths.”

**Steps:** DART uses technique of dynamic test generation as follows

1. In the initial iteration, program is executed with random inputs and later by directed inputs.
2. Performs the symbolic execution along the path governed by program execution and gathers symbolic constraints on inputs.
3. Uses a constraint solver to infer variants of the previous inputs to drive the programs next execution toward an alternative program branch.

This process is repeated until a specific program statement ( *error* ) is reached. whenever symbolic execution doesnt know how to generate a constraint for a program statement depending on some inputs, that constraint will be simplified using inputs concrete (execution) values at that point of execution.

### 3.4.1 Algorithm

The following algorithm gives an overview of exact algorithm in DART [7].

**Algorithm 1** Dynamic Automated Random Testing

```
Require: Input program P
 1: Instrument the program P
 2: repeat
 3:    Initialize the stack for constraints
 4:    Do random initialization of inputs to P
 5:    while directed search is not completed do
 6:       Let s be first line in DART instrumented program P
 7:       while s $\notin$ { abort, halt } do
 8:          if s is an assignment statement ( m $\rightarrow$ e ) then
 9:             Update $S_m$ by symbolic value of e
10:             Update s by next statement in P
11:          end if
12:          if s is conditional statement ( if e goto l) then
13:             Let *sym*, *conc* be symbolic and concrete value of e
14:             if value of *conc* is true, then push the *sym* onto stack and update s by l
15:             else push negation of *sym* onto stack and update s by next statement in P
16:          end if
17:       end while
18:       if s is halt then
19:          solve the path constraints, get next directed inputs
20:       end if
21:       if s is abort then
22:          raise an exception and break
23:       end if
24:    end while
25: until All constraints are in scope of constraint solver
```

**Illustration:** DARTs integration of random testing and directed search using symbolic reasoning is best explained with the following example.

```
int f(int x) {
   return 2 * x;
}
int h(int x, int y) {
  if (x != y)
    if (f(x) == x + 10)
       error;
}
```

In the above code snippet, the function *h* is defective because it may lead to an error statement for some input. The following iterations illustrate the working of algorithm in DART.

1. DART instruments the program. It mean, extra code will be added to program to perform symbolic execution.

2. Assume, $S_x$ and $S_y$ are the symbolic notations for the variables x and y.

3. **Iteration 1:**

    - Let the randomly chose value of x and y be 43 and 21. The program execution on those inputs follows the *then* branch of first conditional statement and *else* branch of second conditional statement.
    - After performing symbolic execution along path guided by concrete program execution, path condition corresponding to the path is ($S_x$ != $S_y$) and ( $2 \times S_x$ != $S_x + 10$ ).
    - Last constraint in above path condition is negated to drive the newer inputs towards another path.
    - After solving them by constraint solver, the feasible inputs are 10 and 21.

4. **Iteration 2:**:

    - The directed input values of x and y from previous iteration are 10 and 21. The program execution on those inputs follows the *then* branch of first conditional statement and *then* branch of second conditional statement.
    - The path condition, after performing symbolic execution, is ($S_x$ != $S_y$) and ( $2 \times S_x = S_x + 10$ ).
    - The program execution along this path reaches the error statement and DART gets terminated.

5. Finally, DART generates an input (10,21) for which program execution reaches error statement.

### 3.4.2 Advantages

DART has several advantages in comparison to static test generation. In this sub section, we explain the advantages of DART in comparison to static test case generation using symbolic execution by considering the following example [7].

**Example 1:**

```
struct foo { int i; char c; }
bar (struct foo *a) {
    if (a->c == 0) {
       *((char *)a + sizeof(int)) = 1;
       if (a->c != 0)  abort();
    }
}
```

DART treats the input pointer to *bar* function as a symbolic variable. In beginning, DART randomly initializes it to NULL or to single allocated cell of appropriate type. Static analysis tools including alias analysis may not guarantee that a→ c has been overwritten. But, DART finds an execution leading to *abort* easily by simply generating an input satisfying the linear constraint (a→c = 0).

**Example 2:** DART alleviates the imprecision aroused due to symbolic execution using run time information. Consider the following example [7] where constraint solver cannot handle the constraints.

```
foobar(int x, int y){
   if (x*x*x > 0)  abort();
}
```

Suppose that constraint solver cannot handle nonlinear arithmetic constraints. If the constraints are out of scope of constraint solver, then constrains will be simplified using run time information. In the above example, DART generate an input which leads to abort statement where as symbolic execution will be struck in solving the constraint.

**Limitations:**

1. Since concrete values are used to simplify constraints whenever symbolic execution is not possible, DART effectiveness critically depends on the symbolic reasoning capability available.

2. Systematically exploring all feasible program paths, in a program having multiple function calls, is typically expensive for large programs. This problem is considered as path explosion problem.

## 3.5 Extensions of DART

In this section, we discuss extensions of DART to alleviate the limitations of DART. We begin with description of each problem and an algorithm to solve the problem.

**Problem 1 - Constraints Limitations:** Replacing of constraints, that are out of scope of constraints solver, by run time information of corresponding variable has advantages and are explained in earlier sections. But there are some problems associated with this replacement. It cannot assure 100% path coverage. It may result in missing some bugs.

The typical cases where constraint solver cannot solve constraints are non linear arithmetic constraints, pointer constraints, array/memory references, bit vector operations. DART proposed a simple strategy to generate random memory graphs where each pointer is either NULL or points to a new memory cell whose nodes are recursively initialized. This strategy suffers from several deficiencies [8] and are explained as follows.

1. “The random generation of list of nodes may not terminate .

2. The random generation produces only trees for list of nodes. For example, it cannot produce a cyclic list.

3. The directed generation does not keep track of any constraints on pointers”.

### 3.5.1 Concolic Unit Testing

In this extension, we discuss the solution for constraints limitation. This solution is just an approximate solution towards the problem. This solution has been implemented in a tool called Concolic Unit Testing (CUTE) [8].

**Idea:** The key idea is to represent inputs using a logical input map that represents all inputs, including (finite) memory graphs, as a collection of scalar symbolic variables and then to build constraints on these inputs by symbolically executing the code under test. We will illustrate the idea using the following example.

**Illustration:**

```
typedef struct cell {
   int v;  struct cell *next;
} cell;
int f(int v) { return 2*v + 1; }
int testme(cell *p, int x) {
   if (x > 0)
     if (p != NULL)
         if (f(x) == p->v)
            if (p->next == p) error;
   return 0;
}
```

The function *testme* is defective because it may lead to an error statement for some value of its input vector, which consists of the input parameters p pointer to cell and x an integer variable.

1. CUTE instruments the program for getting symbolic constraints.
2. Let $P_0$ and $X_0$ be symbolic variables for p and x.
3. **Iteration 1**
   - Let the randomly chosen value of $P_0$ and $X_0$ be NULL and 10. With these inputs, program execution follows the *then* branch of first *if* condition and *else* branch of second *if* condition.
   - After performing the symbolic execution over the path followed by program execution, path condition is ( $X_0 > 0$) and ($P_0 =$ NULL )
   - Last constraint in above path condition is negated to drive the newer inputs towards another path. After solving them by constraint solver, the feasible inputs are 10 and a single allocated cell.

4. **Iteration 2**:

   - The logical input map is a sequence of memory graphs and scalar symbolic variables. For the above example, the logical input map is $\langle$ $P_0$, $P_0 \rightarrow v$, $P_0 \rightarrow next$, $X_0\rangle$, where $P_0$, $P_0 \rightarrow v$, $P_0 \rightarrow next$, $X_0$ are symbolic variables.
   - Directed input from previous iteration to logical input map is
     $\langle$ pointer to cell, 30, NULL, 10$\rangle$
   - Path condition after executing program on the above inputs is
     ( $X_0 > 0$) & ($P_0$ != NULL ) & ( f(x) != $P_0 \rightarrow v$ )
   - Last constraint in above path condition is negated to drive the newer inputs towards another path. After solving them by constraint solver, the directed input is
     $\langle$ pointer to cell, 21 , NULL , 10 $\rangle$.

5. **Iteration 3**:

   - The logical input map is $\langle$ $P_0$, $P_0 \rightarrow v$, $P_0 \rightarrow next$, $X_0\rangle$, where $P_0$, $P_0 \rightarrow v$, $P_0 \rightarrow next$, $X_0$ are symbolic variables.
   - Directed input from previous iteration to logical input map is
     $\langle$ pointer to cell, 21 , NULL , 10 $\rangle$.
   - Path condition after executing program on the above inputs is
     ( $X_0 > 0$) & ($P_0$ != NULL ) & ( f(x) == $P_0 \rightarrow v$ ) and ($P_0 \rightarrow next$ != $P_0$).
   - Last constraint in above path condition is negated to drive the newer inputs towards another path. After solving them by constraint solver, the feasible inputs are $\langle$ pointer to cell, 21 , pointer to same cell , 10 $\rangle$.

6. **Iteration 4**:

   - The logical input map is $\langle$ $P_0$, $P_0 \rightarrow v$, $P_0 \rightarrow next$, $X_0\rangle$, where $P_0$, $P_0 \rightarrow v$, $P_0 \rightarrow next$, $X_0$ are symbolic variables.
   - Directed input from previous iteration to logical input map is
     $\langle$ pointer to cell, 21 , pointer to same cell , 10 $\rangle$.
   - Program execution on above inputs reaches the error statement.

7. Finally, CUTE generates an input $\langle$ pointer to cell, 21 , pointer to same cell , 10 $\rangle$ which makes execution to reach error statement.

**Problem 2 - Path explosion:** Systematically exploring all feasible program path, in a program having multiple function calls, is typically expensive for large programs. This problem is known as path explosion problem.

The idea to solve the above problem is to perform dynamic test case generation com-positionally (notion of summaries) using techniques for inter procedural static analysis that have been used to make static analysis scalable to very large programs.

### 3.5.2 Systematic Modular Automated Random Testing

In this sub section, we discuss another extension for DART to solve the problem of path explosion. Systematic Modular Automated Random Testing (SMART) [9] extends DART by by encoding test results as function summaries and reusing those summaries for later re-use. These function summaries are expressed using input preconditions and output post conditions. For a fixed reasoning capability of constraint solver, compositional approach to dynamic test generation (SMART) is both sound and complete compared to monolithic dynamic test generation (DART). In other words, SMART can perform dynamic test generation compositionally without any reduction in program path coverage.

**Function summary:** A function summary $\phi_f$ for a function f is defined as a formula of propositional logic whose propositions are constraints that can be solvable by constraint solver . It is defined as a disjunction of formulas $\phi_w$ of the form ( $\text{pre}_w \implies \text{post}_w$ ) where $\text{pre}_w$ is a conjunction of constraints on the inputs of f while $\text{post}_w$ is a conjunction of constraints on the outputs of f. $\phi_w$ can be computed from path constraint corresponding to the execution path. We use the following example to illustrate the working of SMART [9].

**Illustration:**

```
int foo(int x, int y) {
   if ( y != 0)
      if (x > 10)
  error;
   return 0;
}
int main(int a) {
    int b;
    if(a > 20)
        b = 0 ;
        return foo(a-1, b);
    else
        b = 1;
        return foo(a+1, b);
    return 0;
 }
```

The following iterations illustrate the working of algorithm in SMART.

- **Iteration 1**

  1. Let the randomly chosen value of a be 30. The program execution on this input follows *then* branch of *if* condition.
  2. Since there are no summaries associated with function *foo*, SMART starts analyzing the function *foo*.

3. The summary will be calculated using directed search over same function with the given context.
4. The summary calculated is ( $y \neq 0$ & $x > 10$ ) $\Longrightarrow$ false and ( $y == 0$) $\Longrightarrow 0$ and ( $y \neq 0$ & $x \leq 10$ ) $\Longrightarrow 0$
5. The path condition is ($a > 20$) and input to constraint solver is $\neg$ ( $a > 20$).
6. After solving them by constraint solver, the feasible inputs is 15.

- **Iteration 2**
  1. The directed input value of x from previous iteration is 15. The program execution on this input follows *else* branch of *if* condition.
  2. Since there exist summary for function *foo* which is suitable to current situation, SMART will use the summary.
  3. Using the summary, SMART comes to conclusion that foo (15, 1) reaches abort statement as follows.
  4. Given $x = 15$ and $y = 1$, ( $y \neq 0$ & $x > 10$ ) $\Longrightarrow$ false will be satisfied.
- Finally, SMART generates an input $x = 15$ for which program execution reaches error statement.

# Chapter 4

# Synergy

This chapter describes an algorithm, called Synergy [15], a new verification algorithm for checking safety properties of sequential programs. Typically, assertions will be used to represent safety properties of programs. Checking safety properties of a program is equivalent to assertion checking problem. Synergy is aimed at either finding a test case that violates the assertion or finding a finite-indexed abstraction that proves that assertion is not violated by any test case. Generally, execution based tools like DART, SMART aim at finding bugs and proof based tools like SLAM, BLAST find proofs. But execution tools may also find proofs only at the expensive search of all possible test cases. Similarly proof based tools may also find bugs only at the expensive refinements. But Synergy searches for bugs and proof simultaneously and uses the information obtained in searching one to others.

## 4.1 Motivation and Introduction

As discussed in last chapter, DART starts with a random inputs and move towards an input whose execution path leads to error. The reason for starting with random inputs is, the path from input to error state is not known. But the path can be known from abstraction of control flow of program. If there exists an error path from abstraction, that error path may not be feasible since it is an abstraction. So, there exist spurious paths in abstraction. But spurious paths can be removed by partitioning the states along the execution path of counter examples. This refinements will be done until the error path is feasible or there is no error path from abstraction. If abstraction doesn't have a path to error region, then concrete program will not have the test case that leads to error region. But if abstraction does have path to error region, then concrete program may or may not have test case that leads to error region. Synergy combines this idea of guiding the refinements by counter test cases and guiding the path to error region using error trace from abstraction. Thus Synergy combines testing and verification for checking the assertion in a sequential programs.

Synergy is a verification algorithm which searches simultaneously for bugs and proof. It tries to put the information obtained in one search to the best possible use in the other search. The search for bugs is guided by the proof under construction, and the search for proof is guided by the program executions that have already been performed. Synergy, thus, is a combination of under approximate and over approximate reasoning. Program executions on concrete inputs produces a precise under approximation of the reachability tree of the program, and partition refinement produces a successively more precise over approximation. As mentioned in motivation, Synergy guides testing towards errors using abstract error traces and guides refinement using test information. Refinement is helpful when there are number of branching statements and Testing is helpful when there are loop statements. Typically, program contains both kind of statements. So, simultaneous testing and verification performs better than independent use of both execution-based and proof-based tools.

## 4.2 Synergy Algorithm

The Synergy algorithm is explained in the current section. The back ground needed for the algorithm is explained in the following.

### 4.2.1 Definitions

- **Abstract error trace**: The path from the initial node to error node in abstraction is called as abstract error trace and is represented by $\tau$.
- **Concrete error trace**: The path from initial node to error node in concrete program where assertion gets violated is called as concrete error trace.
- **Frontier**: The unvisited node by concrete execution in an abstract error trace is known as frontier.
- **Forest**: Synergy maintains Forest F for under approximate reasoning. This forest F contains the collection of test cases that Synergy performs on program P. Each path in forest F corresponds to an concrete execution trace of program P. When ever a path that leading to an error location is added, then a test case has been found which can acts as a witness to violation of assertion.
- **Abstraction**: Synergy maintains relational abstraction A for over approximate reasoning. This abstraction contains set of states and edges connected to states as per program. But there might not exists an equivalent concrete trace for a given abstract trace. Each state is an equivalence class of concrete program states. If there exists a concrete edge between any concrete state in an abstract state to any concrete state in another abstract state, then there will be an abstract edge/transition between those abstract states. When ever there is no abstract error trace in abstraction A, then proof has been found which acts as a witness to say that assertion cannot be violated any test input.

### 4.2.2 Algorithm

The overview of Synergy algorithm is explained as following. It is the compact description of algorithm given in paper.

The algorithm takes 1) Program P and 2) Assertion $\psi$ as inputs and produces the following output.

1. It may output "fail" together with a test case that generates concrete error trace to $\psi$.
2. It may output "proof" together with a finite-indexed partition proving that program P will not reach $\psi$
3. It may not terminate

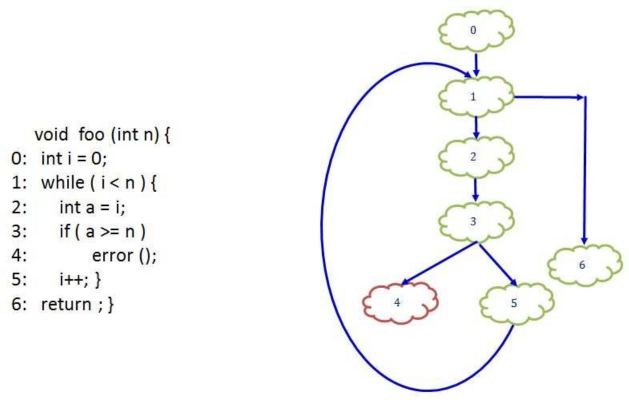
**Algorithm 2** Synergy

```
Require: Input program P = $\langle \Sigma, \sigma^I, \rightarrow \rangle$, Assertion $\psi$
 1: Assumes: $\sigma^I \cap \psi$ = empty
 2: F = empty
 3: $\Sigma_{\simeq} = \{\sigma^I, \psi, \Sigma \setminus \sigma^I \cup \psi\}$
 4: loop
 5:   if there is any concrete path 't' in forest F to reach error then
 6:     return ("Fail", t) where t is a concrete error trace to $\psi$
 7:   end if
 8:   Construct an abstract program A = $\langle \Sigma_{\simeq}, \sigma^I_{\simeq}, \rightarrow_{\simeq} \rangle$ using P and $\simeq$
 9:   Get the abstract trace $\tau$ from abstract program A to assertion $\psi$
10:   if abstract trace $\tau$ is empty then
11:     return ("Pass", $\Sigma_{\simeq}$) where $\Sigma_{\simeq}$ is a finite indexed partition
12:   end if
13:   Get the ordered abstract error trace $\tau_{err}$ from abstract trace $\tau$
14:   Obtain the position of frontier k from $\tau_{err}$ and Forest f
15:   Get the constraints $\phi$ by executing symbolically up to frontier
16:   Generate the suitable test input to extend the frontier by solving the constraints $\phi$
17:   if the frontier is extend-able from S$_k$ to atleast S$_{k+1}$ along $\tau_{err}$ then
18:     Execute the program P on that test input
19:     Add the resultant concrete trace to forest F
20:   else
21:     Refine the abstract state S$_{k-1}$ using the preimage operator as follows.
22:     $\rho$ = Pre(S$_k$) = $\{s \in \Sigma | \exists s^1 \in S_k . s \rightarrow s^1\}$
23:     $\Sigma_{\simeq} = (\Sigma_{\simeq} \setminus S_{k-1}) \cup \{S_{k-1} \cap \rho), S_{k-1} \setminus \rho)\}$
24:   end if
25: end loop
26: return ("Fail", an error trace, from forest F, that satisfies $\sigma^I$ and reaches $\psi$ ;) or
    ( "Pass", finite indexed abstraction $\Sigma_{\simeq}$, proves that $\psi$ cannot be reached by any input that
    satisfies $\sigma^I$ )
```

## 4.3 Examples

In this section, algorithm is illustrated with examples. The algorithm over these examples resulting in either finding a test case to violate an assertion or finding a proof to always satisfy an assertion or non terminating.

### 4.3.1 Synergy results in finding Proof



```
   void foo (int n) {
0:  int i = 0;
1:  while ( i < n ) {
2:     int a = i;
3:     if ( a >= n )
4:          error ();
5:     i++; }
6:  return ; }
```

Figure 4.1: Example on which Synergy results in proof

Fig 4.1 contain the program and its abstraction. Synergy is able to prove the property by discovering three predicates ( $a \geq N$ ), ($i \geq N$) and (false) in three refinement steps. Synergy starts with the initial partition $\{\{ pc = i \} \mid 0 \leq i \leq 6 \}$, and the initial abstract program is isomorphic to the concrete programs control flow graph.

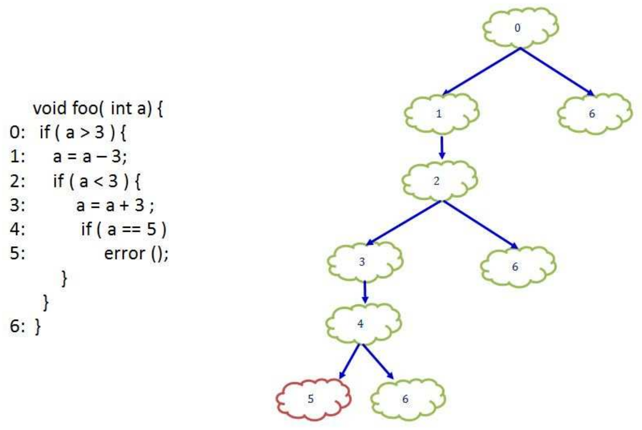
1. **Iteration 1**: Since forest F doesn't contain a path that leads to error state, there is no witness to error. There is no proof since abstraction A contains an abstract error trace { 0, 1, 2, 3, 4 } and frontier is 0. Then Synergy extend the frontier from 0 to 1 by giving an input to program as n = 2. It results in adding concrete path $\{ 0, (1, 2, 3, 5)^{2}, 1, 6 \}$ to Forest F.

2. **Iteration 2**: There is no witness and proof. Abstract A contains an abstract error trace { 0, 1, 2, 3, 5, 1, 2, 3, 4 } and the frontier is 4. But Synergy couldn't extend the frontier from 3 to 4 because of infeasible constraints from 0 to frontier. So, Synergy uses this information to refine the state 3 using the pre image operator applying on edge from 3 to 4 ans state 4. Now Synergy splits the state 3 into $3\wedge P$ and $3\wedge\neg P$ where $P = ( a \geq N)$.

3. **Iteration 3**: There is no witness and proof. Abstraction A contains an abstract error trace $\{ 0, 1, 2, 3\wedge P, 4 \}$ and frontier is $3\wedge P$. But synergy couldn't extend the frontier from 2 to $3\wedge P$. So, Synergy uses this un ability of extension and refines the state 2 into $2\wedge Q$ and $2\wedge\neg Q$ where $Q = ( i \geq N)$. Now, Synergy uses theorem prover after refinement for maintaining abstraction. It concludes that $2\wedge Q$ is not reachable from 1. This is because, in order to enter loop, $i < N$ should be satified. But Q is $i \geq N$. So, Synergy removes the edge from 1 to $2\wedge Q$.

4. **Iteration 4**: There is no witness. But there is no abstract error path from state 0 to error state. This abstraction acts as proof for saying that the assertion will not be violated by any test case.

### 4.3.2 Synergy results in finding test case



```
   void foo( int a) {
0:  if ( a > 3 ) {
1:    a = a – 3;
2:    if ( a < 3 ) {
3:        a = a + 3 ;
4:         if ( a == 5 )
5:             error ();
         }
      }
6: }
```

Figure 4.2: Example on which Synergy results in test case

Fig 4.2 contains the program and its abstraction. Synergy is able to find a test case by moving through at least one step towards error region. Synergy starts with the initial partition {{ pc = i } | $0 \leq i \leq 6$ }, and the initial abstract program is isomorphic to the concrete programs control flow graph.

1. **Iteration 1**: Since forest F doesn't contain a path that leads to error state, there is no witness to error. There is no proof since Abstraction A contains an abstract error trace { 0, 1, 2, 3, 4, 5} and frontier is 0. Then Synergy extend the path from 0 to 1 by giving an input to program as a = 10. It results in adding concrete path { 0, 1, 2, 6 } to Forest F.

2. **Iteration 2**: There is no witness from forest F. There is no proof since abstraction A contains an abstract error trace { 0, 1, 2, 3, 4, 5} and Frontier is 3. Then Synergy extend the path from 2 to 3 by collecting and solving the constraints from 0 to 3. The constraints are ( $a > 3$ and $a < 6$). Then constraint solver solves above constraints and generate an input a = 4. Synergy will run the program by giving a = 4 as input. It results in adding concrete path { 0, 1, 2, 3, 4, 6 } to Forest F.

3. **Iteration 3**: There is no witness from forest F. There is no proof since abstraction A contains an abstract error trace { 0, 1, 2, 3, 4, 5} and Frontier is 5. Then Synergy extend the path from 4 to 5 by collecting and solving the constraints from 0 to 5. The constraints are ( $a > 3$ and $a < 6$ and $a = 5$). Then constraint solver solves above constraints and generate an input a = 5. Synergy will run the program by giving a = 5 as input. It results in adding concrete path { 0, 1, 2, 3, 4, 5 } to Forest F.

4. **Iteration 4**: Now, there is a witness from Forest F which contains the path from 0 to error state. The test case corresponding to the path is 5. Synergy could find an input a = 5 which violates the assertion.

### 4.3.3 Synergy results in non terminating

```
void foo()
{
        int x, y;
1:      x = 0;
2:      y = 0;
3:      while (y >= 0) {
4:              y = y + x;
        }
5:      assert(false);
}
```

Figure 4.3: Example on which Synergy results in non termination

Fig 4.3 contains the program and its abstraction. Synergy tries to find either a test case or proof. But verification is undecidable problem. Unfortunately, Synergy doesn't terminate on this example. Synergy starts with the initial partition $\{\{ \text{pc} = \text{i} \} \mid 0 \leq \text{i} \leq 6 \}$, and the initial abstract program is isomorphic to the concrete programs control flow graph.

1. **Iteration 1**: There is no witness from forest F. There is no proof since abstraction A contains an abstract error trace { 1, 2, 3, 5} and frontier is 1. Then Synergy extend the path from 1 to 2. Unfortunately this program is a non terminating. So synergy will stop this program execution after some time. It results in adding concrete path { 1, 2, (3, 4) $^k$} where k is some constant to Forest F.

2. **Iteration 2**: There is no witness from forest F. There is no proof since abstraction A contains an abstract error trace { 1, 2, (3, 4)$^k$, 5} and frontier is 4 . Then Synergy extend the path from 3 to 4 by collecting and solving the constraints from 1 to 3. The constraints are ( $a > 3$ and $a < 6$). Then constraint solver solves above constraints and generate an input $a = 4$. Synergy will run the program by giving $a = 4$ as input. It results in adding concrete path { 0, 1, 2, 3, 4, 6 } to Forest F.

3. **Iteration 3**: Similar to iteration 2.

## 4.4 Conclusion

Synergy is a new algorithm that has combined the directed testing and abstraction refinement based techniques in a novel manner. This has alleviated the problems due to limitations of independent use of testing and verification. This has made a new approach that made testing and verification as dependable for the problem of assertion checking in sequential programs. But this approach has some limitations. It has made use of constraint solver for both directed testing and maintaining abstraction at each iteration of algorithm. Since use of constraint solver is expensive in terms of time, this algorithm can be improved to run in less time as compared to original algorithm. Along with expensive use of constraint solver, synergy doesn't handle arrays, pointers and inter procedure calls.

# Chapter 5

# DASH

DASH [16] is a verification algorithm, an extension of Synergy algorithm. This algorithm is used to check whether program P satisfies an assertion $\psi$. It alleviated the problem of using constraint solver for maintaining abstraction. It uses only test generation operations and it refines, and maintains abstraction only as a consequence of failed test case generation. Unlike Synergy, DASH handles pointers and inter procedural calls. This chapter describes the DASH algorithm and it's working on examples.

## 5.1 Motivation and Introduction

As discussed in last chapter, Synergy combines testing and verification in a way that information obtained in search for proof is put in tests and vice-versa. The Synergy algorithm works by iteratively refining the tests and the abstraction, using the abstraction to guide generation of new tests and using the tests to guide where to refine the abstraction. But Synergy uses constraint solver for maintaining abstraction along with test case generations that is an expensive work. DASH handles inter procedural call by recursively invocations. Generally, pointers can be handled using pointer alias analysis. Global pointer alias analysis, path sensitive alias analysis and path independent alias analysis are normally used for pointer alias analysis. Refinement can be done using strongest post condition or weakest precondition. But Synergy uses weakest precondition. Unfortunately, using weakest precondition for refinements in case of pointers results in explosion of predicates. On other hand, if strongest post condition is used for refinement, then it may require many iterations to find the strongest post condition or some times it results in non termination in calculation of strongest post condition. So, DASH introduces a new operator called WP$\alpha$ which considers only possible pointer aliases along concrete execution paths. This operator is stronger that weakest precondition but weaker than strongest post condition.

## 5.2 DASH Algorithm

The overview of DASH algorithm is explained as following. It is the compact description of algorithm given in paper. This section also gives sketch of DASH algorithm.

### 5.2.1 Description of algorithm

The algorithm takes 1) Program P and 2) Assertion $\psi$ as inputs and produces the following output.

1. It may output "fail" together with a test case that generates concrete error trace to $\psi$.
2. It may output "proof" together with a finite-indexed partition proving that program P will not reach $\psi$
3. It may not terminate

### 5.2.2 Sketch of DASH

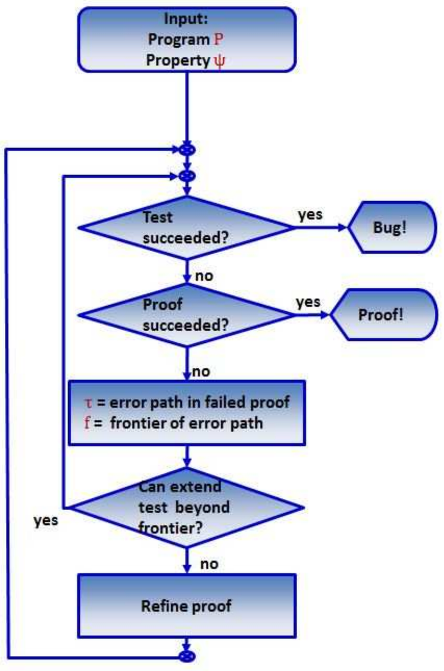


Figure 5.1: Sketch of DASH

**Algorithm 3** DASH

```
Require: Input program P = $\langle \Sigma, \sigma^I, \rightarrow \rangle$, Assertion $\psi$
 1: Assumes: $\sigma^I \cap \psi$ = empty
 2: F = empty
 3: $\Sigma_{\simeq} = \{\sigma^I, \psi, \Sigma \setminus \sigma^I \cup \psi\}$
 4: loop
 5:   if there is any concrete path 't' in forest F to reach error then
 6:     return ("Fail", t) where t is a concrete error trace to $\psi$
 7:   end if
 8:   Construct an abstract program A = $\langle \Sigma_{\simeq}, \sigma^I_{\simeq}, \rightarrow_{\simeq} \rangle$ using P and $\simeq$
 9:   Get the abstract trace $\tau$ from abstract program A to assertion $\psi$
10:   if abstract trace $\tau$ is empty then
11:     return ("Pass", $\Sigma_{\simeq}$) where $\Sigma_{\simeq}$ is a finite indexed partition
12:   end if
13:   Get the ordered abstract error trace $\tau_{err}$ from abstract trace $\tau$
14:   Obtain the position of frontier k from $\tau_{err}$ and Forest f
15:   Get the constraints $\phi$ by executing symbolically up to frontier
16:   if frontier is a function call to procedure Q then
17:     Get the post condition $\psi$' = S$_k$ [e/x], where e is the returned expression in Q and x is the
        variable in the caller P that stores the return value
18:     Invoke DASH with $\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle$, Assertion $\neg\psi'$
19:   else
20:     Generate the suitable test input to extend the frontier by solving the constraints $\phi$
21:   end if
22:   if the frontier is extendable from S$_k$ to at least S$_{k+1}$ along $\tau_{err}$ then
23:     Execute the program P on that test input
24:     Add the resultant concrete trace to forest F
25:   else
26:     Refine the abstract state S$_{k-1}$ using the preimage operator as follows.
27:
          $$WP\downarrow(op,\psi) = \alpha \wedge WP(op,\psi)$$
          $$\rho = WP_{\alpha}(op,\psi) = \neg(\alpha \vee WP\downarrow(op,\psi))$$
28:     States in abstraction are modified as follows.
          $$\Sigma_{\simeq} = (\Sigma_{\simeq} \setminus S_{k-1}) \cup \{S_{k-1} \cap \rho), S_{k-1} \setminus \rho)\}$$
29:     Edges in abstraction are modified as follows.
          $$\begin{aligned} \rightarrow_{\simeq} &= (\rightarrow_{\simeq} \setminus\{(S, S_{k-1}) | S \in \text{Parents}(S_{k-1})\}) \\ &\quad \setminus\{(S_{k-1}, S) | S \in (\text{Children}(S_{k-1}))\} \\ \rightarrow_{\simeq} &= \rightarrow_{\simeq} \cup\{(S, S_{k-1} \wedge \rho) | S \in \text{Parents}(S_{k-1})\} \cup \\ &\quad \{(S, S_{k-1} \wedge \neg\rho) | S \in \text{Parents}(S_{k-1})\} \cup \\ &\quad \{(S_{k-1} \wedge \rho, S) | S \in (\text{Children}(S_{k-1}))\} \cup \\ &\quad \{(S_{k-1} \wedge \neg\rho, S) | S \in (\text{Children}(S_{k-1})) \setminus \{S_k\}\} \end{aligned}$$
30:   end if
31: end loop
32: return ("Fail", an error trace, from forest F, that satisfies $\sigma^I$ and reaches $\psi$ ;) or
    ( "Pass", finite indexed abstraction $\Sigma_{\simeq}$, proves that $\psi$ cannot be reached by any input that
    satisfies $\sigma^I$ )
```

## 5.3 Handling Pointers

This section describes way of handling the pointer by DASH algorithm. This has been illustrated through example.

### 5.3.1 Description

Generally programs containing pointers is difficult to reason and analyze because computing the aliases precisely is very difficult and computationally expensive. Typically whole program may alias analysis is used to improve the precision of weakest precondition. The outcome of this analysis largely influences program analysis tools. Typically whole may program alias analysis is over approximate because it captures the aliases that may possibly happen in the program. There may exist a element in aliases in set that doesn't alias in any of the program execution.

Since whole program may alias analysis is over approximate, the weakest precondition computed will contain exponential disjunctions of predicates.

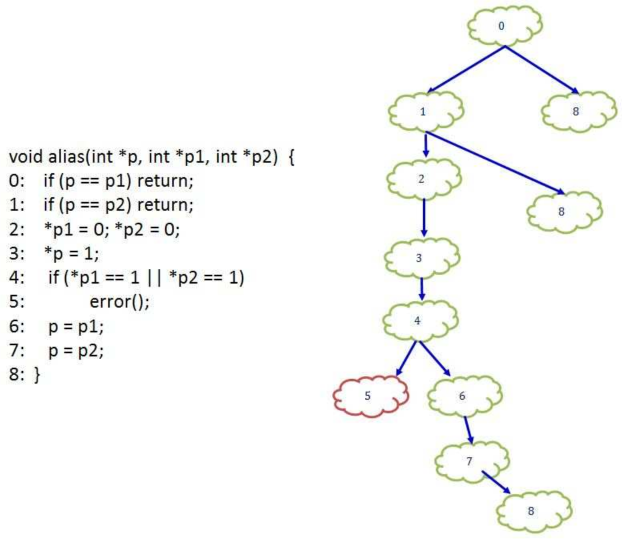
$$\begin{aligned}
WP(i = j,^* a +^* b < 10) \quad = \quad & \{a \neq \&i \wedge b \neq \&i \wedge^* a +^* b < 10\} \vee \\
& \{a = \&i \wedge b \neq \&i \wedge j +^* b < 10\} \vee \\
& \{a \neq \&i \wedge b = \&i \wedge^* a + j < 10\} \vee \\
& \{a = \&i \wedge b = \&i \wedge 2 \times j < 10\}
\end{aligned}$$

Consider the above example. The variable i may be aliases with two variables (a, b). If there are n possible aliases obtained from whole program may alias analysis, then there will be $2^n$ disjunctions of predicates in weakest precondition which is explosion and is difficult to handle. DASH algorithm takes an alternate approach by considering only the aliasing conditions that can happen along the current abstract trace, and computes the weakest precondition specialized to that aliasing condition. Let $\alpha$ be the set of aliasing conditions that can happen along current abstract trace.

$$WP \downarrow (op, \psi) = \alpha \wedge WP(op, \psi)$$
$$WP_\alpha(op, \psi) = \neg(\alpha \vee WP \downarrow (op, \psi))$$

Consider the same above example where the aliasing condition that has happened along current abstract trace is ( b $\neq$ & i). Then WP$_\alpha$ operator applying on edge ( i = j) and post condition as ( $^*$ a + $^*$ b < 10) is equal to (a = & i $\wedge$ b $\neq \&i$ $\wedge$ j + $^*$ b < 10) and remaining 7 disjunctions comes to $\neg$WP$_\alpha$ which reduces the burden of having explosion of disjunctions of predicates.

## 5.3.2 Example



```
void alias(int *p, int *p1, int *p2) {
0:   if (p == p1) return;
1:   if (p == p2) return;
2:   *p1 = 0; *p2 = 0;
3:   *p = 1;
4:    if (*p1 == 1 || *p2 == 1)
5:          error();
6:    p = p1;
7:    p = p2;
8: }
```

Figure 5.2: Example on which DASH results in proof

Fig 5.2 contains the program and its abstraction. DASH is able to prove the property by discovering the predicates ( *p1 == 1 ∨ *p2 == 1), ( (*p == p1 & *p == p2) ∨ *p1 == 1 ∨ *p2 == 1) ,((*p == p1 & *p == p2)) and (false) in four refinement steps. DASH starts with the initial partition {{ pc = i } | $0 \leq i \leq 8$ }, and the initial abstract program is isomorphic to the concrete programs control flow graph.

1. **Iteration1**: There is no witness since forest F doesn't contain a path that leads to error state. Along, there is no proof since Abstraction A contains an abstract error trace { 0, 1, 2, 3, 4, 5 } and Frontier is 0. Then DASH extend one transition from 0 to 1 beyond the frontier by giving feasible inputs. It results in adding concrete path { 0, 1, 2, 3, 4, 6, 7, 8} to Forest F.

2. **Iteration2**: There is no witness and proof. Abstract A contains an abstract error trace { 0, 1, 2, 3, 4, 5} and the frontier is 5. But DASH couldn't make a transition at least one step from 4 to 5 beyond frontier because of infeasible constraints from 0 to frontier. So, DASH uses this information to refine the state 4 using the WP$\alpha$ operator applying on edge from 3 to 4 and on state 4. Now DASH splits the state 3 into 3∧ P and 3∧¬P where P = (*p1 == 1 ∨ *p2 == 1).

3. **Iteration3**: There is no witness and proof. Abstraction A contains an abstract error trace { 0, 1, 2, 3, 4∧P, 5 } and frontier is 4∧P. But DASH couldn't make a transition from 2 to 3∧P at least one step beyond frontier. So, DASH uses this inability of extension and refines the state 3 into 3∧Q and 3∧¬Q where Q = ((*p == p1 & *p == p2) ∨ *p1 == 1 ∨ *p2 == 1).

4. **Iteration4**: There is no witness and proof. Abstraction A contains an abstract error trace { 0, 1, 2, 3∧Q, 4∧P, 5 } and frontier is 3∧Q. But DASH couldn't make a transition from 2 to 3∧Q at least one step beyond frontier. So, DASH uses this information and refines the state 2 into 2∧R and 2∧¬R where R = (*p == p1 & *p == p2).

5. **Iteration5**:There is no witness and proof. Abstraction A contains an abstract error trace { 0, 1, 2∧R, 3∧Q, 4∧P, 5 } and frontier is 2∧R. But DASH couldn't make a transition from 1 to 2∧R at least one step beyond frontier. So, DASH uses this information and refines the state 1 into 1∧S and 1∧¬S where S = (false) and state 1∧S and its transitions will be removed from abstraction since it is a false state.

6. **Iteration6**: There is no witness. But there is no abstract error path from state 0 to error state. This abstraction acts as proof for saying that the assertion will not be violated by any test case.

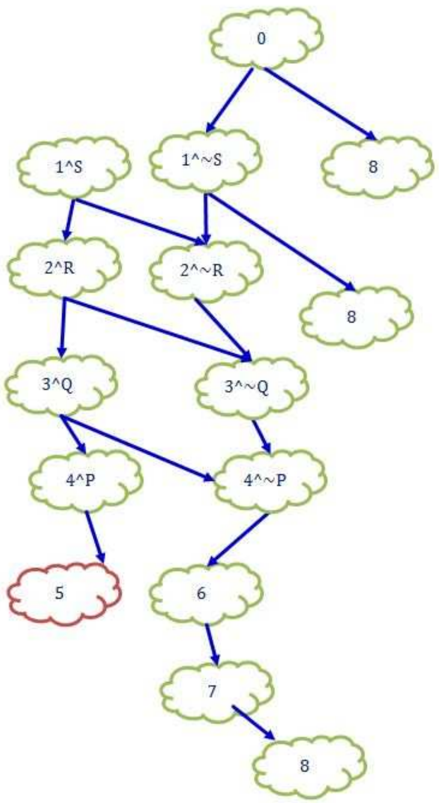


Figure 5.3: Finite indexed abstraction

## 5.4 Handling Inter procedure calls

This section describes way of handling inter procedure calls by DASH algorithm. This has been illustrated through example.

### 5.4.1 Description

Without loss of generality, assume that property $\psi$ that we wish to check is only associated with the main procedure $\mathrm{P}_0$. DASH maintains abstraction A and Forest F for each procedure. The main difference between intra procedural and inter procedural analysis occurs only if the frontier that is going to be extended is a function call. Informally, the inter procedural algorithm works by recursively invoking DASH whenever the standard algorithm dictates that the frontier must be extended across a call-return edge of the graph. The results of recursive call, combined with information from the calling context tell us whether or not there exists a test that can extend the frontier. If this is not possible, then the proof returned by the recursive DASH call is used to compute a suitable predicate.

The another notable change in handling inter procedural calls is handling the function calls between initial state to frontier. Essentially, the abstract error trace 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 any edge in between initial state to frontier is a function call, then DASH runs a test, that satisfies the pre condition, on function and replaces that function call edge by the sequence of states visited by that test. In order to extend the frontier, symbolic execution will be carried on the trace whose function calls are replaced by sequence of states visited by those tests.

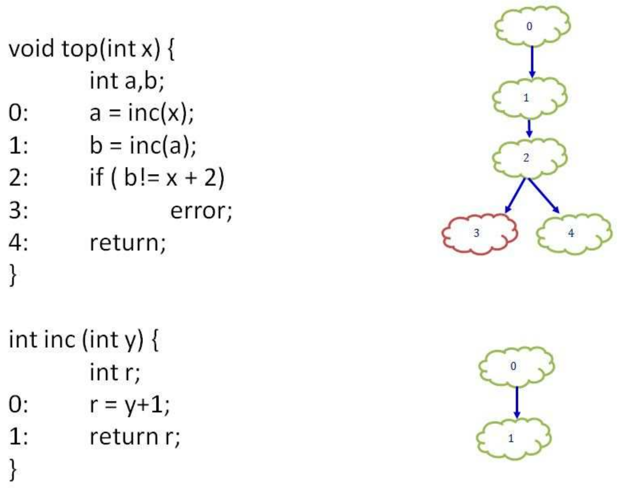
Formally, when the frontier is a function call, then DASH will itself recursively invoked with the following parameters. The constraints on function caller inputs variables are the conditions obtained by symbolic execution until frontier. These acts as initial conditions on inputs of caller function. The assertion that would be checked will be the negation of same assertion but the returned expression in caller function replaces the variable in the caller that stores the return value.

If recursive call to DASH return the fail answer, then it indicates that such a test is feasible and results in extension of the frontier which is a function call. If recursive call to DASH returns the true answer, then it indicates that such a test is not feasible and results in refinement of the region before frontier according to the following predicate.

$$\text{Suitable predicate} = \neg(\vee \rho_i)$$

where $\rho_i$ is a predicate in proof to split the initial region.

### 5.4.2 Example



```
void top(int x) {
        int a,b;
0:      a = inc(x);
1:      b = inc(a);
2:      if ( b!= x + 2)
3:              error;
4:      return;
}

int inc (int y) {
        int r;
0:      r = y+1;
1:      return r;
}
```

Figure 5.4: Example on which DASH handles inter procedural calls

Fig 5.4 contains the program and its abstraction. DASH is able to prove the property by discovering the predicates ( b $\neq$ x + 2), ( a $\neq$ x + 1), (false) in three refinement steps. Initially DASH creates abstractions $A_{top}$, $A_{inc}$ for both procedures 'top' and 'inc'. Then DASH creates empty forests $F_{top}$, $F_{inc}$ to both procedures.

1. **Iteration 1**: There is no witness since forest $F_{top}$ doesn't contain a path that leads to error state. Along, there is no proof since Abstraction $A_{top}$ contains an abstract error trace { 0, 1, 2, 3 } and Frontier is 0. Then DASH extend one transition from 0 to 1 beyond the frontier by giving feasible inputs as x = 10. It results in adding concrete path { 0, 1, 2, 4 } to Forest $F_{top}$.

2. **Iteration 2**: There is no witness and proof. Abstract $A_{top}$ contains an abstract error trace { 0, 1, 2, 3} and the frontier is 3. But DASH couldn't make a transition at least one step from 2 to 3 beyond frontier because of infeasible constraints from 0 to frontier. So, DASH uses this information to refine the state 2 using the WP operator applying on edge from 2 to 3 and on state 3. Now DASH splits the state 3 into 3$\wedge$ P and 3$\wedge\neg$P where P = ( b $\neq$ x + 2).

3. **Iteration 3**: There is no witness and proof. Abstraction $A_{top}$ contains an abstract error trace { 0, 1, 2, 3 } and frontier is 3$\wedge$P. But DASH couldn't make a transition from 2 to 3$\wedge$P at least one step beyond frontier. So, DASH uses this inability of extension and tries to refines the state 2. But DASH notices this as an inter procedural call and invokes itself as follows. Input constraints: ( a = x + 1, y = a), Assertion: ( ret $\neq$ x + 2), Procedure: (inc)

   (a) **Iteration 3.1**: The return node in $A_{inc}$ is split into 2$\wedge P_1$ and 2$\wedge\neg$ $P_1$ where $P_1$ = ( ret $\neq$ x + 2). There is no witness and proof. Abstraction $A_{inc}$ contains an abstract error trace { 0, 1$\wedge P_1$ } and frontier is 1$\wedge P_1$. But DASH couldn't make transition at least one step from 0 to 1$\wedge P_1$ beyond frontier. So, DASH refines the state 1 into 1$\wedge Q_1$ and 1$\wedge\neg Q_1$ where $Q_1$ = (false) and state 1$\wedge Q_1$ and its transitions are removed from abstraction $A_{inc}$.

(b) **Iteration 3.2**: There is no witness in forest. But there is no abstract error trace from state 0 to error state. The abstraction acts as proof and return the suitable predicate to refine as ( y $\neq$ x + 1).

Now DASH splits the state 2 into 1$\wedge$Q and 1$\wedge\neg$Q where Q = ( a $\neq$ x + 1).

4. **Iteration 4**: There is no witness and proof. Abstraction $A_{top}$ contains an abstract error trace { 0, 1$\wedge$Q, 2$\wedge$P, 3 } and frontier is 1$\wedge$Q. But DASH couldn't make a transition from 0 to 1$\wedge$Q at least one step beyond frontier. So, DASH uses this information and refines the state 0. But DASH notices this an inter procedural call and invokes itself as follows. Input constraints: true, Assertion: ( a$\neq$ x + 1), procedure: top

   (a) **Iteration 4.1**: DASH creates a new abstraction $A_{inc}$ and Forest $F_{inc}$ for procedure 'inc'. The return node in $A_{inc}$ is split into 2$\wedge P_1$ and 2$\wedge\neg$ $P_1$ where $P_1$ = ( ret $\neq$ x + 1). There is no witness and proof. Abstraction $A_{inc}$ contains an abstract error trace { 0, 1$\wedge P_1$ } and frontier is 1$\wedge P_1$. But DASH couldn't make transition at least one step from 0 to 1$\wedge P_1$ beyond frontier. So, DASH refines the state 1 into 1$\wedge Q_1$ and 1$\wedge\neg Q_1$ where $Q_1$ = (false) and state 1$\wedge Q_1$ and its transitions are removed from abstraction $A_{inc}$.

   (b) **Iteration 4.2**: There is no witness in forest. But there is no abstract error trace from state 0 to error state. The abstraction acts as proof and return the suitable predicate to refine as (false).

Now DASH splits the state 0 into 0$\wedge$Q and 0$\wedge\neg$Q where Q = (false) and state 0$\wedge$Q and it's transitions are removed from abstraction $A_{top}$.

5. **Iteration 5**: There is no witness. But there is no abstract error path from state 0 to error state. This abstraction acts as proof for saying that the assertion will not be violated by any test case.

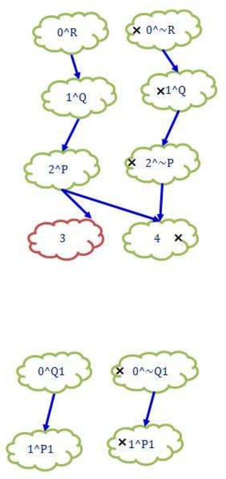


Figure 5.5: Finite indexed abstraction

## 5.5 Conclusion

DASH is a verification algorithm, an extension of Synergy algorithm for checking the safety properties of sequential programs. But DASH is handling only sequential programs. Currently, research work is going to improve the DASH to support concurrent program analysis and liveness properties of program. This algorithm is not scalable which is a hidden limitation. If there is are multiple functional calls with similar arguments, then DASH will be recursively invoked for all those calls which is a time consuming work. It handles constraints in only first order predicate logic. A progress is going on in the direction of approximate symbolic execution to support more complex constraints.

# Chapter 6

# SMASH

Generally, may analysis provides information that is true of all program executions and is used to prove absence of bugs and must analysis provides information that is true of some program execution and is used to prove presence of bugs. SMASH [17] is 3 valued Compositional May-Must analysis algorithm which computes may and must information at each procedure boundaries for later reuse. It performs both may analysis and must analysis simultaneously, and uses both may summaries and must summaries to improve the effectiveness as well as the efficiency of the analysis. It has been implemented using predicate abstraction for may analysis and DART, symbolic execution for must analysis.

## 6.1 Motivation and Introduction

DASH algorithm is introduced to handle the pointers and function calls in programs for analysis. Scalability is an important feature of program analysis tools. Typically, DASH algorithm re invokes itself for each function call to analyze the callee function with constraints on inputs and outputs of the callee function. If a function call is repeatedly called with similar parameters , then DASH will recursively called for all function calls even though if the function call is on similar parameters. The basic motivation for introducing SMASH algorithm is to analyze the function call and save the analysis in the form of summaries for later reuse. If the function is called with similar parameters for next time, then summaries will be used without analyzing the function call once again by DASH.

The principle of SMASH algorithm is as follows [17]. "Compositional approaches to property checking involve decomposing the whole-program analysis into several sub-analyses of individual components, summarizing the results of these sub-analyses, and memorising (caching) those summaries for possible later re-use in other calling contexts. Summarizing at procedure boundaries is indispensable for scalability".

## 6.2 Summaries

This section introduces the notion of summaries for storing the analysis computed over a function.

SMASH performs a modular inter procedural analysis and incrementally decomposes this reachability query into several sub-queries that are generated in a demand-driven manner. Each sub-query is of the form of $\langle \phi_1 \overset{?}{\Longrightarrow} f\phi_2\rangle$ , where $\phi_1$ and $\phi_2$ are state predicates representing respectively a precondition (call- ing context) and postcondition (return context) for a procedure f (or block) in P . The answer to such a query is yes if there exists an execution of f starting in some state $\sigma_1 \in \phi_1$ and terminat- ing in some state $\sigma_2 \in \phi_2$ , no if such an execution does not exist, and unknown (maybe) if the algorithm is unable to deci- sively conclude either way (the last option is needed since program verification is undecidable in general). SMASH uses may and must summaries to answer queries.

### 6.2.1 May summary

A may summary of a procedure P is of the form $\langle \varphi_1 \overset{May}{\Longrightarrow} P\varphi_2\rangle$ where $\varphi_1$ and $\varphi_2$ are predicates over program may states.

**Meaning:** If we invoke procedure P from any state satisfying $\varphi_1$ , the set of all possible states of the program on termination of P is over-approximated by the set of states $\varphi_2$ . This implies that no states satisfying $\neg\varphi_2$ are reachable from states satisfying $\varphi_1$ by executing P.

**Usage:** Intuitively, a may summary of a procedure represents a property that is guaranteed to be true about all executions of the procedure, and a must summary of a procedure represents witness executions of the procedure that are guaranteed to exist. May summaries pro- vide obvious benefits to improving the efficiency of may analy- sis: when a compositional may analysis requires a sub-query for a procedure P , a previously-computed may summary for P can po- tentially be re-used to answer that query without re-analyzing the procedure.

A may summary $\langle \varphi_1 \overset{May}{\Longrightarrow} P\varphi_2\rangle$ implies that, for any state x $\in \psi_1$ , for any state y such that the execution of f starting in state x terminates in state y, we have y $\in \psi_2$.

### 6.2.2 ¬ May summary

A not may summary of a procedure P is of the form $\langle \varphi_1 \overset{\neg May}{\Longrightarrow} P\varphi_2\rangle$ where $\varphi_1$ and $\varphi_2$ are predicates over program may states.

**Meaning:** If we invoke procedure P from any state satisfying $\varphi_1$ , the possible states of the program on termination of P does not belong to the set of states $\varphi_2$ . This implies that states satisfying $\neg\varphi_2$ are not reachable from states satisfying $\varphi_1$ by executing P.

**Usage:** A not-may summary $\langle \psi_1 \overset{\neg May}{\Longrightarrow} f\psi_2\rangle$ implies that for any state x $\in \psi_1$ , there does not exist a state y $\in \psi_2$ such that the execution of f starting in state x terminates in state y. Clearly, a not-may summary $\langle \psi_1 \overset{May}{\Longrightarrow} f\psi_2\rangle$ can be used to give a no answer to a query $\langle \varphi_1 \overset{May}{\Longrightarrow} f\varphi_2\rangle$ for f provided that $\varphi_1 \subseteq \psi_1$ and $\varphi_2 \subseteq \psi_2$.

### 6.2.3 Must summary

A must summary of a procedure P is of the form $\langle\varphi_1 \overset{\neg May}{\Longrightarrow} P\varphi_2\rangle$ where $\varphi_1$ and $\varphi_2$ are predicates over program may states.

**Meaning:** If we invoke procedure P from any state satisfying $\varphi_1$ , the set of all possible states of the program on termination of P is under-approximated by the set of states $\varphi_2$ . This implies that any state satisfying $\varphi_2$ is guaranteed to be reachable from some state satisfying $\varphi_1$ by executing P .

**Usage:** A must summary $\langle\psi_1 \overset{Must}{\Longrightarrow} f\psi_2\rangle$ implies that, for every state y $\in \psi_2$ , there exists a state x $\in \psi_1$ such that the execution of f starting in state x $\in \psi_1$ terminates in state y $\in \psi_2$. Thus a must summary $\langle\psi_1 \overset{Must}{\Longrightarrow} f\psi_2\rangle$ can be used to give a yes answer to a query $\langle\varphi_1 \overset{May}{\Longrightarrow} f\varphi_2\rangle$ provided that $\varphi_1 \subseteq \psi_1$ and $\varphi_2 \cap \psi_2 = \{\}$.

## 6.3 Differences between SMASH and DASH

There are two differences between SMASH and DASH and are aroused due to introduction of concept of summaries. These two are explained as follows.

At first, the frontier, that is going to be extended, is a function call. DASH will be recursively invoked to analyze the query even though there exists a similar query that has been analyzed by DASH. But unlike DASH, when a query $\langle\varphi_1 \overset{?}{\Longrightarrow} P\varphi_2\rangle$ is given over a function P, SMASH chooses one of the following actions.

1. If there exist a previously computed ¬May summary $\langle\hat{\varphi_1} \overset{\neg May}{\Longrightarrow} P\hat{\varphi_2}\rangle$ that answers the given query, then SMASH will use this ¬May summary directly without analyzing the procedure P.

2. If there exist a previously computed must summary $\langle\hat{\varphi_1} \overset{must}{\Longrightarrow} P\hat{\varphi_2}\rangle$ that answers the given query, then SMASH will use this must summary directly without analyzing the procedure P.

3. Otherwise, SMASH will analyze the procedure P and will make either ¬May summary or Must summary to answer the given query.

Secondly, existence of function calls in between the initial state to state beyond the frontier. In DASH, when ever there is a function call between initial state and state before frontier, DASH will take an input that satisfies state predicate at function call and replaces the function call in abstract error trace by sequence of states visited by function execution on that input. Where as in SMASH, if there is any summary that helps for function call, SMASH uses for proceeding further in symbolic execution from initial state to state beyond the frontier.

## 6.4 Illustration

In this section, we will explain the way of computing summaries and usage of it for other queries. At first, we will illustrate SMASH algorithm over examples which computes ¬May, must summaries. Secondly, we will illustrate SMASH algorithm over examples which uses ¬May / Must summary to compute Must / ¬May summary respectively.

### 6.4.1 Computing ¬May summary

We consider the following example to show the computation of ¬May summary to prove that error will not be reached by any inputs to *main* function.

```
void main(int i1,i2) {             int g ( int i ) {
0: int x1,x2;                      0:  if ( i > 0 )
1: x1 = g(i1);                     1:    return i;
2: x2 = g(i2);                           else
3: if ((x1 < 0)||(x2 < 0))         2:    return -i;
4:   error;                             }
5}
```

Initially, assume that there are no summaries associated with both functions. Now, SMASH tries to find an execution in *main* function along the path 0, 1, 2, 3, 4 to reach *error.* Since the frontier is edge from 3 to 4 and SMASH couldn't extend the frontier, SMASH refines the state 3 in *main* function to 3∧P, 3∧¬P and issues the following query to extend the new frontier from 2 to 3∧P.

$$\langle \text{true} \overset{?}{\Longrightarrow}_g (\text{retval} < 0)\rangle \tag{6.1}$$

where retval denotes the return value of the function *g.* Since there are no summaries currently that answers above query, SMASH starts analyzing the function g. Since all the paths in the function *g* returns the non negative values, SMASH encodes this analysis in the form of ¬ May summary as follows.

$$\langle \text{true} \overset{\neg May}{\Longrightarrow}_g (\text{retval} < 0)\rangle \tag{6.2}$$

Then SMASH refines the state 2 in *main* to 2∧Q, 2∧¬Q and tries to extend the frontier from state 1 to state 2∧Q. Now, SMASH issues the following query to extend the frontier.

$$\langle \text{true} \overset{?}{\Longrightarrow}_g (\text{retval} < 0)\rangle \tag{6.3}$$

At this point of time, SMASH is different from DASH because SMASH uses the existing ¬May summary to answer the above query where as DASH analyzes the function *g* to answer the query. So, SMASH uses the existing summary to refine the state 1 and proves that there exists non inputs to function *main* which will make program execution to reach error and encodes this entire summary over function *main* as follows.

$$\langle \text{true} \overset{\neg May}{\Longrightarrow}_{main} (error)\rangle \tag{6.4}$$

### 6.4.2 Computing Must summary

We consider the following example to show the computation of must summary and finds an input which will make program execution to reach error in *main* function.

```
void main(int i1,i2) {          int g( int i) {
0: int x1,x2;                   0:  if ( i > 0 )
1: x1 = f(i1);                  1:     return i;
2: x2 = f(i2);                      else
3: if (x1 > 0 & x2 > 0)         2:     return -1;
4:    error;                    }
}
```

Initially, assume that there are no summaries associated with both functions. Now, SMASH tries to find an execution in *main* function along the path 0, 1, 2, 3, 4 to reach *error*. Since the frontier is edge from 3 to 4 and SMASH couldn't extend the frontier, SMASH refines the state 3 in *main* function to 3∧P, 3∧¬P and issues the following query to extend the new frontier from 2 to 3∧P.

$$\langle \text{true} \overset{?}{\Longrightarrow}_g (\text{retval} > 0)\rangle \tag{6.5}$$

where retval denotes the return value of the function *g*. Since there are no summaries currently that answers above query, SMASH starts analyzing the function g. The path through *else* branch of function *g* doesn't answer the query because the return value is -1 which is not greater than 0. Since the path through *if* branch of function *g* satisfies the condition ( retval > 0 ) and encodes this analysis in the form of must summary as follows.

$$\langle \text{i} > 0 \overset{Must}{\Longrightarrow}_g (\text{retval} > 0)\rangle \tag{6.6}$$

Since SMASH could extend the frontier from 2 to 3∧P, SMASH will try to extend the frontier from 3∧P to 4 for reaching error. So, SMASH tries to generate an input by doing symbolic execution from 0 to 3∧. At this point, SMASH is different from DASH. In DASH, when ever there is a function call between initial state and state before frontier, DASH will take an input that satisfies state predicate at function call and replaces the function call in abstract error trace by sequence of states visited by function execution on that input. Where as in SMASH, if there is any summary that helps for function call, SMASH uses, the intersection of state predicate at function call and pre condition of summary, for proceeding further in symbolic execution.

Here also, SMASH uses the existing must summary at the edge from 1 to 2 and 2 to 3( function calls) for symbolic execution and generates constraints ( i1 > 0 & i2 > 0). Now, theorem prover solves the constraints and generates the feasible inputs. Finally, program execution on those feasible inputs leads to reaching error and can be encoded as following must summary for *main* function.

$$\langle \text{i1} \; > 0\& \text{ i2} > 0 \overset{Must}{\Longrightarrow}_g (\text{retval} > 0)\rangle \tag{6.7}$$

### 6.4.3 Computing ¬May summary using Must summary

We consider the following example to show the computation of ¬May summary using must summary and proves that error will not be reached by any inputs to *main* function.

```
void main ( int i ) {
0: int a = g(i);
1: if ( a < 0)
2:   error;
3:}
```

```
int g (int i) {
0:  int a = h(i);
1:  if ( a < 10)
2:     return k(a);
    else
3:     return 0;}
```

```
int k ( int i) {
0:  if ( i > 0)
1:     return i;
     else
2:     return -i; }
```

In above example, Let h be a complex function and hard to analyze. Assume that function h has following must summaries.

$$\langle \mathrm{i} > 10 \overset{must}{\Longrightarrow}_h (15 < \mathrm{retval} < 20)\rangle \tag{6.8}$$

$$\langle \mathrm{i} < 14 \overset{must}{\Longrightarrow}_h (\mathrm{retval} < 11)\rangle \tag{6.9}$$

Now, SMASH tries to find an execution in *main* function along the path 0, 1, 2 to reach *error.* Since the frontier is edge from 1 to 2 and SMASH couldn't extend the frontier, SMASH refines the state 1 in *main* function to 1∧P, 1∧¬P and issues the following query to extend the new frontier from 0 to 1∧P.

$$\langle \mathrm{true} \overset{?}{\Longrightarrow}_g (\mathrm{retval} < 0)\rangle \tag{6.10}$$

where retval denotes the return value of the function *g.* Since there are no summaries currently over function *g* that answers above query, SMASH starts analyzing the function g. SMASH uses the must summary over h to calculate under approximation to reach if statement. It basically prevent a possibly expensive and hopeless not-may proof that one of those two branches is not feasible. The path through *else* branch of function *g* doesn't answer the query because the return value is 1 which is not equal to 0. Now, SMASH follows the *if* branch of *g* function to answer the above query and issues the following query.

$$\langle \mathrm{i} < 10 \overset{?}{\Longrightarrow}_k (\mathrm{retval} < 0)\rangle \tag{6.11}$$

Since there exists no summary over function *k* that answers the above query, SMASH analyzes the function *k.* Since all the paths in the function *k* returns the non negative values, SMASH encodes this analysis in the form of ¬May summary as follows.

$$\langle \mathrm{true} \overset{\neg May}{\Longrightarrow}_k (\mathrm{retval} < 0)\rangle \tag{6.12}$$

Now, SMASH uses this summary to answer the above query. In turn, it results in answering query over the function *g.*

$$\langle \mathrm{true} \overset{\neg May}{\Longrightarrow}_g (\mathrm{retval} < 0)\rangle \tag{6.13}$$

Finally, with the above ¬May summary, SMASH concludes that there exists no inputs to *main* function which makes the execution to reach error. It can be encoded as follows.

$$\langle \mathrm{true} \overset{\neg May}{\Longrightarrow}_{main} (\mathrm{error})\rangle \tag{6.14}$$

Here SMASH uses the must summary of function *h* to avoid a may-analysis over the return value of function *h* while still being able to build a not-may analysis of the function *main*.

### 6.4.4 Computing Must summary using ¬May summary

We consider the following example to show the computation of must summary using ¬May summary and finds an input which will make program execution to reach error in *main* function.

```
 void main ( int i ) {
0: int a = g(i);
1: if ( a == 0)
2:   error;
3:}
```

```
  int g (int i) {
 0:  if ( i > 0 )
1:    return h(i) + 1;
     else
 2:    return 0 ;
 3:}
```

In above example, Let h be a complex function and hard to analyze. Assume that function h has following ¬May summary.

$$\langle \text{true} \overset{\neg May}{\Longrightarrow}_h (\text{retval} = -1)\rangle \tag{6.15}$$

Now, SMASH tries to find an execution in *main* function along the path 0, 1, 2 to reach *error.* Since the frontier is edge from 1 to 2 and SMASH couldn't extend the frontier, SMASH refines the state 1 in *main* function to 1∧P, 1∧¬P and issues the following query to extend the new frontier from 0 to 1∧P.

$$\langle \text{true} \overset{?}{\Longrightarrow}_g (\text{retval} = 0)\rangle \tag{6.16}$$

where retval denotes the return value of the function *g.* Since there are no summaries currently over function *g* that answers above query, SMASH starts analyzing the function g. Now, SMASH follows the *if* branch of *g* function to answer the above query and issues the following query.

$$\langle \text{i} > 0 \overset{?}{\Longrightarrow}_h (\text{retval} = -1)\rangle \tag{6.17}$$

Since there is an existing ¬May summary that answers the above query over function *h*, SMASH uses this summary and replies as follows.

$$\langle \text{i} > 0 \overset{\neg May}{\Longrightarrow}_h (\text{retval} = -1)\rangle \tag{6.18}$$

Now, SMASH explores the *else* branch of function *g.* The path through *else* branch of function *g* answer the query because the return value is 0 which is equal to 0. Now, SMASH after exploring both branches of function *g* comes to conclusion and replies to the query given to it as follows.

$$\langle \text{i} \leq 0 \overset{must}{\Longrightarrow}_g (\text{retval} = 0)\rangle \tag{6.19}$$

Finally, with the above must summary, SMASH concludes that there exists inputs to *main* function which makes the execution to reach error and makes the following must summary.

$$\langle \text{i} \leq 0 \overset{must}{\Longrightarrow}_{main} (\text{error})\rangle \tag{6.20}$$

# Chapter 7

# Problem Statement

The SMASH algorithm, as described in previous chapter, make summaries at procedure boundaries and utilize those summaries for further queries on that procedure. But SMASH algorithm couldn't handle certain class of recursive programs. It is as stated in paper [17] that SMASH would not be guaranteed to terminate over recursive programs which have infinite data type domains or dynamic memory allocations. In this chapter, we first provide an example of such program. Secondly, it was not known whether SMASH can be terminated on all programs having finite data type domains and not having dynamic memory allocations. We then show an example where SMASH fails on a program with above two properties.

## 7.1 Recursive procedures of infinite data type domain

In this section, we consider the following program to show non termination of SMASH on the recursive procedures containing infinite data type domain.

```
int main ( int x ) {
0:      int a = g(x);
1:      if  ( a < 0)
2:        error;
3:      return 1;
  }
```

```
int g (int x) {
0:     if ( x == 1 )
1:        return 1;
       else
2:        return g(x-1) + 1;
  }
```

In the above example, *g* is a recursive function with an input x and domain of data type of input x is infinite. Initially, assume that there are no summaries associated with both functions. Now, SMASH tries to find an execution in *main* function along the path 0, 1, 2 to reach *error*. Then, SMASH issues following query after performing some analysis on *main* function.

$$\langle \text{true} \overset{?}{\Longrightarrow}_g (\text{retval} < 0) \rangle \tag{7.1}$$

where retval denotes the return value of the recursive function *g*. Since there are no summaries currently that answers above query, SMASH starts analyzing the function g. Since the return value from the path following *if* branch is 1, SMASH will immediately conclude that none of paths through *if* branch don't answer the query. Then, SMASH issues the following query by following the path towards *else* branch in function *g*.

$$\langle \text{x != 0} \overset{?}{\Longrightarrow}_g (\text{retval} < -1)\rangle \tag{7.2}$$

While analyzing the above query over same function *g*, SMASH issues following query by following the path towards *else* branch in function *g*.

$$\langle \text{ x != -1 \& x != 0 } \overset{?}{\Longrightarrow}_g (\text{retval} < -2)\rangle \tag{7.3}$$

Similarly, SMASH issues the following query in order to answer the above query.

$$\langle \text{ x != -2 \& x != -1 \& x != 0 } \overset{?}{\Longrightarrow}_g (\text{retval} < -3)\rangle \tag{7.4}$$

. . . . . . . . . . . . . . .

Since base case doesn't answer any query, SMASH will keep on making queries continuously and leads to non termination. So, if the domain of data types of recursive procedures is infinite, then SMASH wouldn't be guaranteed to terminate.

## 7.2 Recursive procedures of finite data type domain

In this section, we first consider a recursive procedure, containing finite domain of data types and not allowing dynamic memory allocations, to show termination of the SMASH. Secondly, we consider another recursive program with above two properties to show non termination of SMASH.

### 7.2.1 SMASH - Terminating

```
int main ( int x ) {
0:      int a = g(x);
1:      if  ( a < 0)
2:        error;
3:      return 1;
  }
```

```
int g (int x) {
0:     if ( x < 0 or x > 3 ) \\ For finite domain
1:        return 0;
       else
2:        if ( x == 1)
3:            return 1;
          else
4:            return g(x-1) + 1;
}
```

In the above example, *g* is a recursive function with input x. Domain of data type is the set of all possible values taken by a variable of that data type. In order to express the finiteness ( x$\in \{0, 1, 2, 3\}$) of domain of data type of x, lines 0 and 1 are explicitly added to function *g*. Now, x will take value only from it's domain. Assume that there are no summaries associated with both functions.

Now, SMASH tries to find an execution in *main* function along the path 0, 1, 2 to reach *error*. Then, SMASH issues following query after performing some analysis on *main* function.

$$\langle \text{true} \overset{?}{\Longrightarrow}_g (\text{retval} < 0) \rangle \tag{7.5}$$

Since there are no summaries currently that answers above query, SMASH starts analyzing the function *g*. Since the return value from the path following first *if* branch is 0, SMASH will immediately conclude that none of paths through first *if* branch don't answer the query. So, SMASH follows the first *else* branch to answer the query. Since the return value from the paths following second *if* branch is 1, SMASH will immediately conclude that none of paths through second *if* branch don't answer the query. So, SMASH follows the second *else* branch and issues another query to answer current query.

$$\langle x \in \{-1, 1, 2\} \overset{?}{\Longrightarrow}_g (\text{retval} < -1) \rangle \tag{7.6}$$

Since x = -1 doesn't belong to domain of data type of variable x, the query will be simplified to as follows.

$$\langle x \in \{1, 2\} \overset{?}{\Longrightarrow}_g (\text{retval} < -1) \rangle \tag{7.7}$$

While analyzing the above query over same function *g*, SMASH issues the following query.

$$\langle x \in \{1\} \overset{?}{\Longrightarrow}_g (\text{retval} < -2) \rangle \tag{7.8}$$

Since the return value of base case is 1 and is less than -2, SMASH encodes the analysis as ¬ May summary as follows.

$$\langle x \in \{1\} \overset{\neg May}{\Longrightarrow}_g (\text{retval} < -2) \rangle \tag{7.9}$$

In turn, it results in following summaries using above ¬may summary.

$$\langle x \in \{1, 2\} \overset{\neg May}{\Longrightarrow}_g (\text{retval} < -1) \rangle \tag{7.10}$$

$$\langle \text{true} \overset{\neg May}{\Longrightarrow}_g (\text{retval} < 0) \rangle \tag{7.11}$$

Since the query is asked over values that are out of domain of data type of x and base case answers the query, SMASH gets terminated by concluding with an ¬May summary. Finally, SMASH has proven that there exist no inputs to *main* function that reaches error. Thus, SMASH is terminated over this recursive procedure containing finite domain of data types and not allowing dynamic memory allocations.

### 7.2.2 SMASH - Non terminating

Now, we consider another recursive procedure, containing finite domain of data types and not allowing dynamic memory allocations, to show non termination of the SMASH.

```
int f ( int x ) {
0:    int a = g(x);
1:    if ( a < 0 )
2:      error;
3:    return 1;
}
```

```
int g ( int x) {
0:    if ( x < 1 and x > 8 )  \\ For finite domain
1:       return 0;
      else
2:       if ( x == 1)
3:          return 1;
4:       else
5:          a = h(x);
6:          b = g(a-1);
7:          return b;
}
```

In the above example, *g* is a recursive function on inputs x. In order to express the finiteness ( x$\in \{1, 2, 3, 4, 5, 6, 7, 8\}$) of domain of data type of x, lines 0 and 1 are explicitly added to function *g*. Now, x will take value only from it's domain. Let h be a complex function and hard to analyze. Assume that function h has following must summary.

$$\langle \text{ x } = 5 \overset{must}{\Longrightarrow}_h \text{ ( retval } = 6)\rangle \tag{7.12}$$

Now, SMASH tries to find an execution in *main* function along the path 0, 1, 2 to reach error. Then SMASH issues the following query to reach error statement.

$$\langle \text{true} \overset{?}{\Longrightarrow}_g \text{ ( retval } < 0)\rangle \tag{7.13}$$

Since there are no summaries currently that answers above query over function *g*, SMASH starts analyzing the function *g*. Since the return value from the path following first *if* branch is 0, SMASH will immediately conclude that none of paths through first *if* branch don't answer the query. So, SMASH follows the first *else* branch to answer the query. Since the return value from the paths following second *if* branch is 1, SMASH will immediately conclude that none of paths through second *if* branch don't answer the query. So, SMASH follows the second *else* branch. It uses the must summary of function *h* to calculate under approximation and issues the following query.

$$\langle \text{x} = 5 \overset{?}{\Longrightarrow}_g \text{ ( retval } < 0)\rangle \tag{7.14}$$

While analyzing the above query, SMASH uses the must summary of function *h* similarly and issues the following query.

$$\langle \text{x} = 5 \overset{?}{\Longrightarrow}_g \text{ ( retval } < 0)\rangle \tag{7.15}$$

................. Since the post condition of must summary of function *h* is same as pre-condition of given query, SMASH will keep on making same query continuously and leads to non termination. Thus, SMASH is not terminated over this recursive procedure containing finite domain of data types and not allowing dynamic memory allocations.

# Chapter 8

# Solution

In this chapter, we show a way to handle the problem that has been discussed in last chapter. Then, we illustrate the solution over an example.

## 8.1 Feasible Solution

This section explains the solution for termination of SMASH algorithm on recursive programs. Typically, when a query $\langle \varphi_1 \overset{?}{\Longrightarrow} P\varphi_2 \rangle$ is given over a function P, SMASH chooses one of the following actions.

1. If there exist a previously computed ¬May summary $\langle \hat{\varphi}_1 \overset{\neg May}{\Longrightarrow} P\hat{\varphi}_2 \rangle$ that answers the given query, then SMASH will use this ¬May summary directly without analyzing the procedure P.

2. If there exist a previously computed must summary $\langle \hat{\varphi}_1 \overset{must}{\Longrightarrow} P\hat{\varphi}_2 \rangle$ that answers the given query, then SMASH will use this must summary directly without analyzing the procedure P.

3. Otherwise, SMASH will analyze the procedure P and will make either ¬May summary or Must summary to answer the given query.

In-progress queries is the set of all queries that are in progress of answering the query given to it. Essentially, we need to keep track for each function which queries are in progress. This is needed to check whether the current query is answerable from any in-progress query. The basic problem that has been encountered in last section is that queries are populated by SMASH that can be answerable from in-progress queries and in turn leads to population of similar queries. This results in non termination of SMASH algorithm over certain class of recursive programs. The following is the probable solution to solve the problem that has been encountered in earlier section.

1. If there exist a previously computed ¬May summary $\langle \hat{\varphi}_1 \overset{\neg may}{\Longrightarrow} P\hat{\varphi}_2 \rangle$ that answers the given query, then SMASH will use this ¬May summary directly without analyzing the procedure P.

2. If there exist a previously computed must summary $\langle \hat{\varphi}_1 \overset{must}{\Longrightarrow} P\hat{\varphi}_2 \rangle$ that answers the given query, then SMASH will use this must summary directly without analyzing the procedure P.

3. If there exist no in-progress query that can answer the given query, then allow SMASH to analyze the procedure P and make either ¬May summary or Must summary to answer the given query.

4. If there exists an in-progress query that can answer the given query, then reply to that query by making query itself as ¬May summary. Then, the state before frontier gets refined and partitions the state into state following path leading to query that is answerable from any in-progress query and another state which possibly make a new query over the same function. In this case, it leads to following situations.

   (a) If it encounters an error and lands up with must summary, then it can be used directly to answer the query.

   (b) If it doesn't encounter an error and land up with ¬May summary, then we cannot say that there are no errors because we have willingly removed the case where SMASH leads to non termination by populating the query, that is answerable from in-progress queries, as ¬May summary.

## 8.2 Illustration

Consider the same example in previous section, where SMASH results in non termination over the function which has finite domain of data types and not allowing dynamic memory allocations

```
int f ( int x ) {
0:    int a = g(x);
1:    if ( a < 0 )
2:      error;
3:    return 1;
}
```

```
int g ( int x) {
0:    if ( x < 1 and x > 8 )  \\ For finite domain
1:       return 0;
      else
2:       if ( x == 1)
3:          return 1;
4:       else
5:          a = h(x);
6:          b = g(a-1);
7:          return b;
}
```

In the above example, *g* is a recursive function on inputs x. In order to express the finiteness ( x$\in \{1,2,3,4,5,6,7,8\}$) of domain of data type of x, lines 0 and 1 are explicitly added to function *g*. Now, x will take value only from it's domain. Let h be a complex function and hard to analyze. Assume that function h has following must summary.

$$\langle \text{ x } = 5 \overset{must}{\Longrightarrow}_h \text{ ( retval } = 6)\rangle \tag{8.1}$$

Now, SMASH tries to find an execution in *main* function along the path 0, 1, 2 to reach error. Then SMASH issues the following query to reach error statement.

$$\langle \text{true} \overset{?}{\Longrightarrow}_g \text{ ( retval } < 0)\rangle \tag{8.2}$$

Since there are no summaries currently that answers above query over function $g$, SMASH starts analyzing the function $g$. Since the return value from the path following first *if* branch is 0, SMASH will immediately conclude that none of paths through first *if* branch don't answer the query. So, SMASH follows the first *else* branch to answer the query. Since the return value from the paths following second *if* branch is 1, SMASH will immediately conclude that none of paths through second *if* branch don't answer the query. So, SMASH follows the second *else* branch. It uses the must summary of function $h$ to calculate under approximation and issues the following query.

$$\langle \mathrm{x} = 5 \overset{?}{\Longrightarrow}_g (\text{ retval } < 0)\rangle \tag{8.3}$$

While analyzing the above query, SMASH uses the must summary of function $h$ similarly and issues the following query.

$$\langle \mathrm{x} = 5 \overset{?}{\Longrightarrow}_g (\text{ retval } < 0)\rangle \tag{8.4}$$

This is the point where the solution can be adapted and will be different from SMASH algorithm. Since the above query is answerable by in-progress query, this query will be converted to ¬May summary as follows.

$$\langle \ \mathrm{x} \ = 5 \overset{\neg May}{\Longrightarrow}_g (\text{ retval } < 0)\rangle \tag{8.5}$$

In turn, SMASH results in making the following summaries using ¬ May summary.

$$\langle \ \mathrm{x} \ = 5 \overset{\neg may}{\Longrightarrow}_g (\text{ retval } < 0)\rangle \tag{8.6}$$

Now, SMASH refines the regions using ¬May summary and generates the following query over function $h$.

$$\langle x \in \{2,3,4,5,6,7,8\} \overset{?}{\Longrightarrow}_h (\text{ retval } != 6)\rangle \tag{8.7}$$

and proceeds further.

# References

[1] G. J. Myers. The Art of Software Testing. *John Wiley and Sons, New York* .

[2] E. W. Dijkstra. The humble programmer. *Communications of ACM* 15, (1972) 859 – 866.

[3] P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Software* 25, (2008) 30–37.

[4] J. Edvardsson. A survey on automatic test data generation. *Computer Science and Engineering in Link ping* 21 – 28.

[5] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM* 19, (1976) 385–394.

[6] J. C. King and R. Floyd. An interpretation oriented theorem prover over integers. *Computer systems and science* 6, (1972) 305–323.

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In PLDI. 2005 213–223.

[8] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In ESEC/SIGSOFT FSE. 2005 263–272.

[9] P. Godefroid. Compositional dynamic test generation. In POPL. 2007 47–54.

[10] Z. Manna and A. Pnueli. Temporal verification of Reactive systems. *Springer* .

[11] R. W. Floyd. Assigning meaning to Programs. In in Proceedings of Symposium on Applied Mathematics. 1967 19–32.

[12] E. M. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, 1999.

[13] T. B. Sriram Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In in Proceedings of the Eighth International SPIN Workshop (SPIN 01), Lecture Notes in Computer Science 2057, Springer-Verlag,. 2001 103–122.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In POPL. 2002 58–70.

[15] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In SIGSOFT FSE. 2006 117–127.

[16] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. Tetali, and A. V. Thakur. Proofs from Tests. *IEEE Trans. Software Eng.* 36, (2010) 495–508.

[17] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In POPL. 2010 43–56.

[18] L. .de Moura, and N. Bjorner. Z3: An Efficient SMT Solver. In TACAS. 2008.