Proofs from Tests

Nels E. Beckman* nbeckman@cs.cmu.edu
Aditya V. Nori† adityan@microsoft.com
Sriram K. Rajamani † sriram@microsoft.com
Robert J. Simmons* rjsimmon@cs.cmu.edu

*Carnegie Mellon University †Microsoft Research India

January 2008
Technical Report
MSR-TR-

We present an algorithm Dash to check if a program P satisfies a safety property φ. The unique feature of the algorithm is that it uses only test generation operations, and it refines and maintains a sound program abstraction as a consequence of failed test generation operations. Thus, each iteration of the algorithm is extremely inexpensive, and can be implemented without any extra theorem prover calls and without any global may-alias information. In particular, we introduce a new refinement operator WP_α that uses only the alias information obtained by executing a test to refine abstractions in a sound manner. We present a full exposition of the Dash algorithm, its theoretical properties, and its implementation. Our empirical results show significant improvement over Slam in the runtimes of several examples. We also show several cases where Slam is unable to terminate due to pointer analysis imprecisions, whereas Dash is able to terminate in seconds.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

In his 1972 Turing Lecture titled "The Humble Programmer" Edsger W. Dijkstra said, "Program testing is a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence" [9]. While Dijkstra's statement holds if we consider program testing as a black-box activity, tests can indeed be used to progressively guide the construction of proofs if we are allowed to instrument the program and inspect the states that a program goes through during testing.

Over the past few years, there has been dramatic progress in using light-weight symbolic execution [13, 24] to do automatic test generation. In this paper, we present a new algorithm to show that similar light-weight symbolic execution can also be used to *prove* that programs satisfy safety properties.

We build on the SYNERGY algorithm [14], which simultaneously performs program testing and program abstraction. The tests are an "underapproximation" of the program behavior, and the abstraction is an "overapproximation" of the program. The goal is to either find a test that reaches an error state (in which case we have discovered a true violation of the property), or find an abstraction that is precise enough to show that no path in the state space of the program can reach any error state (in which case we have proved that the program satisfies the desired safety property). 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.

Our new algorithm, DASH, makes three significant advances over SYNERGY. First, DASH uses test generation not only to guide *where* to perform the refinement of the abstraction, but also to decide *how* the abstraction should be refined. Unlike the SYNERGY algorithm, there are *no* extra theorem prover calls in the DASH algorithm to maintain the abstraction. The theorem prover is used *only* to do test generation, and refinement is done as a byproduct of a failed test generation attempt. Second, the DASH algorithm handles programs with pointers without using any whole-program may-alias analysis (the SYNERGY algorithm does not handle programs with pointers). DASH refines the abstraction in a sound manner using only aliasing relationships that actually arise in some test. Finally, the DASH algorithm is an interprocedural algorithm, and it uses recursive invocations of itself to handle procedure calls (the SYNERGY algorithm does not handle procedure calls).

Current approaches to prove properties of programs with pointers use a whole program "may-alias" to conservatively reason about the abstraction due to pointer aliases (see Section 4.2 in [3], and Section 6 in [17]). The alias analysis needs to be flow sensitive, field sensitive, and even path sensitive, to be strong enough to prove certain properties (see examples in Section 2), and scalable pointer analyses with these precision requirements do not exist. In addition, there are situations, such as analyzing x86 binaries directly, where global alias information is difficult to obtain. The DASH algorithm uses a different technique to perform refinement without using may-alias information. We define a new

operator WP_α that combines the usual weakest precondition operator [10] with an alias set α. The alias set α is obtained during execution of the specific test that the algorithm is attempting to extend. The predicate obtained from the WP_α operator is weaker than applying the strongest postcondition on the test, and it is stronger than the predicate obtained by applying the usual weakest precondition operator. If the test generation fails, we show that the predicate WP_α can be used to refine the abstraction in a sound manner, without using any extra theorem prover calls (see Section 4.2.1). This has the effect of analyzing only the alias possibilities that actually occur during concrete executions without resorting to a global (and necessarily imprecise) alias analysis that reasons about all executions. Consequently, in many cases, we can show that Dash produces abstractions that are exponentially smaller than those considered by Slam [4] and Blast [17].

2 Overview

Over the past few years, several tools based on predicate abstraction and counterexample-guided abstraction refinement, such as Slam [4] and Blast [17], have been built in order to compute proofs of programs for various properties. The algorithms implemented in these tools have two main bottlenecks. First, they entail several expensive calls to a theorem prover, which adversely impacts scalability. Second, they use global may-alias information, which is typically imprecise and impacts the ability of these tools to prove properties that involve complex aliasing. There has also been dramatic progress in testing techniques like Dart and Cute using light-weight symbolic execution [13, 24]. These testing tools focus on finding errors in programs by way of explicit path model checking and are unable to compute proofs. Our work can be viewed as combining the successful ideas from proof-based tools like Slam and Blast with testing-based tools like Dart and Cute with the goal of improving scalability.

The input to the Dash algorithm consists of a program P with an infinite state space Σ and a set of error states φ. Dash maintains two data structures. First, it maintains the collection of tests as a forest F. Each path in the forest F corresponds to a concrete execution of the program. The algorithm grows F by adding new tests, and as soon as an error state is added to F, a real error has been found. Second, it maintains a finite relational abstraction $\Sigma_\simeq$ of the infinite state space Σ. The states of the abstraction, called *regions*, are equivalence classes of concrete program states from Σ. There is an abstract transition from region S to region S' if there are two concrete states $s \in S$ and $s' \in S'$ such that there is a concrete transition from s to s'. Thus, the abstraction maintains an overapproximation of all concrete executions. In particular, if there is no path of abstract transitions from the initial region to the error region φ, we can be sure that there is no path of concrete transitions that lead from some concrete initial state to some concrete error state $s \in \varphi$, and a proof of correctness has been found.

Each iteration of Dash attempts to make progress by either growing the

```
struct DE {
 int lock;
 int y;
};
```

```
void prove-me1(DE *p, DE *p1, DE *p2)
{
0:  p1 = malloc(sizeof(DE)); p1->lock = 0;
1:  p2 = malloc(sizeof(DE)); p2->lock = 0;
2:  p->lock = 1;
3:  if (p1->lock == 1 || p2->lock == 1)
4:    error();
5:  p = p1;
6:  p = p2;
}
```

Figure 1: Example where SLAM has difficulty without a flow-sensitive alias analysis.

forest F to make it closer to reaching a concrete error state or refining the abstraction $\Sigma_{\simeq}$ (by splitting the regions) to make it closer to a proof of correctness. Each iteration starts with an (abstract) error path τ_e from the initial region to the error region with a prefix τ such that (1) τ corresponds to a concrete path in F and (2) no region in τ_e after the prefix τ is visited in F. If some abstract error path exists, such an "ordered" path τ_e can be always shown to exist. DASH now tries to find a new test which follows the ordered path τ_e for at least one transition past the prefix τ. It uses directed testing [13, 24] to generate such a test. These techniques perform a light-weight symbolic execution along the path τ_e, and collect constraints at every state as functions of the inputs to the program. In programs with pointers, the symbolic execution along τ_e is done in a "pointer-aware" manner keeping track of the aliases between variables in the program. If the generated constraints are unsatisfiable, the test generation fails. A key insight in the DASH algorithm is that if the test generation attempt to extend the forest F beyond the prefix τ fails, then the alias conditions α, obtained by the symbolic execution up to the prefix τ, can be used to refine the abstraction $\Sigma_{\simeq}$. This refinement does not make any theorem prover calls, and does not use a global alias analysis. We define a new operator WP_α to perform such a refinement. The WP_α operator specializes the weakest precondition operator using only the alias conditions α that occur along the test up to the prefix τ. Using the predicate generated by the WP_α operator, we can refine the region at the end of the prefix τ and remove the abstract transition from the prefix τ along the ordered trace τ_e. This technique, which we call *template-based refinement*, is described in Figure 11. The DASH algorithm continues by choosing a new ordered error path until either F finds a concrete execution that reaches the error φ or the refined abstraction $\Sigma_{\simeq}$ provides a proof of correctness that φ can never be reached. Since the problem is undecidable in general, it is possible that DASH does not terminate.

Example 1. Consider the program in Figure 1. This program has three inputs `p`, `p1` and `p2`, all of which are pointers to structs of type `DE` (with two fields `DE->lock` and `DE->y`). At lines `1` and `2`, pointers `p1` and `p2` are pointed to newly allocated memory, and `p1->lock` and `p2->lock` are both set to `0`. Thus, the assignment to `p->lock` at line `3` cannot affect the values of `p1->lock` or `p2->lock`, and the error statement at line `4` can never be reached. The interest-

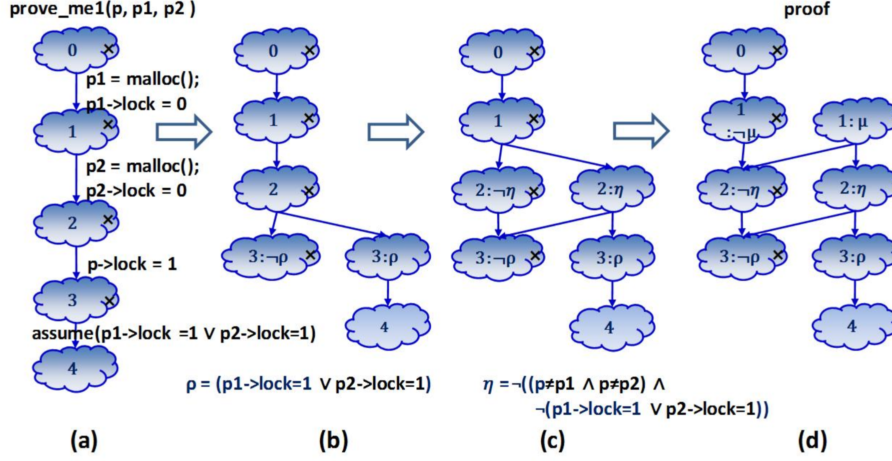

Figure 2: Abstraction computed by DASH on the example program from Figure 1.

ing feature of this example is that `p` may alias with `p1` or `p2` due to assignments at lines `5` and `6`. Thus, a flow-insensitive may-alias analysis will have to conservatively assume that at the assignment at line `2`, the variable `p` may alias with `p1` or `p2`, and consider all possible alias combinations. However, as we describe below, DASH is able to prove this program correct while only ever considering the alias combination $(p \neq p1 \wedge p \neq p2)$ that occurs along concrete executions.

DASH first creates the initial abstraction $\Sigma_{\simeq}$ for the program `prove-me1` and this is isomorphic to its control flow graph (shown in Figure 2(a)). We show regions of the abstraction $\Sigma_{\simeq}$ as "clouds," and states from the forest F using "$\times$"s in the figure. Next, the initial forest is created by running `prove-me1` with a random test that assigns values to its inputs `p`, `p1` and `p2`, thus creating a forest $F_{\texttt{prove-me1}}$ of concrete states. Since running this test did not result in the error location being reached (there is no $\times$ representing a concrete state in the error region 4), DASH examines an (abstract) error path $\tau_e = \langle 0, 1, 2, 3, 4 \rangle$ that leads to the error region. It also considers the prefix $\tau = \langle 0, 1, 2, 3 \rangle$ of τ_e that has concrete states visited in the region from the forest F, as shown in Figure 2(a). DASH now tries to add a test to $F_{\texttt{prove-me1}}$ that follows τ_e for at least one transition beyond the prefix τ by using directed testing [13, 24], that is, a test that covers the transition $(3, 4)$. However, the constraints obtained by symbolic execution of the prefix τ together with the transition $(3, 4)$ are unsatisfiable, so such a test does not exist. DASH now refines the region 3 using the operator WP_α (defined in Section 4.2.1). In this case, the WP_α operator returns the predicate $\rho = (\texttt{p1->lock=1}) \vee (\texttt{p2->lock=1})$. DASH splits the region 3 into two regions, $3{:}\rho$ and $3{:}\neg\rho$. Due to the properties of the WP_α operator, we can now refine the current abstraction at the frontier

```
void foo(DE *p, DE *p1, DE *p2, ... , DE *pn)
{
1:  p1 = malloc(sizeof(DE)); p1->lock = 0;
    p2 = malloc(sizeof(DE)); p2->lock = 0;
    ...
    pn = malloc(sizeof(DE)); pn->lock = 0;
3:  p->lock = 1;
6:  if (p1->lock==1 || ... || pn->lock==1)
7:    error();
8:  p = p1;
    p = p2;
    ...
    p = pn;
}
```

Figure 3: Example for which SLAM suffers from an exponential blow-up in predicate size.

(which is the region 3) according to the template described in Figure 11. This refinement can be done without any theorem prover calls. It involves deleting the edges $(2, 3)$, $(3, 4)$ and adding the edges $(2, 3\!:\!\neg\rho)$, $(2, 3\!:\!\rho)$ and $(3\!:\!\rho, 4)$, resulting in the refined abstraction shown in Figure 2(b). Next, DASH chooses a new abstract error path $\tau_e = \langle 0, 1, 2, 3{:}\rho, 4\rangle$ with prefix $\tau = \langle 0, 1, 2\rangle$ and tries to drive a test along the transition $(2, 3{:}\rho)$. Since the generated constraints are unsatisfiable, this is also not possible. DASH uses the WP_α operator to obtain the predicate $\eta = \neg(\texttt{(p}\neq\texttt{p1}\wedge\texttt{p}\neq\texttt{p2)}\wedge\neg\texttt{(p1->lock=1}\vee\texttt{p2->lock=1))}$. Intuitively, the subexpression $\alpha = \texttt{(p}\neq\texttt{p1}\wedge\texttt{p}\neq\texttt{p2)}$ corresponds to the alias relations that hold between the variables `p`, `p1`, and `p2` after the program executes the path τ. The subexpression $\texttt{p1->lock=1}\vee\texttt{p2->lock=1}$ is the weakest precondition along the transition $(2, 3{:}\rho)$ assuming the aliasing constraints imposed by α. Again, the region 2 can be refined by applying the template from Figure 11 without using any additional theorem prover calls, resulting in the refined abstraction shown in Figure 2(c). DASH continues by choosing a new abstract error path, and eventually produces the abstraction shown in Figure 2(d). Since there is no path in this abstraction from region 0 to region 4, this is a proof that the error at line 4 cannot be reached in this program for any input.

Tools like SLAM [4] and BLAST [17], which use Morris' general axiom of assignment [20] to handle pointer aliases soundly, have to consider 4 possible aliasing conditions: $p = p1$ or $p \neq p1$, and $p = p2$ or $p \neq p2$. Instead, DASH considers *only* the alias possibility $(p \neq p1 \wedge p \neq p2)$ that occurs along the concrete execution of the test, resulting in exponential savings in the size of the proof of correctness.

More formally, it can be easily shown that there is an exponential blow-up in the predicates computed by SLAM for the class of programs defined by the program shown in Figure 3 parameterized by n (we have verified this by running SLAM and measuring run times as a function of n, as seen in Figure 18), whereas DASH does not encounter this blowup since it uses alias information from tests to reason only about the alias combinations that actually happen.

Example 2. Consider the example shown in Figure 4. This program has a

```
struct DE {
 int lock;
 int y;
};

void Inc(DE *de}
{
  de->y++;
}
```

```
void prove-me2(DE *de, int x)
{
0:   de->lock = 0;
1:   do {
2:     assert(de->lock == 0);
3:     de->lock = 1;
4:     x = de->y;
5:     if (*) {
6:       de->lock = 0;
7:       Inc(de);
       }
8:   } while (x != de->y);
}
```

Figure 4: An example illustrating the effect of weak pointer analysis on SLAM.

```
void prove-me2(bool b1, bool b2)

0: b1 = true;            //de->lock = 0
1: do {
2:   assert(b1);         // assert(de->lock==0)
3:   b1 = false;         // de->lock = 1
4:   b2 = true;          // x = de->y
5:   if (*) {
6:     b1 = true;        // de->lock = 0
7:     b2 = b2?false:*;  // effect of Inc on b2
       b1 = *;           // effect of Inc on b1
     }
8: } while (!b2)
}
```

Figure 5: Boolean program abstraction for the example in Figure 4. The boolean variable `b1` represents the predicate `(de->lock==0)` and `b2` represents the predicate `(x==de->y)`.

struct of type `DE` with two fields `DE->lock` and `DE->y`, an input pointer `de` to a struct of type `DE`, and an integer local variable `x`. The loop invariant at line 8 of the program is given by (`de->lock==0` $\wedge$ `x!=de->y`) $\vee$ (`de->lock!=0` $\wedge$ `x==de->y`). We need to prove that the assertion `assert(de->lock==0)` at line 2 always holds. If we run SLAM [4] on this example, we get the boolean program shown in Figure 5 with two boolean variables —`b1` representing the predicate `(de->lock==0)` and `b2` representing the predicate `(x==de->y)`. Each line in the boolean program conservatively abstracts how the statements of the original C program can affect the predicates. For instance, line 3 sets `de->lock` to 1, so line 3 in the boolean program sets `b1` to `false`. Note how the boolean program models the effect of function call `Inc(de)` in line 7. The effect on `b2` represented by the assignment `b2 = b2?false:*` models the fact that `de->y` is incremented. Thus, if `b2` is `true` before the call, then it is `false` after the call, otherwise, its value is nondeterministic (denoted by `*`). Since alias (and consequently mod-ref) analysis used by SLAM is field-insensitive, even though the procedure `Inc` modifies only `de->y` the field-insensitive mod-ref analysis needs to conservatively assume that `de->lock` could be updated as well. As a result, SLAM sets `b1 = *` at line 7 of the boolean program. As a consequence,

```
void prove-me3(int lock, bool b1, bool b2,...,bool bn)
int x;
int *p;
0: *p = malloc(int);x=0;*p=0;
1: lock = 1;
2: if(b1)
3:     x = x + 1;     //does not affect lock
   else
4:     *p = *p + 1;   //does not affect lock
5:...
6:...
7: if(bn)
8:     x = x + 1;      //does not affect lock
   else
9:     *p = *p + 1;    //does not affect lock
10:assert(lock == 1);
}
```

Figure 6: Example to illustrate DASH avoiding an exploration of an exponential number of paths.

the false error path `0,1,2,3,4,5,6,7,8,1,2` leading to the assertion violation still exists in the abstract state space, and SLAM is unable to prove the example correct.

Automatic testing tools like DART or CUTE can show that specific paths in Figure 4 do not violate the assertion in line `2`, since they can precisely track aliases along these paths. However, since the program has a loop (and consequently, an infinite number of paths) these tools will not be able to explore all the paths.

Since DASH uses abstractions as well as aliasing information from concrete tests, it is able to prove this program correct. DASH models the loop in the program as a cycle in the abstraction. During refinements of the abstraction, DASH uses only the aliasing information that occurs on concrete paths executed by the tests. Since the concrete executions of `Inc(de)` do not change `de->lock`, the refinements result in an abstraction that has two properties. First, the abstraction has a cycle to model an unbounded number of executions of the loop. Second, there is no path from the initial region to the error region in the abstraction, and hence the abstraction is a proof that the error region can never be reached.

Example 3. Figure 6 shows an example with 2^n paths due to n conditionals. We wish to check if the program passes the assertion in line `10` along all these paths. Even though DASH uses test case generation to do refinement, because it maintains an abstraction, it avoids exploring all the 2^n different paths. This is because, in each iteration, the DASH algorithm splits some region using the predicate $\rho =$`(lock!=1)`. After $O(n)$ splits, the algorithm generates the refinement shown in the Figure 7. Tools like SLAM and BLAST can prove the above program correct in $O(1)$ iterations, since they first discover the predicate `(lock!=1)` and then use a pointer analysis to first establish that none of the code in the conditionals affects this predicate. As a result, the generated boolean program abstracts all the code in the conditionals using `skip` statements. While

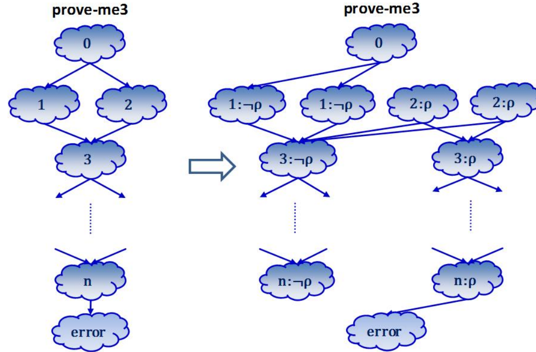

Figure 7: Initial abstraction and proof computed by Dash on the example program from Figure 6.

such an optimization can be implemented to reduce the number of iterations taken by Dash from $O(n)$ iterations to $O(1)$ iterations, we have not yet implemented any such optimizations. Our empirical results show that even without such optimizations, Dash performs very well, since the cost of each iteration is very low. In contrast to Dash or Slam, testing tools like Dart and Cute need to explore 2^n paths since they do not have the benefit of an abstraction to guide them.

Interprocedural Property Checking. For programs with several procedures, we describe a modular approach to generalize Dash. First, the notion of forests and abstractions can be easily extended to programs with multiple procedures by maintaining a separate forest F_P and a separate abstraction $\Sigma_{\simeq_P}$ for every procedure P. The only case in the Dash algorithm that needs to be generalized is when the frontier we are trying to extend happens to be a procedure call-return edge (S, S'). In such a case, Dash simply invokes itself recursively on the called procedure by appropriately translating the constraint induced by the path τ (the prefix of the abstract error path τ_e) into appropriate initial states of the called procedure and translating the predicate on the target region S' into appropriate error states of the called procedure.

We explain this through the example in Figure 8, where procedure `top` makes two calls to an increment procedure `inc`. We show how Dash proves that the call to `error()` (statement 4 in `top`) is unreachable.

Dash first creates the initial abstractions $\Sigma_{\simeq_{\texttt{top}}}$ and $\Sigma_{\simeq_{\texttt{inc}}}$ for the procedures `top` and `inc` respectively (shown in Figure 9(a)). Note that these initial abstractions are isomorphic to the control flow graphs of their respective procedures. Next, the initial forests are created by running a random test (with test input $x = 2$) for `top`, thus creating a forest of concrete states (assume that every concrete state $\times$ is connected to its parent within a procedure) for each procedure (Figure 9(a)). Since running the test did not result in the error location being reached (there is no concrete state $\times$ in the the error state 3), Dash examines

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

Figure 8: A simple example for interprocedural property checking.

an (abstract) error path $\tau_e = \langle 0, 1, 2, 3\rangle$ with prefix $\tau = \langle 0, 1, 2\rangle$ in $\Sigma_{\simeq_{\texttt{top}}}$ (shown in Figure 9(a)). DASH now tries to add a test to $F_{\texttt{top}}$ that follows τ_e for at least one transition beyond the prefix τ by using directed testing [13, 24], that is, a test that covers the transition $(2, 3)$. It turns out that such a test is not possible and therefore DASH refines the abstraction $\Sigma_{\simeq_{\texttt{top}}}$ by removing the abstract transition $(2, 3)$. This is done using the WP_α operator that returns the predicate $\rho = (b \neq x + 2)$. Then, applying the template from Figure 11, DASH refines the region 2 to two regions – $2{:}\rho$ and $2{:}\neg\rho$, and we obtain the abstraction shown in Figure 9(b).

Next, DASH continues by choosing a new abstract error path $\tau_e = \langle 0, 1, 2{:}\rho, 3\rangle$ in the procedure `top`, with prefix $\tau = \langle 0, 1\rangle$. Since the abstract transition $(1, 2{:}\rho)$ that is to be tested now corresponds to a call to the procedure `inc`, we make a recursive call to DASH on the procedure `inc`. This call to DASH checks whether a test can be run on `inc` with a precondition induced by τ and postcondition induced by the region $2{:}\rho$ in `top`. It turns out that this recursive call to DASH returns a "fail" indicating that such a test is not feasible, and this results in a refinement of the abstract region 2 with respect to the predicate η (shown in Figure 9(c)). We discuss how η can be computed in Section 4.4. After some more iterations, DASH computes the abstraction $\Sigma_{\simeq_{\texttt{top}}}$ (shown in Figure 9(d)) that proves that the error location is unreachable in the procedure `top`.

3 Related work

Several papers have predicted that testing and verification can be combined in deep ways [12, 15]. Yorsh, Ball and Sagiv have proposed an approach that involves both abstraction and testing [25]. Their approach examines abstract counterexamples and fabricates new concrete states along them as a heuristic to increase the coverage of testing. They can also detect when the current program abstraction is a proof. Unlike DASH, they do not have a refinement algorithm. Kroening, Groce and Clarke describe a technique to perform abstraction refinement using concrete program execution [18]. Their refinement algorithm is based on partial program simulation using SAT solvers. In contrast, DASH uses tests to choose the frontiers of abstract counterexamples, and tries to either extend or refine each frontier with exactly one theorem prover call. The SYNERGY algorithm [14] also combines testing and abstraction refinement based verifica-

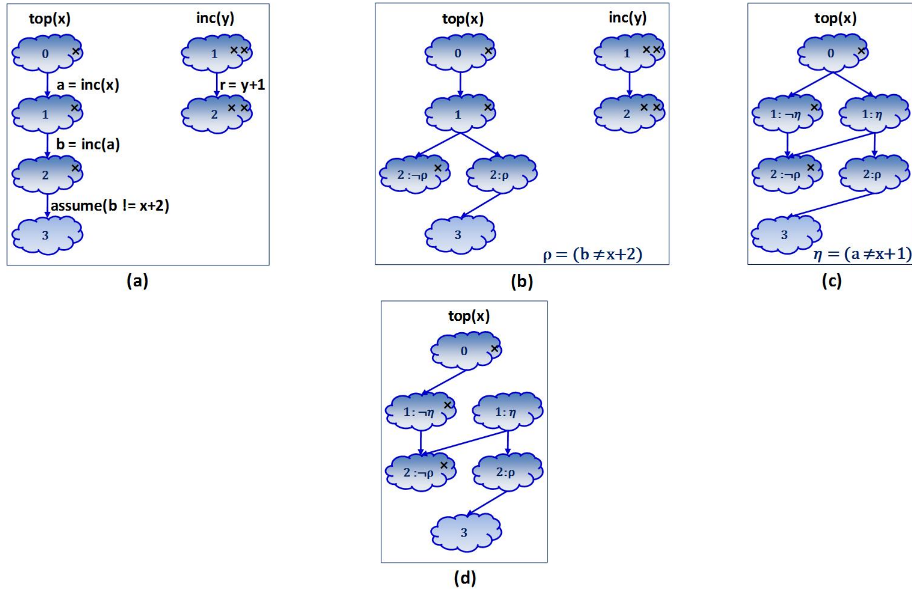

Figure 9: Abstractions computed by DASH on the example program from Figure 8.

tion algorithms in a novel way. SYNERGY uses tests to decide where to refine the abstraction and makes theorem prover calls to maintain the abstraction. We have compared DASH with SYNERGY in Section 1.

Verification tools such as SLAM employ an interprocedural dataflow engine to analyze programs with multiple procedures [5]. This involves computing abstract summaries for every procedure in the program. Recently, interprocedural extensions to testing tools like DART [11] and CUTE [19] have been proposed, and these compute concrete summaries (tests) for every procedure in the program. DASH is a modular interprocedural analysis algorithm that combines testing and abstraction. Intuitively, DASH analyzes called functions using path-sensitive information from the caller, and the result of this analysis is fed back to the caller in the form of both concrete as well as abstract summaries (though we do not describe them as summaries in the description of the algorithm). DASH currently does not reuse summaries computed in one context in a different context. We plan to address this in future work.

Several methods for doing refinement have been proposed, including backward propagation from error states [6], forward propagation from initial states [4], and using interpolants [16]. In all these cases, a theorem prover call is required at every step of the trace to refine the abstraction, and a global may-alias analysis is needed to maintain the refined abstraction. In addition, several theorem prover calls are used to maintain the abstraction after doing the refinement. In contrast, DASH is built primarily around test generation. In the event of a failed

test generation, DASH has enough information to know that the frontier between the regions covered by tests and the regions not covered by tests *is* a suitable refinement point without having to do any further theorem prover calls. As we show in Theorem 3, Section 4.2.1, we can use the operator WP_α to compute a refinement at the frontier that is guaranteed to make progress without making any extra theorem prover calls and without using any global may-alias information. Thus, every iteration of DASH is considerably more efficient; its efficiency is comparable to that of test generation tools such as CUTE and DART. The price we pay is that DASH may have to perform more iterations, since the discovered predicate is lazily propagated backward one step at a time through only those regions which are discovered to be relevant; therefore, several iterations of DASH are comparable to a single iteration of a tool like SLAM. However, as our empirical results show, this tradeoff works very well in practice.

Namjoshi and Kurshan [21] have proposed doing refinements without using theorem provers, using the weakest precondition operator. However, their scheme does not use tests to identify the point where refinement needs to be done. Unlike DASH, their work does not handle pointers or aliasing.

4 Algorithm

We will consider C programs and assume that they have been transformed to a simple intermediate form where:

(a) All statements are labeled with a *program location*.

(b) All *expressions* are side-effect free and do not contain multiple dereferences of pointers (e.g., $(*)^{k>1}p$).

(c) *Intraprocedural control flow* is modeled with `if (e) goto l` statements, where e is an expression and l is a program location.

(d) All *assignments* are of the form `*m = e`, where m is a memory location and e is an expression.

(e) All *function calls* (call-by-value function calls) are of the form `*m = f(x1,x2,...,xn)`, where m is a memory location.

Though our presentation considers only pointer dereferences of the form `*p`, our implementation also supports structs with fields, and pointers to structs with dereferences of the form `p->f`.

Syntax. Let *Stmts* be the set of valid statements in the simple intermediate form. Formally, a program $\mathcal{P}$ is given by a recursive state machine (RSM) [2] $\langle P_0, P_1, \ldots, P_n \rangle$, where each *component procedure* $P_i = \langle N_i, L_i, E_i, n_i^0, \lambda_i, V_i \rangle$ is defined by the following.

- A finite set N_i of nodes, each uniquely identified by a program location from the finite set L_i of program locations.

- A set of control flow edges $E_i \subseteq N_i \times N_i$.
- A special start node $n_i^0 \in N_i$ which represents the procedure's entry location.
- A labeling $\lambda_i : E_i \rightarrow Stmts$, that labels each edge with a statement in the program. If $\lambda_i(e)$ is a function call, then we will refer to the edge e as a *call-return* edge. We will denote the set of all call-return edges in E_i by $\mathsf{CallRet}(E_i)$.
- A set V_i of variables (consisting of parameters, local variables and global variables) that are visible to the procedure P_i. We will assume that all lvalues and expressions are of type either pointer or integer. Additionally, V_i will contain a special variable pc_i which takes values from L_i.

We will refer to the procedure P_0 as the `main` procedure, and this is where the execution of the program $\mathcal{P}$ begins.

Semantics. It suffices to consider only the *data state* of a procedure $P = \langle N, L, E, n_0, \lambda, V \rangle$ for our purpose. Let Σ be the (possibly infinite) state space of P, defined as the set of all valuations to the variables in V. Every statement $op \in Stmts$ defines a state transition relation $\overset{op}{\rightarrow}: \Sigma \times \Sigma$, and this naturally induces a transition relation $\rightarrow : \Sigma \times \Sigma$ for the procedure P. Let $\sigma^I \subseteq \Sigma$ denote the set of initial states of P. We use $\overset{*}{\rightarrow}$ to denote the reflexive and transitive closure of the transition relation $\rightarrow$. A property $\varphi \subseteq \Sigma$ is a set of bad states that we do not want the program to reach. An instance of the property checking problem is a pair (P, φ). The answer to (P, φ) is "fail" if there is some initial state $s \in \sigma^I$ and some error state $s' \in \varphi$ such that $s \overset{*}{\rightarrow} s'$, and "pass" otherwise.

Our objective is to produce certificates for both "fail" and "pass" answers. A certificate for "fail" is an *error trace*, that is, a finite sequence $s_0, s_1, \ldots, s_n$ of states such that: (1) $s_0 \in \sigma^I$, (2) $s_i \rightarrow s_{i+1}$ for $0 \leq i < n$, and (3) $s_n \in \varphi$.

A certificate for "pass" is a finite-indexed partition $\Sigma_\simeq$ of the state space Σ which proves the absence of error traces. We refer to the equivalence classes of the partition $\Sigma_\simeq$ as *regions*. The partition $\Sigma_\simeq$ induces an abstract procedure $P_\simeq = \langle \Sigma_\simeq, {\sigma^I}_\simeq, \rightarrow_\simeq \rangle$, where ${\sigma^I}_\simeq = \{S \in \Sigma_\simeq \mid S \cap \sigma^I \neq \emptyset\}$ is the set of regions that contain initial states, and $S \rightarrow_\simeq S'$ for $S, S' \in \Sigma_\simeq$ if there exist two states $s \in S$ and $s' \in S'$ such that $s \rightarrow s'$. We allow for the possibility that $S \rightarrow_\simeq S'$ when there do not exist states $s \in S$ and $s' \in S'$ such that $s \rightarrow s'$.

Let $\varphi_\simeq = \{S \in \Sigma_\simeq \mid S \cap \varphi \neq \emptyset\}$ denote the regions in $\Sigma_\simeq$ that intersect with φ. An *abstract error trace* is a sequence $S_0, S_1, \ldots, S_n$ of regions such that: (1) $S_0 \in {\sigma^I}_\simeq$, (2) $S_i \rightarrow_\simeq S_{i+1}$ for all $0 \leq i < n$, and (3) $S_n \in \varphi_\simeq$.

The finite-indexed partition $\Sigma_\simeq$ is a *proof* that the procedure P cannot reach the error φ if there is no abstract error trace in $P_\simeq$.

4.1 The Dash Algorithm

We will first assume that the program $\mathcal{P} = \langle P \rangle$ has one procedure P, and discuss how we handle programs with multiple procedures in Section 4.4. The algorithm

$\textsc{Dash}(P = \langle \Sigma, \sigma^I, \rightarrow \rangle,\ \varphi)$
Returns:
("fail", t), where t is an error trace of P reaching φ; or
("pass", $\Sigma_\simeq$), where $\Sigma_\simeq$ is a proof that P cannot reach φ.

```
 1: Σ≃ := ⋃_{l∈L} {{(pc, v) ∈ Σ | pc = l}}
 2: σ^I≃ := {S ∈ Σ≃ | pc(S) is the initial pc}
 3: →≃ := {(S, S') ∈ Σ≃ × Σ≃ | Edge(S, S') ∈ E}
 4: P≃ := ⟨Σ≃, σ^I≃, →≃⟩
 5: F := Test(P)
 6: loop
 7:   if φ ∩ F ≠ ∅ then
 8:     choose s ∈ φ ∩ F
 9:     t := TestForWitness(s)
10:     return ("fail", t)
11:   end if
12:   τ := GetAbstractTrace(P≃, φ)
13:   if τ = ε then
14:     return ("pass", Σ≃)
15:   else
16:     τ_o := GetOrderedAbstractTrace(τ, F)
17:     ⟨t, ρ⟩ := ExtendFrontier(τ_o, F, P)
18:     if ρ = true then
19:       F := AddTestToForest(t, F)
20:     else
21:       let S_0, S_1, ..., S_n = τ_o and
22:       (k−1, k) = Frontier(τ_o) in
23:       Σ≃ := (Σ≃ \ {S_{k−1}}) ∪
24:                {S_{k−1} ∧ ρ, S_{k−1} ∧ ¬ρ}
25:       →≃ := (→≃ \ {(S, S_{k−1}) | S ∈ Parents(S_{k−1})})
26:                 \{(S_{k−1}, S) | S ∈ (Children(S_{k−1}))}
27:       →≃ := →≃ ∪ {(S, S_{k−1} ∧ ρ) | S ∈ Parents(S_{k−1})}∪
28:                {(S, S_{k−1} ∧ ¬ρ) | S ∈ Parents(S_{k−1})}∪
29:                {(S_{k−1} ∧ ρ, S) | S ∈ (Children(S_{k−1}))}∪
30:                {(S_{k−1} ∧ ¬ρ, S) | S∈(Children(S_{k−1}) \ {S_k})}
31:     end if
32:   end if
33: end loop
```

Figure 10: The $\textsc{Dash}$ algorithm.

$\textsc{Dash}$ shown in Figure 10 takes the property checking instance (P, φ) as input and can have three possible outcomes:

(1) It may output "fail" together with a test t that certifies that P can reach φ.

(2) It may output "pass" together with a proof $\Sigma_\simeq$ that certifies that P cannot reach φ.

(3) It may not terminate.

$\textsc{Dash}$ maintains two data structures:

(1) A finite forest F of states where for every state $s \in F$, either $s \notin \sigma^I$ and $parent(s) \in F$ is a concrete predecessor of s (that is, $parent(s) \rightarrow s$), or $s \in \sigma^I$ and $parent(s) = \epsilon$.

(2) A finite-indexed partition $\Sigma_\simeq$ of the state space Σ of P.

The regions of $\Sigma_{\simeq}$ are defined by *pc* values and predicates over program variables. Let $\mathsf{pc}(S)$ denote the program location associated with region S, and let $\mathsf{Edge}(S, S')$ be a function that returns the control flow edge $e \in E$ that connects regions S and S'. Initially (lines 1–4), there is exactly one region for every *pc* in the procedure P; therefore, the abstract procedure $P_{\simeq}$ is initially isomorphic to the control flow graph of the procedure P. The function Test (line 5) tests the procedure P using test inputs for P, and returns the reachable concrete states of P in the form of a forest F (which is empty if no test inputs for P are available). The test inputs for P may come from previous runs of the algorithm, from external test suites, or from automatic test generation tools.

In each iteration of the main loop, the algorithm either expands the forest F to include more reachable states (with the hope that this expansion will help produce a "fail" answer), or refines the partition $\Sigma_{\simeq}$ (with the hope that this refinement will help produce a "pass" answer). The algorithm locates a path from an initial region to the error region through the abstract procedure, and then discovers the boundary (the *frontier*) along this path between regions which are known to be reachable and a region which is not known to be reachable. Directed test generation, similar in spirit to CUTE [24], is then used to expand the forest F with a test that crosses this frontier. If such a test cannot be created, we refine the partition $\Sigma_{\simeq}$ at this "explored" side of the frontier. Thus, abstract error traces are used to direct test generation, and the non-existence of certain kinds of tests is used to guide the refinement of $P_{\simeq}$.

Every iteration of DASH first checks for the existence of a test reaching the error (line 7). If there is such a test, then $\varphi \cap F \neq \emptyset$, so the algorithm chooses a state $s \in \varphi \cap F$ and calls the auxiliary function TestForWitness to compute a concrete test that reaches the error. TestForWitness (line 9) uses the *parent* relation to generate an error trace – it starts with a concrete state s and successively looks up the *parent* until it finds a concrete state s_0 (a root of F) that belongs to an initial region. $\mathsf{TestForWitness}(s)$ returns the state sequence $s_0, s_1, \ldots, s_n$ such that $s_n = s$ and $s_i \rightarrow s_{i+1}$ for all $0 \leq i < n$.

If no test to the error exists in the forest F, the algorithm calls GetAbstractTrace (line 12) to find an abstract error trace τ through the abstract graph. If no such trace exists, then the current partition $\Sigma_{\simeq}$ is a proof that P cannot reach any state in φ, and GetAbstractTrace returns $\tau = \epsilon$. Otherwise, GetAbstractTrace returns the abstract trace $\tau = S_0, S_1, \ldots, S_n$ such that $S_n = \varphi$. The next step is to convert this trace into an ordered abstract trace. An abstract trace $S_0, S_1, \ldots, S_n$ is *ordered* if the following two conditions hold:

(1) There exists a *frontier* $(k{-}1, k) \stackrel{\mathsf{def}}{=} \mathsf{Frontier}(S_0, S_1, \ldots, S_n)$ such that (a) $0 \leq k \leq n$, and (b) $S_i \cap F = \emptyset$ for all $k \leq i \leq n$, and (c) $S_j \cap F \neq \emptyset$ for all $0 \leq j < k$.

(2) There exists a state $s \in S_{k-1} \cap F$ such that $S_i = \mathsf{Region}(parent^{k-1-i}(s))$ for all $0 \leq i < k$, where the abstraction function Region maps each state $s \in \Sigma$ to the region $S \in \Sigma_{\simeq}$ with $s \in S$.

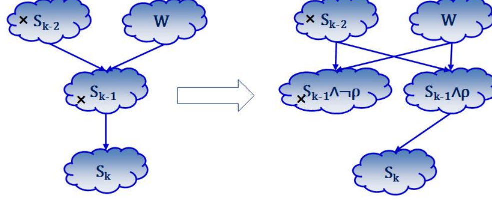

Figure 11: Refinement split performed by DASH at the frontier.

ExtendFrontier(τ, F, P)
Returns:
$\langle t, true\rangle$, if the frontier can be extended; or
$\langle \epsilon, \rho\rangle$, if the frontier cannot be extended.

```
1: (k−1, k) := Frontier(τ)
2: ⟨φ1, S, φ2⟩ := ExecuteSymbolic(τ, F, P)
3: t := IsSAT(φ1, S, φ2, P)
4: if t = ε then
5:     ρ := RefinePred(S, τ)
6: else
7:     ρ := true
8: end if
9: return ⟨t, ρ⟩
```

Figure 12: The auxiliary function ExtendFrontier.

We note that whenever there is an abstract error trace, then there must exist an ordered abstract error trace. The auxiliary function GetOrderedAbstractTrace (line 16) converts an arbitrary abstract trace τ into an ordered abstract trace τ_o. This works by finding the last region in the abstract trace that intersects with the forest F, which we call S_f. The algorithm picks a state in this intersection and follows the *parent* relation back to an initial state. This leads to a concrete trace $s_0, s_1, \ldots, s_{k-1}$ that corresponds to an abstract trace $S_0, S_1, \ldots S_{k-1}$ where $S_{k-1} = S_f$. By splicing together this abstract trace and the portion of the abstract error trace from S_f to S_n, we obtain an ordered abstract error trace. It is crucial that the ordered abstract error trace follows a concrete trace up to the frontier, as this ensures that it is a feasible trace up to that point.

The algorithm now calls the function ExtendFrontier (line 17). The function ExtendFrontier, shown in Figure 12, *is the only function in the* DASH *algorithm that uses a theorem prover*. It takes an ordered trace τ_o, forest F, and procedure P as inputs and returns a pair $\langle t, \rho\rangle$, where t is a test and ρ is a predicate. They can take the following values:

- $\langle t, true\rangle$, when t is a test that extends the frontier. The test t is then added to the forest F by AddTestToForest (line 19), which runs an instrumented version of the program to obtain the trace of concrete states that are added to F.
- $\langle \epsilon, \rho\rangle$, when no test that extends the frontier is possible. In this case, ρ is a suitable refinement predicate that is used to used to refine the partition

```
ExecuteSymbolic(τ, F, P)
Returns: ⟨φ1, S, φ2⟩.
 1: (k−1, k) := Frontier(τ = ⟨S0, S1, . . . , Sn⟩)
 2: S := [v ↦ v0 | ∗v ∈ inputs(P)]
 3: φ1 := SymbolicEval(S0, S)
 4: φ2 := true
 5: i := 0
 6: while i ≠ k−1 do
 7:    op := λ(Edge(Si, Si+1))
 8:    match(op)
 9:        case(∗m = e):
10:            S := S+[SymbolicEval(m, S) ↦ SymbolicEval(e, S)]
11:        case(if e goto l):
12:            φ1 := φ1 ∧ SymbolicEval(e, S)
13:    i := i + 1
14:    φ1 := φ1 ∧ SymbolicEval(Si, S)
15: end while
16: op := λ(Edge(Sk−1, Sk))
17: match(op)
18:     case(∗m = e):
19:         φ2 := φ2∧
20:                 ∗(SymbolicEval(m, S)) = SymbolicEval(e, S)
21:         S′ := S + [SymbolicEval(m, S) ↦ SymbolicEval(e, S)]
22:     case(if e goto l):
23:         φ2 := φ2 ∧ SymbolicEval(e, S)
24:         S′ := S
25: φ2 := φ2 ∧ SymbolicEval(Sk, S′)
26: return ⟨φ1, S, φ2⟩
```

Figure 13: The auxiliary function ExecuteSymbolic.

$\Sigma_{\simeq}$ at the frontier (lines 21–30), resulting in a split of region S_{k-1} (as shown in Figure 11) that eliminates the spurious abstract error trace τ_o.

The function ExecuteSymbolic, which is called at line 2 of ExtendFrontier, performs symbolic execution on τ using techniques inspired by CUTE [24]. Let $\tau = \langle S_0, S_1, \ldots, S_n\rangle$, and let $(k-1, k) = \mathsf{Frontier}(\tau)$. ExecuteSymbolic returns $\langle \phi_1, \mathcal{S}, \phi_2\rangle$, where ϕ_1 and $\mathcal{S}$ are respectively the path constraint and symbolic memory map obtained by performing symbolic execution on the abstract trace $\langle S_0, S_1, \ldots, S_{k-1}\rangle$, and ϕ_2 is the result of performing symbolic execution on the abstract trace $\langle S_{k-1}, S_k\rangle$ (not including the region S_{k-1}) starting with the symbolic memory map $\mathcal{S}$. ExecuteSymbolic is described in Figure 13. It first initializes the symbolic memory map $\mathcal{S}$ with $v \mapsto v_0$ for every input variable $*v$ in the program, where v_0 is the initial symbolic value for $*v$ (line 2 in Figure 13) and performs symbolic execution in order to compute ϕ_1 and ϕ_2. The function SymbolicEval$(e, \mathcal{S})$ evaluates the expression e with respect to values from the symbolic memory $\mathcal{S}$.

ExtendFrontier calls the function IsSAT (line 3 in Figure 12) that checks whether $\mu = \phi_1 \wedge \mathcal{S} \wedge \phi_2$ is satisfiable[1] by making a call to a theorem prover. If μ is satisfiable, IsSAT uses the satisfying assignment/model to generate a test t for P that extends the frontier, otherwise it sets $t = \epsilon$. If it is not possible to extend the frontier (that is, $t = \epsilon$ as shown in line 4), then ExtendFrontier calls RefinePred (line 5) which returns a predicate ρ that is a suitable candidate for

[1]Every entry in $\mathcal{S}$ is looked upon as an equality predicate here.

refining $\Sigma_{\simeq}$ at S_{k-1} according to the template in Figure 11. It is useful to note that RefinePred makes no theorem prover calls in order to compute ρ.

4.2 Suitable Predicates

If we cannot drive a test-case past the frontier, then RefinePred should return a predicate that is in some sense "good." If we examine Figure 11, there are definitely two ways in which a refinement predicate can be bad. If ρ is too weak, then it will be possible to derive a test along the same ordered abstract trace, in which case RefinePred will be called with the exact same arguments and will return ρ again. Alternatively, if ρ is too strong, then there may be a transition from some region in $S_{k-1} \wedge \neg p$ to some region in S_k, and we will not be justified in removing the edge between these two regions. By formalizing the notion of a *suitable predicate*, we can show that any suitable predicate will allow DASH to make progress in a sound manner and also that the predicate returned by RefinePred is a suitable predicate.

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

Given two abstract error traces $\tau = \langle S_0, S_1, \ldots, S_n\rangle$ and $\tau' = \langle T_0, T_1, \ldots, T_n\rangle$ of the same length, we say that $\tau \sqsubset \tau'$ if either of the following conditions is true.

(a) $\forall_{0 \le i \le n} T_i \subseteq S_i$, and $\exists k \in [0, n]$ such that $T_k \subset S_k$.

(b) Let $(x, x+1) = \mathsf{Frontier}(\tau)$ and $(y, y+1) = \mathsf{Frontier}(\tau')$, then $\forall_{0 \le i \le n} T_i = S_i$, and $y > x$.

Essentially, this means that $\tau \sqsubset \tau'$ if τ' is a strictly "better" trace, either because the frontier in τ' has been pushed forward or because at least one region in τ' holds strictly fewer states. This is formalized by Definition 2:

Definition 2 (Progress) *Let $\Gamma = \langle \tau_0, \tau_1, \ldots\rangle$ be a sequence of abstract error traces examined by* DASH. *Then we say that* DASH *makes progress if there do not exist i and j such that $i < j$ and $\tau_j \sqsubset \tau_i$.*

Theorem 1 *If a suitable predicate for an abstract error trace τ is used to perform refinement, then the* DASH *algorithm makes progress.*

Proof: Let $\tau = \langle S_0, S_1, \ldots, S_n\rangle$. By definition (see Figure 11), it follows that a suitable predicate ρ with respect to τ would eliminate the edge (S_{k-1}, S_k) in a sound manner by splitting S_{k-1} into two regions, $S_{k-1} \wedge \rho$ and $S_k \wedge \rho$. Since all concrete states in S_{k-1} that can be obtained by traversing the abstract error trace belong to the region $S_{k-1} \wedge \neg\rho$, and the edge $(S_{k-1} \wedge \rho, S_k)$ does not exist,

RefinePred($\mathcal{S}$, ϕ_2, τ)
Returns: a suitable predicate ρ.
1: $(k-1, k)$:= Frontier($\tau = \langle S_0, S_1, \ldots, S_m \rangle$)
2: op := λ(Edge(S_{k-1}, S_k))
3: α := Aliases($\mathcal{S}, op, S_k$)
4: return $\mathsf{WP}_\alpha(op, S_k)$

Figure 14: Computing suitable predicates.

it follows that Definition 2 is satisfied if a refinement is performed on any of the states. Alternatively, if a test is generated, then the second condition in Definition 2 will be satisfied, thus proving the theorem. ■

Corollary 2 *A suitable predicate ensures that the refinement is sound.*

Theorem 1 allows us to perform *template-based refinement* (as shown in Figure 11) without any calls to a theorem prover after computing a suitable predicate. We will next describe how the auxiliary function RefinePred computes a suitable predicate.

4.2.1 Computing Suitable Predicates

For a statement $op \in Stmts$ and a predicate ϕ, let $\mathsf{WP}(op, \phi)$ denote the *weakest precondition* [10] of ϕ with respect to statement *op*. $\mathsf{WP}(op, \phi)$ is defined as the weakest predicate whose truth before *op* implies the truth of ϕ after *op* executes. The weakest precondition $\mathsf{WP}(\mathtt{x} = \mathtt{e}, \phi)$ is the predicate obtained by replacing all occurrences of x in ϕ (denoted $\phi[e/x]$). For example, $\mathsf{WP}(\mathtt{x} = \mathtt{x}{+}\mathtt{1}, x{<}1) = (x{+}1){<}1 = (x{<}0)$. However, in the case of pointers, $\mathsf{WP}(op, \phi)$ is not necessarily $\phi[e/x]$. For example, $\mathsf{WP}(\mathtt{x} = \mathtt{x} + \mathtt{1}, *p + *q < 1)$ is not $*p + *q < 1$, if either *p or *q or both alias x. In order to handle this, if the predicate ϕ mentions k locations[2] (say $y_1, y_2, \ldots, y_k$), then $\mathsf{WP}(\mathtt{x} = \mathtt{e}, \phi)$ would have 2^k disjuncts, with each disjunct corresponding to one possible alias condition of the k locations with x [3]. Therefore, $\mathsf{WP}(\mathtt{x} = \mathtt{x} + \mathtt{1}, *p + *q < 1)$ has 4 disjuncts as follows:

$$\begin{array}{ll}
(\&x{=}p \wedge \&x{=}q \wedge 2x{<}{-1}) & \vee \\
(\&x{\neq}p \wedge \&x{=}q \wedge *p + x{<}\ 0) & \vee \\
(\&x{=}p \wedge \&x{\neq}q \wedge x + *q{<}\ 0) & \vee \\
(\&x{\neq}p \wedge \&x{\neq}q \wedge *p + *q{<}\ 1) &
\end{array}$$

Typically, a whole-program may-alias analysis is used to improve the precision (that is, prune the number of disjuncts) of the weakest precondition and the outcome of this analysis largely influences the performance of tools like Slam. However, as motivated by the example in Figure 5, imprecisions in a whole-program may-alias analysis are ineffective in pruning the disjuncts. Dash takes an alternate approach. It considers only the aliasing α that can happen along

[2] A location is either a variable, a structure field access from a location, or a dereference of a location.

the current abstract trace, and computes the weakest precondition specialized to that aliasing condition, as shown by the function RefinePred in Figure 14.

We first define the projection of the weakest precondition with respect to alias condtion α as follows:

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

For efficiency, $\mathsf{WP}{\downarrow_\alpha}(op, \phi))$ can be computed by only considering the alias possibility α. For example, if $\alpha = (\&x \neq p \wedge \&x = q)$ we have that

$$\mathsf{WP}{\downarrow_\alpha}(\mathtt{x} = \mathtt{x} + 1, *p + *q < 1) = (\&x \neq p \wedge \&x = q \wedge x < 0)$$

The refinement predicate computed by RefinePred is

$$\mathsf{WP}_\alpha(op, \phi_2) \stackrel{\mathsf{def}}{=} \neg(\alpha \wedge \neg\mathsf{WP}{\downarrow_\alpha}(op, \phi_2))$$

Next, we show that such a predicate satisfies the conditions for a suitable predicate.

Theorem 3 *The predicate* $\mathsf{WP}_\alpha(op, \phi_2)$ *computed by the auxiliary function* RefinePred *is a suitable predicate.*

Proof: There are two parts of this proof for the two requirements of Definition 1. Let $\mathcal{C}$ be the set of concrete states obtained by executing the ordered trace up to the frontier. Any concrete state $c \in \mathcal{C}$ must satisfy the existing predicate on the region S_{k-1} as well as the alias relations defined by α. It is also not possible to generate a test that extends the frontier (otherwise, the theorem prover call in Line 3 of Figure 16 would succeed, and RefinePred would not be called). Thus, it must be the case that $\forall c \in \mathcal{C},\ c \notin \mathsf{WP}{\downarrow_\alpha}(op, \phi_2)$. This implies that $\forall c \in \mathcal{C},\ c \in (\alpha \wedge \neg\mathsf{WP}{\downarrow_\alpha}(op, \phi_2))$ and hence the predicate $\mathsf{WP}_\alpha(op, \phi_2)$ satisfies the first requirement of Definition 1.

The second part of Definition 1 requires that no state in $S_{k-1} \wedge \neg\mathsf{WP}_\alpha(op, \phi_2)$ have a transition to a state in S_k. Every state that can make this transition satisfies $\mathsf{WP}(op, \phi_2)$ by the definition of weakest precondition. Because every state in $S_{k-1} \wedge \neg\mathsf{WP}_\alpha(op, \phi_2)$ must also satisfy the alias relations defined by α, any state in $S_{k-1} \wedge \neg\mathsf{WP}_\alpha(op, \phi_2)$ that can transition to S_k must satisfy $\mathsf{WP}{\downarrow_\alpha}(op, \phi_2)$ specifically. Because every state satisfying $\neg\mathsf{WP}_\alpha(op, \phi_2)$ also must *not* satisfy $\mathsf{WP}{\downarrow_\alpha}(op, \phi_2)$, no states with a transition to S_k can exist, and therefore $\mathsf{WP}_\alpha(op, \phi_2)$ is a suitable predicate. ■

We note that while WP is another possible choice for a suitable predicate for the refinement shown in Figure 11, the predicate computed by WP contains an exponential number of disjuncts in the presence of aliasing. Thus, the use of WP_α avoids an exponential number of disjuncts when compared to other approaches that use WP such as [14] and [21].

DASH-MAIN($\mathcal{P}$, φ)
Returns:
("fail", t), where t is an error trace of $\mathcal{P}$ reaching φ; or
("pass", $\Sigma_{\simeq}$), where $\Sigma_{\simeq}$ is a proof that $\mathcal{P}$ cannot reach φ.

1: let $\langle P_0, P_1, \ldots, P_n \rangle = \mathcal{P}$ in
2: DASH($P_0 = \langle \Sigma_0, \sigma_0^I, \rightarrow_0 \rangle$, φ)

Figure 15: The DASH algorithm for programs with multiple procedures.

4.3 Soundness and Complexity

DASH is sound in the sense that if DASH terminates on (P, φ), then either of the following is true: (1) if DASH returns ("pass", $\Sigma_{\simeq}$), then $\Sigma_{\simeq}$ is a proof that P cannot reach φ, and (2) if DASH returns ("fail", t), then t is a test for P that violates φ. However, there is no guarantee that DASH will terminate (this is a shortcoming of all tools that use counterexample driven refinement, such as SLAM and BLAST).

Though we cannot bound the number of iterations of DASH we can bound the number of theorem prover calls made in each iteration. During a DASH iteration, a test generation entails one theorem prover call (call to IsSat in line 3 of the auxiliary function ExtendFrontier). If a test that extends the frontier is not possible, then generating a suitable predicate for refinement does not involve a theorem prover call.

4.4 Handling Programs with Procedures

We will assume without loss of generality that the property φ that we wish to check is only associated with the main procedure P_0 in the program $\mathcal{P}$. Therefore, DASH-MAIN($\mathcal{P}$= $\langle P_0, P_1, \ldots, P_n \rangle$, φ) (shown in Figure 15) calls the function DASH from Figure 10 on the property checking instance (P_0, φ). As in the single procedure case, we maintain a forest F and an abstraction $P_{\simeq}$ for every procedure P in the program. The interprocedural analysis differs from the intraprocedural algorithm described earlier only in the definition of the auxiliary function ExtendFrontier. The modified version of ExtendFrontier is shown in Figure 16. Informally, the interprocedural algorithm makes a recursive call to DASH at every frontier that corresponds to a function call in order to figure out whether there exist tests that extend this frontier. If this is not possible, then the proof returned by the recursive DASH call is used to compute a suitable predicate.

Specifically, the auxiliary function ExtendFrontier makes a call to DASH at frontiers that correspond to call-return edges. ExtendFrontier first calls the auxiliary function GetWholeAbstractTrace (line 1). GetWholeAbstractTrace takes an ordered abstract error trace $\tau = \langle S_0, S_1, \ldots, S_n \rangle$ and forest F as input, and returns an "expanded" whole abstract error trace τ_w. Essentially, τ_w is the abstract trace τ with all call-return edges up to its frontier replaced with the abstract trace traversed in the called function (and this works in a recursive manner). If Edge(S_i, S_{i+1}) is a call-return edge that occurs before the fron-

ExtendFrontier($\tau,\ F,\ P$)
Returns:
($t,\ true$), if the frontier can be extended; or
($\epsilon,\ \rho$), if the frontier cannot be extended.

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

Figure 16: The auxiliary function ExtendFrontier for interprocedural analysis.

tier, GetWholeAbstractTrace runs a test t (obtained from the concrete witness in S_i) on the called procedure GetProc(e) and replaces Edge(S_i, S_{i+1}) with the sequence of regions corresponding to the test t.

The function ExecuteSymbolic (line 3) performs symbolic execution on the whole abstract error trace τ_w as described in Figure 13. If the frontier corresponds to a call-return edge (line 5) with a call to procedure $Q = \langle \Sigma, \sigma^I, \rightarrow \rangle$, ExtendFrontier calls Dash on the property checking instance $(\langle \Sigma, \sigma \wedge \phi, \rightarrow \rangle, \neg\phi')$. The predicate ϕ corresponds to the constraints on Q's input variables which are computed directly from the symbolic memory $\mathcal{S}$ (by the auxiliary function InputConstraints at line 7), and $\phi' = S_k[e/x]$, where e is the returned expression in Q and x is the variable in the caller P that stores the return value. Note that because both ϕ and ϕ' may mention local variables with the same names as variables in the called function, either the identifiers in these predicates or the identifiers in the called function need to be varied appropriately at the point where Dash is called recursively. While this must be done carefully so that AddTestToForest can correctly match up concrete states with abstract states, these details are omitted here.

If Dash$(\langle \Sigma, \sigma \wedge \phi, \rightarrow \rangle, \neg\phi')$ returns ("fail", t), then we know that the frontier can be extended by the test t; otherwise m corresponds to a proof that the frontier cannot be extended across the frontier. Computing a WP_α in this event would be expensive if the called function had several paths, but we can glean information from the way Dash splits the initial region to get a suitable

Program	Lines	Property	SLAM			DASH		
			Iters	TP-calls	Time(secs)	Iters	TP-calls	Time(secs)
bluetooth-correct	700	SpinLock	-	-	-	553	553	27.90
floppy-correct	6500	InterlockedQueuedIrps	*	*	*	726	726	14.56
floppy-correct	6500	SpinLock	*	*	*	826	826	14.17
floppy-buggy	6500	SpinLock	*	*	*	493	493	8.90
serial-buggy	10380	SpinLock	*	*	*	982	982	16.95
serial-correct	10380	SpinLock	*	*	*	2297	2297	48.66
bluetooth-correct	700	CancelSpinLock	5	1183	4.26	275	275	2.15
bluetooth-buggy	700	CancelSpinLock	5	1413	5.69	171	171	1.59
bluetooth-buggy	700	SpinLock	6	2453	8.1	171	171	1.69
diskperf-correct	2365	CancelSpinLock	2	15	1.76	123	123	1.95
diskperf-buggy	2365	CancelSpinLock	3	92	2.55	3	3	1.21
diskperf-correct	2365	MarkIrpPending	5	278	2.35	318	318	3.15
diskperf-buggy	2365	MarkIrpPendnig	5	440	2.35	2	2	1.22
floppy-correct	6500	CancelSpinLock	3	2851	7.81	538	538	5.41
floppy-buggy	6500	CancelSpinLock	3	2490	7.19	91	91	1.61
floppy-buggy	6500	InterlockedQueuedIrps	7	6688	24.84	1147	1147	17.21
floppy-correct	6500	MarkIrpPending	4	2513	11.84	568	568	5.68
floppy-buggy	6500	MarkIrpPending	3	2506	10.95	110	110	1.98

Table 1: Comparison of SLAM with DASH. "*" indicates timeout after 30 minutes, and "-" indicates that the tool gave up due aliasing issues.

predicate that is more general than the path predicate ϕ. This predicate is computed by the auxiliary function GetInitPred in line 12 which takes the proof m returned by DASH and returns a suitable predicate ϕ_2. The rest of the interprocedural algorithm is identical to DASH.

5 Evaluation

We have implemented DASH using the CIL infrastructure [22], and the F# programming language [1]. We use the Z3 theorem prover [8] that can also do model generation.

The implementation of DASH is very close to the description in Section 4. The only notable exception is that, when faced with an if-branch in a program, DASH will perform an inexpensive test to see whether the WP_α of a weaker predicate, one that ignores the branch condition, still satisfies the template described in Figure 11. This can be done by evaluation, and does not require a theorem prover call. The effect of this optimization is that we avoid getting "stuck" in irrelevant loops. We have left the consideration of more thorough generalization techniques for future work.

Implementing the interprocedural DASH algorithm in the presence of pointers was non-trivial. Each invocation of the DASH algorithm carries its own abstract graph, as well as a logical memory map representing the state of memory when the function was called. The top-level invocation of DASH assumes that there is no aliasing in this map, but recursive calls may begin with aliasing constraints introduced during the execution of the program. When a recursive call begins, a fresh abstraction is generated from the control flow graph of that function and is augmented with initial and error regions as described in Section 4.4.

We did three sets of evaluations to compare DASH and SLAM[3].

[3] In order to make a fair comparison with SLAM, we modified SLAM so that it also calls the theorem prover Z3.

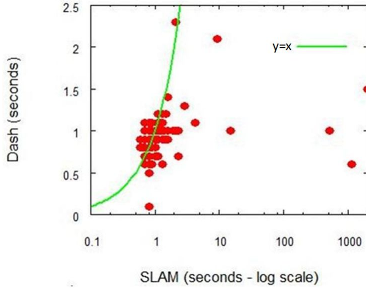

Figure 17: Scatter plot of the relative runtimes of SLAM and DASH on 95 C programs in SLAM's regression suite.

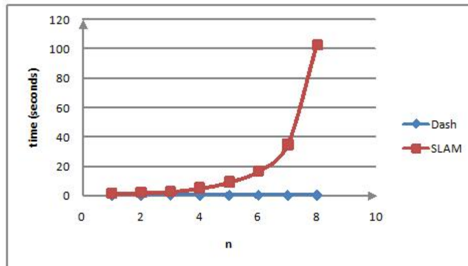

Figure 18: Plot illustrating the exponential time taken by SLAM on the program (parameterized by n) in Figure 3. DASH, on the other hand, takes almost constant time on this class of programs.

Device driver benchmarks. Table 1 compares SLAM and DASH on device driver code. In the first 6 cases where SLAM either times out or gives up due to pointer aliasing, DASH is able to prove that the program satisfies the property or find a test that witnesses the violation very efficiently. This is due to the fact that the refinement done by DASH using WP_α considers only the aliasing possibilites that occur along test executions. For the `floppy-correct` program and `SpinLock` property, the situation is similar to the simplified code snippet in Figure 4 in Section 2 (the example code in Figure 4 was motivated by looking at the floppy driver code relating to this property and simplifying it for presentation). As seen in the table, even though DASH takes several more iterations when compared to SLAM, each iteration is very efficient, and the overall runtime of DASH is smaller than SLAM. This is because in each iteration, SLAM makes a large number of theorem prover calls to compute the boolean program abstraction, whereas DASH makes exactly one theorem prover call per iteration.
SLAM regression suite. We ran DASH on 95 C programs in SLAM's regression suite. A scatter plot of the relative runtimes of SLAM and DASH can be seen in Figure 17. SLAM and DASH gave identical outputs (that is, pass/fail) on

each of the 95 programs. Note that the plot has SLAM runtime in a log scale, and the curve $y = x$ is shown. Every point to the right of the curve is a case where DASH is faster than SLAM. The total time taken by SLAM for all the 95 programs (put together) is 20 minutes. DASH finishes all the 95 programs in 17 seconds, a speedup of 70X. With test caching enabled (where tests are reused across runs of DASH), DASH finishes all the 95 programs in 4 seconds, a speedup of 300X!

Microbenchmark for alias issues. Finally, we varied the parameter n in the template program in Figure 3 and compared the runtimes of SLAM and DASH. The results are shown in Figure 18. As explained in Section 2, SLAM's runtime varies exponentially with n due to the fact that it considers and rules out an exponential number of aliasing possibilities, whereas DASH takes almost constant time.

6 Conclusion

We believe that light-weight approaches like DASH enable application of proof techniques to a larger class of programs. Our eventual goal is the following: whenever we can run a program, instrument a program to observe states, and do light-weight symbolic execution, we want to be able to do proofs! We believe that DASH has all the concepts needed to achieve this goal.

DASH handles only sequential programs, and checks only safety properties. However, recent work has built on checkers like SLAM to do concurrency analysis with bounded number of context switches [23], and check termination properties [7]. By improving the scalability of the core proof engines (like SLAM), we believe that DASH can also improve the scalability of these tools for concurrency and termination analysis.

References

[1] http://research.microsoft.com/fsharp/fsharp.aspx.

[2] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI '01: Programming Language Design and Implementation*, pages 203–213. ACM Press, 2001.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: SPIN workshop on Model checking of Software*, pages 103–122. Springer-Verlag New York, Inc., 2001.

[5] T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE '01: Program Analysis for Software Tools and Engineering*, pages 97–103. ACM Press, 2001.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV '00: Computer Aided Verification*, pages 154–169. Springer-Verlag, 2000.

[7] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI '06: Programming Language Design and Implementation*, pages 415–426. ACM, 2006.

[8] L. de Moura and N. Bjorner. Z3: An efficient smt solver. In *TACAS '08: Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[9] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[10] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1997.

[11] P. Godefroid. Compositional dynamic test generation. In *POPL '07: Principles of Programming Languages*, pages 47–54. ACM Press, 2007.

[12] P. Godefroid and N. Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In *IFM '05: Integrated Formal Methods*, pages 20–32, 2005.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI '05: Programming Language Design and Implementation*, pages 213–223. ACM Press, 2005.

[14] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *FSE '06: Foundations of Software Engineering*, pages 117–127. ACM Press, 2006.

[15] E. Gunter and D. Peled. Model checking, testing and verification working together. *Form. Asp. Comput.*, 17(2):201–221, 2005.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL '04: Principles of Programming Languages*, pages 232–244. ACM Press, 2004.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[18] D. Kroening, A. Groce, and E. M. Clarke. Counterexample guided abstraction refinement via program execution. In *ICFEM '04: International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, pages 224–238, 2004.

[19] R. Majumdar and K. Sen. LATEST : Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley, March 2007.

[20] J. M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, Lecture Notes of an International Summer School, pages 25–34. "D. Reidel Publishing Company, 1982.

[21] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV '00: Computer Aided Verification*, pages 435–449. Springer-Verlag, 2000.

[22] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: International Conference on Compiler Construction*, pages 213–228. Springer-Verlag, 2002.

[23] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI '04: Programming Language Design and Implementation*, pages 14–24. ACM, 2004.

[24] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE '05: Foundations of Software Engineering*, pages 263–272. ACM Press, 2005.

[25] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA '06: International Symposium on Software Testing and Analysis*, pages 145–156. ACM Press, 2006.