We present an algorithm \textsc{Dash} to check if a program $P$ satisfies a safety property $\varphi$. 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 $\text{WP}_\alpha$ 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 \textsc{Dash} algorithm, its theoretical properties, and its implementation. Our empirical results show significant improvement over \textsc{Slam} in the runtimes of several examples. We also show several cases where \textsc{Slam} is unable to terminate due to pointer analysis imprecisions, whereas \textsc{Dash} is able to terminate in seconds.
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 \( \text{WP}_\alpha \) that combines the usual weakest precondition operator \([10]\) with an alias set \( \alpha \). The alias set \( \alpha \) is obtained during execution of the specific test that the algorithm is attempting to extend. The predicate obtained from the \( \text{WP}_\alpha \) 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 \( \text{WP}_\alpha \) 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 counter-example-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 \( \Sigma \) and a set of error states \( \varphi \). 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 \( \Sigma_\approx \) of the infinite state space \( \Sigma \). The states of the abstraction, called regions, are equivalence classes of concrete program states from \( \Sigma \). 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 \( \varphi \), 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
Figure 1: Example where SLAM has difficulty without a flow-sensitive alias analysis.

Dash now tries to find a new test which follows the ordered path $\tau_e$ for at least one transition past the prefix $\tau$. It uses directed testing [13, 24] to generate such a test. These techniques perform a light-weight symbolic execution along the path $\tau_e$, and collect constraints at every state as functions of the inputs to the program. In programs with pointers, the symbolic execution along $\tau_e$ 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 $\tau$ fails, then the alias conditions $\alpha$, obtained by the symbolic execution up to the prefix $\tau$, can be used to refine the abstraction $\Sigma$. This refinement does not make any theorem prover calls, and does not use a global alias analysis. We define a new operator $\text{WP}_\alpha$ to perform such a refinement. The $\text{WP}_\alpha$ operator specializes the weakest precondition operator using only the alias conditions $\alpha$ that occur along the test up to the prefix $\tau$. Using the predicate generated by the $\text{WP}_\alpha$ operator, we can refine the region at the end of the prefix $\tau$ and remove the abstract transition from the prefix $\tau$ along the ordered trace $\tau_e$. 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 $\varphi$ or the refined abstraction $\Sigma$ provides a proof of correctness that $\varphi$ 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-
The abiding feature of this example is that \( p \) may alias with \( p_1 \) or \( p_2 \) 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 \( p_1 \) or \( p_2 \), 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 \((p \neq p_1 \land p \neq p_2)\) that occurs along concrete executions.

DASH first creates the initial abstraction \( \Sigma_\equiv \) for the program \texttt{prove-me1} and this is isomorphic to its control flow graph (shown in Figure 2(a)). We show regions of the abstraction \( \Sigma_\equiv \) as “clouds,” and states from the forest \( F \) using “\( \times \)”s in the figure. Next, the initial forest is created by running \texttt{prove-me1} with a random test that assigns values to its inputs \( p \), \( p_1 \) and \( p_2 \), thus creating a forest \( F_{\text{prove-me1}} \) 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 \( \tau_e \) 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 \( F_{\text{prove-me1}} \) that follows \( \tau_e \) for at least one transition beyond the prefix \( \tau \) 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 \( \tau \) together with the transition \((3, 4)\) are unsatisfiable, so such a test does not exist. DASH now refines the region 3 using the operator \( WP_\alpha \) (defined in Section 4.2.1). In this case, the \( WP_\alpha \) operator returns the predicate \( \rho = (p_1->lock=1) \lor (p_2->lock=1) \). DASH splits the region 3 into two regions, 3: \( \rho \) and 3: \( \lnot \rho \). Due to the properties of the \( WP_\alpha \) 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;
2: 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;
9: p = p2;
...pn = 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 = (0,1,2,3: \rho,4) with prefix \tau = (0,1,2) 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_\alpha operator to obtain the predicate \eta = \neg ((p \neq p1 \land p \neq p2) \land \neg (p1->lock=1 \lor p2->lock=1)). Intuitively, the subexpression \alpha = (p \neq p1 \land p \neq p2) corresponds to the alias relations that hold between the variables p, p1, and p2 after the program executes the path \tau. The subexpression p1->lock=1 \lor p2->lock=1 is the weakest precondition along the transition (2,3: \rho) assuming the aliasing constraints imposed by \alpha. 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 \land 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) {
    de->lock = 0;
    do {
        assert(de->lock == 0);
        de->lock = 1;
        x = de->y;
        if (*) {
            de->lock = 0;
            Inc(de);
        }
    } while (x != de->y);
}

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

void prove-me2(bool b1, bool b2) {
    b1 = true; // de->lock = 0
    do {
        assert(b1); // assert(de->lock==0)
        b2 = true; // x = de->y
        if (*) {
            b1 = true; // de->lock = 0
            b2 = b2?false:*; // effect of Inc on b2
            b1 = *; // effect of Inc on b1
        }
    } 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 ∧ x!=de->y) ∨ (de->lock!=0 ∧ 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(bm)
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
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 imple-
mented 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 proce-
dures, 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 $\Sigma_{\equiv_P}$
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 $\tau$ (the prefix of the abstract error path $\tau_e$) 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 $\Sigma_{\equiv_{top}}$ and $\Sigma_{\equiv_{inc}}$ for the procedures
`top` and `inc` respectively (shown in Figure 9(a)). Note that these initial abstrac-
tions 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 con-
crete 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

Figure 7: Initial abstraction and proof computed by DASH on the example
program from Figure 6.
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: int r;  
1: r = y+1;  
2: return r;  
}  

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 $\Sigma_{\approx_{\text{inc}}}$ (shown in Figure 9(a)). DASH now tries to add a test to $F_{\text{top}}$ that follows $\tau_e$ for at least one transition beyond the prefix $\tau$ 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 $\Sigma_{\approx_{\text{inc}}}$ by removing the abstract transition $(2, 3)$. This is done using the $\text{WP}_\alpha$ 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 $\text{top}$, 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 $\text{inc}$, we make a recursive call to DASH on the procedure $\text{inc}$. This call to DASH checks whether a test can be run on $\text{inc}$ with a precondition induced by $\tau$ and postcondition induced by the region $2: \rho$ in $\text{top}$. 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 $\eta$ (shown in Figure 9(c)). We discuss how $\eta$ can be computed in Section 4.4. After some more iterations, DASH computes the abstraction $\Sigma_{\approx_{\text{inc}}}$ (shown in Figure 9(d)) that proves that the error location is unreachable in the procedure $\text{top}$.

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 \( \text{WP}_\alpha \) 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., \((\ast)^{k>1}p\)).

(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 \(\ast m = e\), where m is a memory location and e is an expression.

(e) All function calls (call-by-value function calls) are of the form \(\ast m = f(x_1,x_2,\ldots,x_n)\), where m is a memory location.

Though our presentation considers only pointer dereferences of the form \(\ast p\), our implementation also supports structs with fields, and pointers to structs with dereferences of the form \(p\rightarrow f\).

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 \(P_i = \langle N_i, L_i, E_i, n^0_i, \lambda_i, V_i \rangle\) 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 $\lambda_i : E_i \rightarrow \textit{Stmts}$, 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 $\text{CallRet}(E_i)$.

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 values 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 $P$ 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 $\Sigma$ be the (possibly infinite) state space of $P$, defined as the set of all valuations to the variables in $V$. Every statement $\textit{op} \in \textit{Stmts}$ defines a state transition relation $\textit{op} : \Sigma \times \Sigma$, 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 $\rightarrow$ 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 $(P, \varphi)$. The answer to $(P, \varphi)$ is “fail” if there is some initial state $s \in \sigma^f$ and some error state $s' \in \varphi$ such that $s \rightarrow s'$, 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) $s_0 \in \sigma^f$, (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 $\Sigma_\varphi$ of the state space $\Sigma$ which proves the absence of error traces. We refer to the equivalence classes of the partition $\Sigma_\varphi$ as regions. The partition $\Sigma_\varphi$ induces an abstract procedure $P_\varphi = \langle \Sigma_\varphi, \sigma_\varphi^f, \rightarrow_\varphi \rangle$, where $\sigma_\varphi^f = \{ S \in \Sigma_\varphi \mid S \cap \sigma^f \neq \emptyset \}$ is the set of regions that contain initial states, and $S \rightarrow_\varphi S'$ for $S, S' \in \Sigma_\varphi$ if there exist two states $s \in S$ and $s' \in S'$ such that $s \rightarrow s'$. We allow for the possibility that $S \rightarrow_\varphi S'$ when there do not exist states $s \in S$ and $s' \in S'$ such that $s \rightarrow s'$.

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

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

### 4.1 The Dash Algorithm

We will first assume that the program $P = \langle P \rangle$ has one procedure $P$, and discuss how we handle programs with multiple procedures in Section 4.4. The algorithm
Dash shown in Figure 10 takes the property checking instance \((P, \varphi)\) as input and can have three possible outcomes:

1. It may output “fail” together with a test \(t\) that certifies that \(P\) can reach \(\varphi\).

2. It may output “pass” together with a proof \(\Sigma_\approx\) that certifies that \(P\) cannot reach \(\varphi\).

3. It may not terminate.

Dash maintains two data structures:

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

2. A finite-indexed partition \(\Sigma_\approx\) of the state space \(\Sigma\) of \(P\).
The regions of $\Sigma_=$ are defined by $pc$ values and predicates over program variables. Let $pc(S)$ denote the program location associated with region $S$, and let $\text{Edge}(S, S')$ 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 $P_=$ is initially isomorphic to the control flow graph of the procedure $P$. The function $\text{Test}$ (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 $\Sigma_=$ (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 $\text{frontier}$) 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 $\Sigma_=$ 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 $P_=$.

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 $\text{TestForWitness}$ to compute a concrete test that reaches the error. $\text{TestForWitness}$ (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. $\text{TestForWitness}(s)$ 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 $\text{GetAbstractTrace}$ (line 12) to find an abstract error trace $\tau$ through the abstract graph. If no such trace exists, then the current partition $\Sigma_=$ is a proof that $P$ cannot reach any state in $\varphi$, and $\text{GetAbstractTrace}$ returns $\tau = \epsilon$. Otherwise, $\text{GetAbstractTrace}$ 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 $(k-1, k)$ defined $\text{Frontier}(S_0, S_1, \ldots, S_n)$ 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 $S_i = \text{Region}(\text{parent}^{k-1-i}(s))$ for all $0 \leq i < k$, where the abstraction function $\text{Region}$ maps each state $s \in \Sigma$ to the region $S \in \Sigma_=$ with $s \in S$. 

14
ExtendFrontier(τ, F, P)
Returns:
⟨t, true⟩, if the frontier can be extended; or
⟨ε, ρ⟩, if the frontier cannot be extended.
1: (k−1, k) := Frontier(τ)
2: (φ1, S, φ2) := ExecuteSymbolic(τ, F, P)
3: t := lsSAT(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⟩
ExecuteSymbolic($\tau, F, P$)
Returns: $(\phi_1, S, \phi_2)$.
1: $(k-1, k) := \text{Frontier}(\tau = \langle S_0, S_1, \ldots, S_n \rangle)$
2: $S := [v \mapsto v_0 \mid *v \in \text{inputs}(P)]$
3: $\phi_1 := \text{SymbolicEval}(S_0, S)$
4: $\phi_2 := \text{true}$
5: $i := 0$
6: while $i \neq k-1$ do
7: $op := \wedge(\text{Edge}(S_i, S_{i+1}))$
8: match$(op)$
9: case$(+m = e)$:
10: $S := S + \langle \text{SymbolicEval}(m, S) \rightarrow \text{SymbolicEval}(e, S) \rangle$
11: case$(\text{if } e \text{ goto } l)$:
12: $\phi_2 := \phi_1 \wedge \text{SymbolicEval}(e, S)$
13: $i := i + 1$
14: $\phi_1 := \phi_1 \wedge \text{SymbolicEval}(S_i, S)$
15: end while
16: $op := \lambda(\text{Edge}(S_{k-1}, S_k))$
17: match$(op)$
18: case$(+m = e)$:
19: $\phi_2 := \phi_2 \wedge + \langle \text{SymbolicEval}(m, S) \rangle = \text{SymbolicEval}(e, S)$
20: $S' := S + \langle \text{SymbolicEval}(m, S) \rightarrow \text{SymbolicEval}(e, S) \rangle$
21: case$(\text{if } e \text{ goto } l)$:
22: $\phi_2 := \phi_2 \wedge \text{SymbolicEval}(e, S)$
23: $S' := S$
24: $\phi_2 := \phi_2 \wedge \text{SymbolicEval}(S_k, S')$
25: return $(\phi_1, S, \phi_2)$

Figure 13: The auxiliary function ExecuteSymbolic.

$\Sigma_{\infty}$ 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 $\tau_0$.

The function ExecuteSymbolic, which is called at line 2 of ExtendFrontier, performs symbolic execution on $\tau$ using techniques inspired by CUTE [24]. Let $\tau = \langle S_0, S_1, \ldots, S_n \rangle$, and let $(k-1, k) = \text{Frontier}(\tau)$. ExecuteSymbolic returns $(\phi_1, S, \phi_2)$, where $\phi_1$ and $S$ 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 $\phi_2$ 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 $S$. ExecuteSymbolic is described in Figure 13. It first initializes the symbolic memory map $S$ 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 $\phi_1$ and $\phi_2$. The function SymbolicEval$(e, S)$ evaluates the expression $e$ with respect to values from the symbolic memory $S$.

ExtendFrontier calls the function IsSAT (line 3 in Figure 12) that checks whether $\mu = \phi_1 \wedge S \wedge \phi_2$ is satisfiable by making a call to a theorem prover. If $\mu$ 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 $\rho$ that is a suitable candidate for

---

1Every entry in $S$ is looked upon as an equality predicate here.
refining $\Sigma_\approx$ at $S_{k-1}$ according to the template in Figure 11. It is useful to note that $\text{RefinePred}$ makes no theorem prover calls in order to compute $\rho$.

### 4.2 Suitable Predicates

If we cannot drive a test-case past the frontier, then $\text{RefinePred}$ 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 $\rho$ is too weak, then it will be possible to derive a test along the same ordered abstract trace, in which case $\text{RefinePred}$ will be called with the exact same arguments and will return $\rho$ again. Alternatively, if $\rho$ is too strong, then there may be a transition from some region in $S_{k-1} \land \neg p$ 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 $\text{Dash}$ to make progress in a sound manner and also that the predicate returned by $\text{RefinePred}$ is a suitable predicate.

**Definition 1 (Suitable predicate)** Let $\tau$ be an abstract error trace and let $(S, T)$ be its frontier. A predicate $\rho$ is said to be suitable with respect to $\tau$ only if all possible concrete states obtained by executing $\tau$ up to the frontier belong to the region $S \land \neg \rho$, and if there is no transition from any state in $S \land \neg \rho$ 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 $\tau \sqsubseteq \tau'$ if either of the following conditions is true.

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

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

Essentially, this means that $\tau \sqsubseteq \tau'$ if $\tau'$ is a strictly “better” trace, either because the frontier in $\tau'$ has been pushed forward or because at least one region in $\tau'$ 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 $\text{Dash}$. Then we say that $\text{Dash}$ 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 $\tau$ is used to perform refinement, then the $\text{Dash}$ 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 $\rho$ with respect to $\tau$ would eliminate the edge $(S_{k-1}, S_k)$ in a sound manner by splitting $S_{k-1}$ into two regions, $S_{k-1} \land \rho$ and $S_k \land \rho$. Since all concrete states in $S_{k-1}$ that can be obtained by traversing the abstract error trace belong to the region $S_{k-1} \land \neg \rho$, and the edge $(S_{k-1} \land \rho, S_k)$ does not exist,
RefinePred($S$, $\phi_2$, $\tau$)

Returns: a suitable predicate $\rho$.

1: $(k-1, k) := \text{Frontier}(\tau = (S_0, S_1, \ldots, S_m))$
2: $op := \lambda(\text{Edge}(S_{k-1}, S_k))$
3: $\alpha := \text{Aliases}(S, op, S_k)$
4: return $\text{WP}_\alpha(op, S_k)$

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

\[
\begin{align*}
& (\& x = p \land \& x = q \land 2x < -1) \lor \\
& (\& x \neq p \land \& x = q \land *p + x < 0) \lor \\
& (\& x = p \land \& x \neq q \land x + *q < 0) \lor \\
& (\& x \neq p \land \& x \neq q \land *p + *q < 1)
\end{align*}
\]

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 $\alpha$ 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 \texttt{RefinePred} in Figure 14.

We first define the projection of the weakest precondition with respect to alias condition \( \alpha \) as follows:

\[
\text{WP}_{\downarrow \alpha}(\text{op},\phi) = \alpha \land \text{WP}(\text{op},\phi)
\]

For efficiency, \( \text{WP}_{\downarrow \alpha}(\text{op},\phi) \) can be computed by only considering the alias possibility \( \alpha \). For example, if \( \alpha = (\& x \neq p \land \& x = q) \) we have that

\[
\text{WP}_{\downarrow \alpha}(x = x + 1, *p + *q < 1) = (\& x \neq p \land \& x = q \land x < 0)
\]

The refinement predicate computed by \texttt{RefinePred} is

\[
\text{WP}_\alpha(\text{op},\phi_2) \overset{\text{def}}{=} \neg(\alpha \land \neg \text{WP}_{\downarrow \alpha}(\text{op},\phi_2))
\]

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

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

\textit{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 \( \alpha \). 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 \texttt{RefinePred} would not be called). Thus, it must be the case that \( \forall c \in \mathcal{C}, c \notin \text{WP}_{\downarrow \alpha}(\text{op},\phi_2) \). This implies that \( \forall c \in \mathcal{C}, c \in (\alpha \land \neg \text{WP}_{\downarrow \alpha}(\text{op},\phi_2)) \) and hence the predicate \( \text{WP}_\alpha(\text{op},\phi_2) \) satisfies the first requirement of Definition 1.

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

We note that while \( \text{WP} \) is another possible choice for a suitable predicate for the refinement shown in Figure 11, the predicate computed by \( \text{WP} \) contains an exponential number of disjuncts in the presence of aliasing. Thus, the use of \( \text{WP}_\alpha \) avoids an exponential number of disjuncts when compared to other approaches that use \( \text{WP} \) such as [14] and [21].
Dash-Main(\(P, \varphi\))

Returns:
("fail", \(t\)), where \(t\) is an error trace of \(P\) reaching \(\varphi\); or
("pass", \(\Sigma\)), where \(\Sigma\) is a proof that \(P\) cannot reach \(\varphi\).

1: let \((P_0, P_1, \ldots, P_n) = P\) in
2: \(\text{Dash}(P_0 = (\Sigma_0, \sigma^0_0, \rightarrow_0), \varphi)\)

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 \((P, \varphi)\), then either of the following is true: (1) if DASH returns ("pass", \(\Sigma\)), then \(\Sigma\) is a proof that \(P\) cannot reach \(\varphi\), and (2) if DASH returns ("fail", \(t\)), then \(t\) is a test for \(P\) that violates \(\varphi\). 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 \(\text{IsSat}\) in line 3 of the auxiliary function \(\text{ExtendFrontier}\)). 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 \(\varphi\) that we wish to check is only associated with the main procedure \(P_0\) in the program \(P\). Therefore, \(\text{Dash-Main}(P = (P_0, P_1, \ldots, P_n), \varphi)\) (shown in Figure 15) calls the function DASH from Figure 10 on the property checking instance \((P_0, \varphi)\). As in the single procedure case, we maintain a forest \(F\) and an abstraction \(P_\approx\) 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 \(\text{ExtendFrontier}\). The modified version of \(\text{ExtendFrontier}\) 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 \(\text{ExtendFrontier}\) makes a call to DASH at frontiers that correspond to call-return edges. \(\text{ExtendFrontier}\) first calls the auxiliary function \(\text{GetWholeAbstractTrace}\) (line 1). \(\text{GetWholeAbstractTrace}\) takes an ordered abstract error trace \(\tau = (S_0, S_1, \ldots, S_n)\) and forest \(F\) as input, and returns an "expanded" whole abstract error trace \(\tau_w\). Essentially, \(\tau_w\) is the abstract trace \(\tau\) 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 \(\text{Edge}(S_i, S_{i+1})\) 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. $\tau_w = (S_0, S_1, \ldots, S_n) := \text{GetWholeAbstractTrace}(\tau, F)$
2. $(k-1, k) := \text{Frontier}(\tau_w)$
3. $(\phi_1, S, \phi_2) := \text{ExecuteSymbolic}(\tau_w, F, P)$
4. if $\text{Edge}(S_{k-1}, S_k) \in \text{CallReturn}(E)$ then
5. let $(\Sigma, \sigma^I, \neg \rightarrow) = \text{GetProc}(\text{Edge}(S_{k-1}, S_k))$ in
6. $\phi := \text{InputConstraints}(S)$
7. $\phi' := S_k[e/x]$
8. $(r, m) := \text{Dash}((\Sigma, \sigma^I \land \phi, \neg \rightarrow), \neg \phi')$
9. if $r = \text{"fail"}$ then
10. let $(\Sigma, \sigma^I, \neg \rightarrow, \neg \phi'') = \text{GetProc}(\text{Edge}(S_{k-1}, S_k))$ in
11. $\rho := \text{GetInitPred}(m)$
12. $t := \epsilon$
13. end if
14. else
15. $t := \text{IsSAT}(\phi_1, S, \phi_2, P)$
16. if $t = \epsilon$ then
17. $\rho := \text{RefinePred}(S, \tau_w)$
18. else
19. $\rho := true$
20. end if
21. end if
22. return ($t, \rho$)

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 $\text{Edge}(S_i, S_{i+1})$ with the sequence of regions corresponding to the test $t$.

The function ExecuteSymbolic (line 3) performs symbolic execution on the whole abstract error trace $\tau_w$ as described in Figure 13. If the frontier corresponds to a call-return edge (line 5) with a call to procedure $Q = (\Sigma, \sigma^I, \rightarrow)$, ExtendFrontier calls DASH on the property checking instance $((\Sigma, \sigma^I \land \phi, \rightarrow), \neg \phi')$. The predicate $\phi$ corresponds to the constraints on $Q$'s input variables which are computed directly from the symbolic memory $S$ (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 $\phi$ and $\phi'$ 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($((\Sigma, \sigma \land \phi, \rightarrow), \neg \phi')$) 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$_\alpha$ 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
Table 1: Comparison of SLAM with DASH. "*" indicates timeout after 30 minutes, and "-" indicates that the tool gave up due aliasing issues.

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Property</th>
<th>Init</th>
<th>TP-calls</th>
<th>Time (secs)</th>
<th>Init</th>
<th>TP-calls</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bluetooth-correct</td>
<td>700</td>
<td>SpinLock</td>
<td>-</td>
<td></td>
<td></td>
<td>553</td>
<td>553</td>
<td>27.90</td>
</tr>
<tr>
<td>floppy-correct</td>
<td>6500</td>
<td>InterlockedQueuedIrps</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>floppy-buggy</td>
<td>6500</td>
<td>SpinLock</td>
<td>*</td>
<td></td>
<td></td>
<td>826</td>
<td>826</td>
<td>14.17</td>
</tr>
<tr>
<td>serial-correct</td>
<td>10380</td>
<td>SpinLock</td>
<td>-</td>
<td></td>
<td></td>
<td>493</td>
<td>493</td>
<td>8.90</td>
</tr>
<tr>
<td>serial-buggy</td>
<td>10380</td>
<td>SpinLock</td>
<td>*</td>
<td></td>
<td></td>
<td>982</td>
<td>982</td>
<td>16.95</td>
</tr>
<tr>
<td>bluetooth-buggy</td>
<td>700</td>
<td>CancelSpinLock</td>
<td>5</td>
<td></td>
<td></td>
<td>1183</td>
<td></td>
<td>4.26</td>
</tr>
<tr>
<td>floppy-buggy</td>
<td>6500</td>
<td>CancelSpinLock</td>
<td>5</td>
<td></td>
<td></td>
<td>1413</td>
<td></td>
<td>5.69</td>
</tr>
</tbody>
</table>
| diskperf-buggy   | 2365  | CancelSpinLock            | 2    |          |             | 15   | 1.76     | 123         | 1.95
| diskperf-buggy   | 2365  | SpinLock                  | 3    |          |             | 92   | 2.55     | 3           | 1.21
| diskperf-buggy   | 2365  | MarkIrpPending            | 5    |          |             | 278  | 2.35     | 318         | 3.15
| floppy-buggy     | 6500  | MarkIrpPending            | 5    |          |             | 440  | 2.35     | 2           | 1.22
| floppy-buggy     | 6500  | CancelSpinLock            | 3    |          |             | 2851 | 7.81     | 538         | 5.41
| floppy-buggy     | 6500  | CancelSpinLock            | 3    |          |             | 2490 | 7.19     | 91          | 1.61
| floppy-correct   | 6500  | InterlockedQueuedIrps     | 7    |          |             | 6688 | 24.84    | 1147        | 17.21
| floppy-buggy     | 6500  | MarkIrpPending            | 4    |          |             | 2513 | 11.84    | 568         | 5.68
| floppy-buggy     | 6500  | MarkIrpPending            | 3    |          |             | 2506 | 10.95    | 110         | 1.98

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 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$_\alpha$ considers only the aliasing possibilities 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


