Proofs from Tests

Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, SaiDeep Tetali, Aditya V. Thakur

Abstract—We present an algorithm DASH to check if a program P satisfies a safety property φ. The unique feature of this 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 inexpensive, and can be implemented without any global may-alias information. In particular, we introduce a new refinement operator WP_α that uses only the alias information obtained by symbolically executing a test to refine abstractions in a sound manner. We present a full exposition of the DASH algorithm and its theoretical properties. We have implemented DASH in a tool called YOGI that plugs into Microsoft's Static Driver Verifier framework. We have used this framework to run YOGI on 69 Windows Vista drivers with 85 properties and find that YOGI scales much better than SLAM, the current engine driving Microsoft's Static Driver Verifier.

Index Terms—Software model checking; Directed testing; Abstraction refinement

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" [1]. 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 [2], [3], [4] 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 [5], which simultaneously performs program testing and program abstraction in order to check safety properties of programs. The tests are an "under-approximation" of the program's behavior, and the abstraction is an "over-approximation" of the program's behavior. The goal is to either find a test that reaches an error state, or find an abstraction that is precise enough to show that no path in the program can reach any error state. 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 during a symbolic execution of a 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 proving properties of programs with pointers try to reason about aliasing using a conservative whole program "may-alias" analysis (see Section 4.2 in [6], and Section 6 in [7]). 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 [8] with an alias set α. The alias set α is obtained during symbolic execution of the specific test that the algorithm is attempting to extend. 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 tools such as SLAM [9] and BLAST [7].

Even though DASH uses alias information from tests to avoid explosion in the computation of WP, the idea is useful in several other settings. For instance, explosion

```
struct ProtectedInt
{
  int *lock;
  int *y;
};

int *lock1, *lock2;

void LockUnlock(struct ProtectedInt *pi, int x)
{
0:  int do_return = 0;
1:  if (pi->lock == lock1) {
2:    do_return = 1;
3:    pi->lock = lock2;
    }
4:  else if (pi->lock == lock2) {
5:    do_return = 1;
6:    pi->lock = lock1;
    }
    //initialize all locks to be unlocked
7:  *(pi->lock) = 0;
8:  *lock1 = 0;
9:  *lock2 = 0;

10: if (do_return) return;
11: else {
12:   do {
13:       if (*(pi->lock) != 0)
14:         error();
15:       *(pi->lock) = 1; // lock
16:     if(*lock1 ==1 || *lock2 ==1)
17:       error();
18:     x = *(pi->y);
19:     if (NonDet()) {
20:       (*(pi->y))++;
21:       if (*(pi->lock) != 1)
22:         error();
23:       *(pi->lock) = 0; // unlock
24:     }
25:   } while(x != *(pi->y));
    }
26: if (*(pi->lock) != 1)
27:   error();
28: *(pi->lock) = 0; // unlock
}
```

Fig. 1. The LockUnlock function acquires and releases `pi->lock` in strict alternation.

due to aliasing even happens using other methods for refinement, such as interpolants [10]. Thus, we believe that the WP_α idea of using aliasing information from tests can be useful in other verification tools.

This journal paper is an extended version of our conference paper [11] with more comprehensive experimental results and a more detailed explanation of the DASH algorithm.

2 OVERVIEW

Over the past few years, several tools based on predicate abstraction and counterexample-guided abstraction refinement, such as SLAM [9] and BLAST [7], have been built in order to compute proofs of programs for various properties. The algorithms implemented in these tools have two main bottlenecks. First, the algorithms require many expensive calls to a theorem prover at every step, 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, EXE and CUTE using light-weight symbolic execution [2], [3], [4]. These testing tools focus on finding errors in programs by way of explicit path model checking and are unable, in general, to establish that errors do not exist. Our work can be viewed as combining the successful ideas from proof-based tools like SLAM and BLAST with testing-based tools like DART, EXE and CUTE with the goal of improving scalability.

Motivating Example. We use the example program in Figure 1 as a motivating example. An error is said to have occurred in this program if the `error()` function is ever called. This program locks `pi->lock` at line `15` and unlocks `pi->lock` at line `23` and line `28`. Locking and unlocking can happen an arbitrary number of times as they occur inside a do-while loop (lines `12 - 25`). The `NonDet()` function call at line `19` models nondeterminism, and is assumed to return either true or false arbitrarily. Even though the function `LockUnlock` never raises any error, proving this automatically is challenging. First, if `lock1` or `lock2` alias with `pi->lock`, the function returns without entering the loop. However, due to the assignments in lines `3` and `6`, a flow-sensitive analysis would fail to determine the fact that inside the do-while loop `lock1` or `lock2` do not alias `pi->lock`. Thus, it is challenging to show that the error in line `17` cannot be reached with anything short of path-sensitive alias information. As we show below, DASH uses alias sets that occur by performing symbolic execution on paths explored by concrete tests, and a new operator WP_α to obtain only the path-sensitive alias information that is required to prove correctness. Second, the do-while loop continues to execute only if the 'then' branch of the conditional at line `19` is entered. The loop invariant (at line `25`) is that `*(pi->lock)` is `1` (meaning 'locked') if and only if `(x == *(pi->y))`. As we will show, DASH can automatically discover this non-trivial invariant.

DASH Algorithm. 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 previously-run 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'. This abstraction is initially just the control flow graph of a program, but is gradually be refined over time in an attempt to prove that error states are unreachable. At all times this abstraction represents an over-approximation of all concrete executions, so that if there is no path from the initial region to the error region φ, we can be sure that there is no path of concrete transitions from some concrete initial state to a concrete error state and a proof of correctness has been found.

In every iteration of the DASH algorithm, we first find an abstract error path (a path through the abstraction

$\Sigma_\simeq$) from the initial region to the error region. If no such abstract error path exists, then a proof of correctness has been found. If any such abstract error path exists, then we can always find an "ordered" path τ_e with a prefix τ_e^p such that (1) τ_e^p corresponds to a concrete path in F and (2) no region in τ_e after the prefix τ_e^p is visited in F. We will refer to the edge in τ_e that connects the last region in the prefix τ_e^p to the first unvisited region in τ_e, as the *frontier*. For example, Figure 2(a) shows an abstract graph. We represent regions of the abstraction $\Sigma_\simeq$ as "clouds" and represent states from the forest F using "$\times$"s in the figure. This abstract graph has an initial region `0` and error region `9`. The edge connecting region `4` and region `7` in Figure 2(a) is the frontier for the abstract error path $\tau_e = \langle 0,1,2,3,4,7,8,9\rangle$. DASH now tries to find a new test which follows the ordered path τ_e for at least one transition past the frontier.

Techniques developed for directed testing [2], [3], [4] are used to generate this test. Specifically, a light-weight symbolic execution along the path τ_e^p collects constraints at every state as functions of the inputs to the program. In programs with pointers, the symbolic execution along τ_e^p is done in a "pointer-aware" manner keeping track of the aliases between variables in the program. Let φ_1 be the constraint generated by performing symbolic execution along τ_e^p, and let φ_2 be the constraint generated by extending the symbolic execution along the frontier edge (see Figure 2 (a)). We have two possibilities:

- If the formula $\varphi_1 \wedge \varphi_2$ is satisfiable, then any test input satisfying it moves the frontier forward and closer to the error region. For instance, in Figure 2, the generated test (satisfying $\varphi_1 \wedge \varphi_2$) moves the frontier forward from edge $(4,7)$ in Figure 2(a) to the edge $(8,9)$ in Figure 2(b).
- If $\varphi_1 \wedge \varphi_2$ is unsatisfiable, we can prove that there exist so called 'suitable' predicates (defined precisely later) to refine the region at the frontier so that the frontier moves backward. For instance, in Figure 2(b) we find that the region `4` can be partitioned into two regions $4:\neg\rho$ and $4:\rho$ such that all concrete states reached by tests along τ_e^p are all in the region $4:\neg\rho$, and there is no edge in the abstraction graph from region $4:\neg\rho$ to region 7. Due to this refinement the frontier backward from edge $(4,7)$ in Figure 2(a) to the edge $(3,4:\rho)$ in Figure 2(c). A key insight in the DASH algorithm is that if the formula $\varphi_1 \wedge \varphi_2$ is unsatisfiable, then the alias conditions α from the formula φ_1 can be used to compute a suitable predicate ρ for refinement (see Section 4.2 for details).

In either case, the DASH algorithm continues by choosing a new ordered error path until either some test that reaches the error φ is added to F or until 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. Next, we illustrate the DASH algorithm on examples.

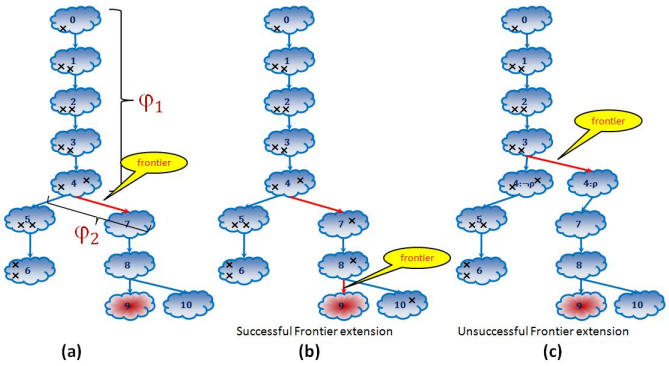

Fig. 2. The role of a frontier in the DASH algorithm.

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

Fig. 3. Simplified example to illustrate DASH.

Handling aliasing. As an example in this section, we use the simple program `alias` shown in Figure 3. This program has three inputs `p`, `p1` and `p2`, all of which are pointers to `int`. At lines `0` and `1`, pointers `p1` and `p2` are compared with `p` and the function returns if `p1` or `p2` alias `p`.

Since there is no aliasing between `p` and the other pointers, the assignment of `1` to `*p` will not affect the value of `*p1` or `*p2`, and the error will not be reached. However, a flow-insensitive alias analysis will miss this because, `p` does alias `p1` and `p2` at later points in the program. 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 paths explored by concrete executions.

DASH first creates the initial abstraction $\Sigma_\simeq$ for the program `alias` that is isomorphic to its control flow graph (shown in Figure 4(a)). As before, we represent regions of the abstraction $\Sigma_\simeq$ as "clouds" and represent states from the forest F using "$\times$"s in the figure. In order to save space, we do not show regions for line numbers `6`, `7` and the exit location of the function. The abstract transitions are shown using solid lines, and the edges of the forest F are shown using dashed lines. DASH performs four refinements of this program as shown in Figure 4. First, the initial forest is created by running `alias` with a random test that assigns values to its inputs `p`, `p1` and `p2`, thus creating a forest $F_{\texttt{alias}}$ of concrete states. Let us suppose that this test created values such that `p1` and `p2` do not alias `p`. Running this test did not result in the error location being reached (there is no $\times$ representing a concrete state in the error region 5).

In the first iteration, DASH finds the (abstract) error path $\tau_e = \langle 0,1,2,3,4,5\rangle$ and the corresponding prefix

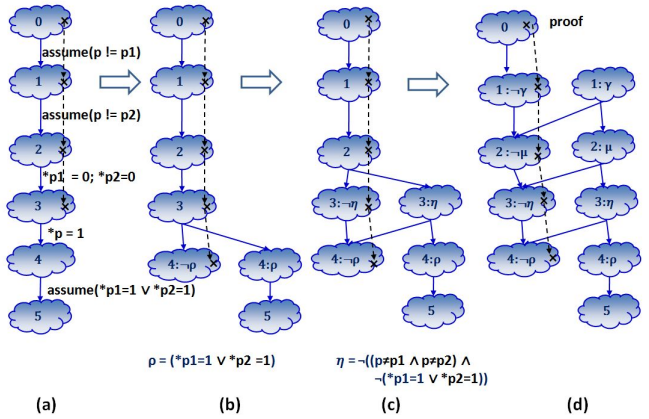

Fig. 4. Abstraction computed by DASH on the example program from Figure 1.

$\tau_e^p = \langle 0, 1, 2, 3, 4 \rangle$ of τ_e as shown in Figure 4(a). DASH now performs symbolic execution for one transition beyond the prefix τ_e^p (that is, the path $\langle 0, 1, 2, 3, 4, 5 \rangle$). The formula obtained from this symbolic execution is infeasible. Thus, this path is determined to be infeasible, and therefore DASH refines region 4 using the predicate $\rho = (*p1 = 1) \vee (*p2 = 1)$, which is computed by the weakest precondition operator WP applied to the assume statement `assume((*p1 ==1) || (*p2 == 1))`.

In the second iteration, DASH examines an (abstract) error path $\tau_e = \langle 0, 1, 2, 3, 4 : \rho, 5 \rangle$ that leads to the error region. It considers the prefix $\tau_e^p = \langle 0, 1, 2, 3 \rangle$ of τ_e that contains concrete states in the forest F, as shown in Figure 4(b). Then, it tries to add a test to $F_{\texttt{alias}}$ that covers the transition $(3, 4 : \rho)$. This also turns out to be not possible, so the DASH algorithm refines region 3. If we were to do this refinement using the WP operator, we note that WP(`*p=1`, $(*p1 = 1) \vee (*p2 = 1)$) has to consider all possible alias combinations between p, $p1$ and $p2$. Instead, DASH uses the WP_α operator (defined in Section 4.2.1) with respect to the alias combination $(p \neq p1 \wedge p \neq p2)$ that occurs by performing symbolic execution along the path executed by the test, and computes the predicate $\eta = \neg((p \neq p1 \wedge p \neq p2) \wedge \neg((*p1 = 1) \vee (*p2 = 1)))$ as shown in Figure 4(c). In comparison, tools like SLAM [9] and BLAST [7] have to consider 4 possible aliasing conditions – $(p = p1 \wedge p = p2)$, $(p = p1 \wedge p \neq p2)$, $(p \neq p1 \wedge p = p2)$ and $(p \neq p1 \wedge p \neq p2)$ in order to be sound. In two more iterations, the abstraction shown in Figure 4(d) is obtained, and since there is no path in the abstraction from the initial region 1 to the error region 5, we have a proof that the program is correct.

The reader might wonder at this point as to how a sound proof can be obtained by considering only the alias combination possible to drive $(p \neq p1 \wedge p \neq p2)$ at line 3. The only reachable alias configuration at region 3 is $(p \neq p1 \wedge p \neq p2)$, and such executions fall inside the region $3{:}\neg\eta$. The other three alias combinations (1) $(p \neq p1 \wedge p = p2)$, (2) $(p = p1 \wedge p \neq p2)$, (3) $(p = p1 \wedge p = p2)$, are grouped inside a single region $3{:}\eta$. However, since $3{:}\eta$ is not reachable by any concrete test, there is no need to

```
void foo(int *p, int *p1, int *p2, ... , int *pn)
{
0:  if(p == p1) return;
1:  if(p == p2) return;
    ...
    ...
    if(p == pn) return;
2:  *p1 = 0; *p2=0; ...; *pn = 0
3:  if (*p1==1 || ... || *pn==1)
4:    error();
5:  p = p1;
    p = p2;
    ...
    p = pn;
}
```

Fig. 5. Aliasing example with n pointers

separately enumerate these three unreachable alias combinations (If indeed one of these aliases were reachable, DASH would be able to drive a test into $3:\eta$ which might result in the region being partitioned further). Thus, WP_α enables DASH to partition the alias space so as to group all the unreachable alias configurations into a small number of regions without having to enumerate them individually.

Moreover, it can be 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 5 parameterized by n (we have verified this by running SLAM and measuring run times as a function of n), whereas DASH does not encounter this blowup since it uses alias information from tests to reason only about the alias combinations that actually happen.

Handling loops. Consider again the example from Figure 1. Similar to our explanation above, DASH can show that the error at line `17` can never be reached while only considering the alias possibility $(lock1 \neq pi \rightarrow lock) \wedge (lock2 \neq pi \rightarrow lock)$ inside the while loop. However, proving that the calls to error() at lines `14`, `22` and `27` are unreachable requires computing the loop invariant at line `25` that `pi->lock` is locked if and only if `pi->x = y`. The detailed explanation of how DASH computes this invariant automatically is tedious, and requires walking through several iterations of the algorithm. For brevity, we give a higher level sketch of how the invariant is computed. At first, DASH considers an abstract error trace that reaches the error at line `14`. Since it is possible to get concrete tests up until the conditional at line `13`, the frontier for this error trace is at line `13`. DASH attempts to extend this test case to line `14` and discovers that such a test case cannot be generated. Thus, it computes WP of the conditional at line `13` and generates the predicate $(*(pi \rightarrow lock) = 0)$. In a subsequent iteration, this predicate is propagated back across the while loop using WP(`assume(x != *(pi->y))`, $(*(pi \rightarrow lock) = 0)$), resulting in the predicate $(x \neq *(pi \rightarrow y) \wedge *(pi \rightarrow lock) = 0)$. After propagating this predicate throughout the loop body, DASH is able to establish that the only paths that re-enter the loop are from the region $(x \neq *(pi \rightarrow y) \wedge *(pi \rightarrow lock) = 0)$ at line `25`. Thus, the program abstraction allows DASH to determine the desired loop invariant and establish that there is no path in the abstraction from the initial region to any error

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

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

Fig. 6. A simple example for interprocedural property checking.

region. In contrast, testing tools like DART or CUTE must handle paths one by one, and hence they will explore only a finite number of paths through loops. Such tools are unable to prove examples such as Figure 1 correct.

Verification tools such as SLAM and BLAST on one hand, and testing tools such as DART, EXE and CUTE on the other hand, have complementary strengths. Verification tools are able to use abstractions to handle loops, obtain high coverage but suffer primarily due to imprecision in alias analysis. In contrast, testing tools are able to use precise alias information in specific paths, but are unable to handle loops and have limited coverage. DASH is able to combine the advantages of these two families of tools.

Interprocedural Property Checking. For programs with several procedures, we describe a modular approach to extend 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 where the DASH algorithm needs to be modified is when the frontier we are trying to extend happens to be a procedure call-return edge (S, T). In such a case, DASH simply invokes itself recursively on the called procedure, by appropriately translating the constraint induced by the prefix τ_e^p into appropriate initial states of the called procedure and translating the predicate on the target region T into appropriate error states of the called procedure.

We explain this through the example in Figure 6, where procedure `top` that makes two calls to an increment procedure `inc`. We show how DASH proves that the call to `error()` (line 3 in `top`) is unreachable.

DASH first creates the abstractions $\Sigma_{\simeq_{\texttt{top}}}$ and $\Sigma_{\simeq_{\texttt{inc}}}$ for the procedures `top` and `inc` respectively (shown in Figure 7(a)). Next, the initial forests are created by running a random test (say $x=2$) 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 7(a)). Since running the test does not result in the error location being reached, DASH examines 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 7(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 combination of symbolic execution and theorem proving [2], [4], that is, a test that covers the edge $(2, 3)$. 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, by applying template-based refinement, DASH refines the partition 2 to two partitions – 2:ρ and 2:$\neg\rho$, and we obtain the abstraction shown in Figure 7(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 state $2:\rho$ 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 refinement of the region 1 with respect to the predicate η (shown in Figure 7(c)). We show how η is computed in Section 4.2.1. After some more iterations, DASH computes the abstraction $\Sigma_{\simeq_{\texttt{top}}}$ (shown in Figure 7(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], [13]. Yorsh, Ball and Sagiv have proposed an approach that involves both abstraction and testing [14]. 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 [15]. 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 [5] also combines testing and abstraction refinement based verification 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. This involves computing abstract summaries for every procedure in the program. Recently, interprocedural extensions to testing tools have been proposed [16], [17]. The extension to DART [16] computes concrete summaries (tests) for every procedure in the program. DASH is an 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

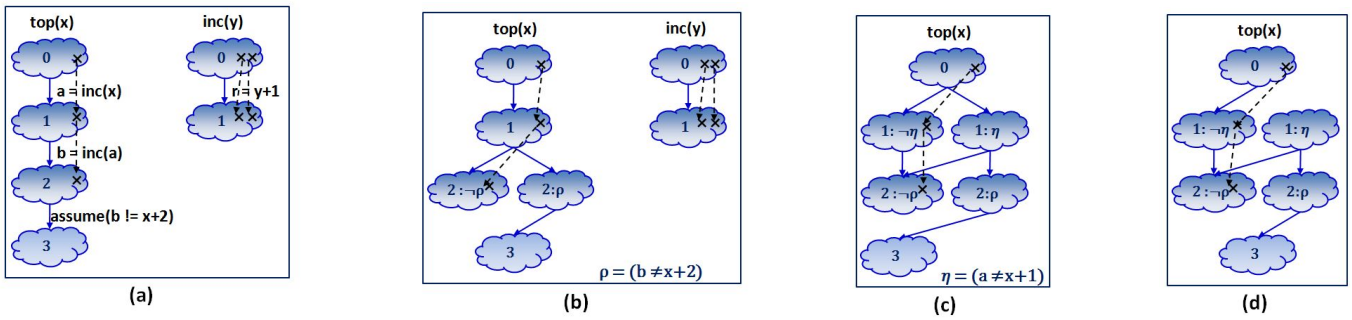

Fig. 7. Abstractions computed by DASH on the example program from Figure 6.

them as summaries in the description of the algorithm). DASH currently does not reuse summaries computed in one context in a different context. This is ongoing work described in [18].

Several methods for doing refinement have been proposed, including backward propagation from error states [19], forward propagation from initial states [9], and using interpolants [10]. 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.

An alternative way to handle aliasing is to model memory as an array, all pointers as indexes into the array, and use the theory of arrays to handle case analysis for aliasing in the theorem prover. This approach is followed by verification tools that are based in verification condition generation such as ESC [20], and BOOGIE [21]. While the theory of arrays is a useful way to handle aliasing for modular local reasoning, our approach is more useful for global reasoning. In order to perform modular local reasoning, one would need other structural ways of constraining the aliases in a program such as ownership models or frame conditions. Since DASH deals with existing C programs or x86 binaries, which have been developed without any constraints on aliasing, such structural ways of constraining aliases are not possible.

Namjoshi and Kurshan [22] 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.

Thomas Ball has suggested the idea of using forward symbolic simulation for pointers together with weakest precondition computation to reduce the number of aliasing predicates generated by SLAM [23]. The idea of WP_α is related in that it uses alias information from tests to reduce the explosion in the number of cases to be considered for weakest precondition computation. However, the design of WP_α is unique to DASH in the sense that we can prove that the algorithm makes progress (see Theorem 1) only if WP_α is applied at the frontier, after a failed test case generation attempt.

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 and (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 structures with fields, and pointers to structures 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 tuple of procedures $\langle P_0, P_1, \ldots, P_n \rangle$, 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. Let Σ be the (possibly infinite) state space of a procedure $P = \langle N, L, E, n_0, \lambda, V \rangle$, defined as the set of all valuations to the variables in V. Every statement $op \in Stmts$ defines a state transition relation $\xrightarrow{op}: \Sigma \times \Sigma$, 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 $\xrightarrow{*}$ 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 $s \xrightarrow{*} s'$, 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 DASH

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, 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
```

Fig. 8. The DASH algorithm.

shown in Figure 8 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.

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 ($parent(s) \rightarrow s$), or $s \in \sigma^I$ and $parent(s) = \epsilon$, and (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. It is important to note that in our experimental evaluation (Section 5) we assume that Test returns the empty set.

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. Symbolic execution along the abstract path to the error region, similar in spirit to CUTE [4], 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. 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* $(k-1, k) \stackrel{\text{def}}{=} \text{Frontier}(S_0, S_1, \ldots, S_n)$ 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 $S_i = \text{Region}(parent^{k-1-i}(s))$ 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 $s \in S$.

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

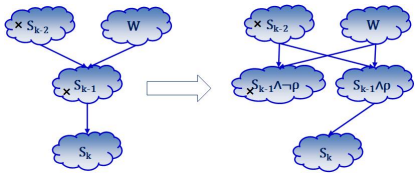

Fig. 9. Refinement split performed by DASH at the frontier.

ExtendFrontier(τ, 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: $\langle \phi_1, \mathcal{S}, \phi_2\rangle$:= ExecuteSymbolic(τ, P)
3: t := IsSAT($\phi_1, \mathcal{S}, \phi_2, P$)
4: **if** $t = \epsilon$ **then**
5: $\quad \rho$:= RefinePred($\mathcal{S}, \tau$)
6: **else**
7: $\quad \rho := true$
8: **end if**
9: return $\langle t, \rho\rangle$

Fig. 10. The auxiliary function ExtendFrontier.

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 10, *is the only function in the* DASH *algorithm that uses a theorem prover*. It takes an ordered trace τ_o 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 $\Sigma_{\simeq}$ at the frontier (lines 21–30), resulting in a split of region S_{k-1} (as shown in Figure 9) 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 [4]. Let $\tau = \langle S_0, S_1, \ldots, S_n\rangle$, and let $(k-1, k) = \text{Frontier}(\tau)$. 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 11. 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

```
ExecuteSymbolic(τ, 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⟩
```

Fig. 11. The auxiliary function ExecuteSymbolic.

value for $*v$ (line 2 in Figure 11) 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 10) 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 refining $\Sigma_{\simeq}$ at S_{k-1} according to the template in Figure 9. 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 past the frontier, then RefinePred should return a predicate that is "good" in some sense. We seek a predicate ρ that is suitable to perform a refinement as shown in Figure 9. We require such a predicate ρ, to satisfy the two conditions formally stated below.

*Definition 1 (*Suitable predicate*):* Let τ be an abstract error trace and let (S, S') be its frontier. A predicate ρ is said to be suitable with respect to τ if: (1) all possible concrete states obtained by executing τ up to the frontier belong to the region $(S \wedge \neg\rho)$, and (2) there is no transition from any state in $(S \wedge \neg\rho)$ to a state in S'.

A refinement in the style of Figure 9 makes *progress* in the sense that it eliminates the current abstract trace $S_0, S_1, \ldots, S_{k-1}, S_k$. This is because every state that can be reached by the prefix $S_0, S_1, \ldots, S_{k-1}$ needs to be in the region $(S_{k-1} \wedge \neg\rho)$ and there is no abstract transition from the region $(S_{k-1} \wedge \neg\rho)$ to S_k, by the above definition. We call this *template-based refinement* since it is done without any calls to a theorem prover after computing a suitable predicate. 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 $(x, x+1) =$ Frontier(τ) and $(y, y+1) =$ Frontier(τ'), 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 9), 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, 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 9) without any calls to a theorem prover after computing a suitable predicate. Next, we 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 WP(op, ϕ) denote the *weakest precondition* [8] of ϕ with respect to statement op. WP(op, ϕ) is defined as the weakest predicate whose truth before op implies the truth of ϕ after op executes. The weakest precondition WP$(\mathtt{x} = \mathtt{e}, \phi)$ is the predicate obtained by replacing all occurrences of x in ϕ (denoted $\phi[e/x]$). For example, WP$(\mathtt{x} = \mathtt{x}+1, x < 1) = (x+1) < 1 = (x < 0)$. However, in the case of pointers, WP(op, ϕ) is not necessarily $\phi[e/x]$. For example, WP$(\mathtt{x} = \mathtt{x} + 1, *p + *q < 1)$ is not $*p + *q < 1$, if either $*p$ or $*q$ or both alias x. In order to handle this, if the predicate ϕ mentions k locations[2] (say $y_1, y_2, \ldots, y_k$), then WP$(\mathtt{x} = \mathtt{e}, \phi)$ would

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

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

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

Fig. 12. Computing suitable predicates.

have 2^k disjuncts, with each disjunct corresponding to one possible alias condition of the k locations with x [6]. Therefore, $\mathsf{WP}(\mathtt{x} = \mathtt{x} + 1, *p + *q < 1)$ 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 3, 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 the current abstract trace, and computes the weakest precondition specialized to that aliasing condition, as shown by the function RefinePred in Figure 12. The aliasing condition α is allowed to have disjunctions, though this happens rarely in practice.

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

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

It is important to note that the α computed by the auxiliary function $\mathsf{Aliases}(\mathcal{S}, op, S_k)$ consists of only those aliasing conditions in $\mathcal{S}$ that hold between locations that occur in op and S_k. For efficiency, $\mathsf{WP}{\downarrow_\alpha}(op, \phi))$ can be computed by only considering the aliasing condition 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

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

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. Since it is not possible to generate a test that extends the frontier, it must be the case that $\forall c \in \mathcal{C}, c \notin \mathsf{WP}{\downarrow_\alpha}\,(op, \phi_2)$ (since every path results in exactly one α). This implies that $\forall c \in \mathcal{C}, c \in (\alpha \wedge \neg \mathsf{WP}{\downarrow_\alpha}\,(op, \phi_2))$. Therefore $\mathcal{C} \cap \neg(\alpha \wedge \neg \mathsf{WP}{\downarrow_\alpha}(op, \phi_2)) = \emptyset$, and so the predicate $\mathsf{WP}_\alpha(op, \phi_2)$ satisfies the first half of the definition.

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. □

4.2.2 Extensions

We note that while WP or interpolants [10] are other possible choices for a suitable predicate for the refinement shown in Figure 9, the predicate computed by both these techniques contain 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 [5] and [22]. Though we consider only WP, we believe that a similar optimization to reduce the number of aliasing possibilities using tests can also be done with interpolants.

Even though our current intermediate form for C does not support arrays, the DASH algorithm easily generalizes to handle arrays. With arrays, the aliasing condition α that results from symbolic execution of the abstract trace can have disjuncts, since an assignment that accesses an array using an index requires a case analysis over the values taken by the index. However, the calculation of suitable predicates using the WP_α operator works even if the alias condition α has disjuncts, and the guarantees offered by Theorem 3 continue to hold.

4.3 Soundness and Complexity

In this section we will present theoretical results that characterize the correctness and complexity of the DASH algorithm. The first lemma states that DASH is sound – that is, every error and proof found by DASH is a valid one.

*Lemma 1 (*Soundness): If DASH terminates on (P, φ), then either of the following is true:

- If DASH returns ("pass", $\Sigma_\simeq$), then $\Sigma_\simeq$ is a proof that P cannot reach φ.
- If DASH returns ("fail", t), then t is a test for P that violates φ.

Proof: If DASH returns ("pass", $\Sigma_\simeq$), it follows from Corollary 2 and Theorem 3 that $P_\simeq = \langle \Sigma_\simeq, {\sigma^I}_\simeq, \rightarrow_\simeq \rangle$ simulates the program P with respect to the property φ and thus is a proof that P cannot reach φ. On the other hand, since DASH returns ("fail", t) only if there is a concrete witness in the region φ, t is a test that violates φ. □

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)

Fig. 13. The DASH algorithm for programs with multiple procedures.

Though we cannot bound the number of iterations of DASH we can bound the number of theorem prover calls made in each iteration.

*Lemma 2 (*Complexity): The DASH algorithm makes precisely one theorem prover call per iteration.

Proof: 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. □

We note that the above theorem also holds for multiple procedures. In programs with multiple procedures, if the frontier happens to be at a procedure call, DASH is recursively invoked on the called procedure, and this recursive invocation can result in several iterations, each entailing one theorem prover call. Next, we discuss the how DASH can be extended to handle multiple procedures.

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 13) calls the function DASH from Figure 8 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 14. Informally, the interprocedural algorithm works by recursively invoking DASH whenever the standard algorithm dictates that the frontier must be extended across a call-return edge of the graph. The results of recursive call, combined with information from the calling context tell us whether or not there exists a test that can extend the frontier. If this is not possible, then the proof returned by the recursive DASH call is used to compute a suitable predicate.

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), so that it is really a trace of every abstract program point through which the test passed. If Edge(S_i, S_{i+1}) is a call-return edge that occurs before the frontier, GetWholeAbstractTrace runs a test t (obtained from the concrete witness in S_i) on the called procedure GetProc(e) and replaces Edge(S_i, S_{i+1}) with the sequence of regions corresponding to the test t.

ExtendFrontier(τ, P)
Returns:
(t, $true$), if the frontier can be extended; or
(ϵ, ρ), if the frontier cannot be extended.

1: $\tau_w = \langle S_0, S_1, \ldots, S_n \rangle$:= GetWholeAbstractTrace(τ, F)
2: $(k-1, k)$:= Frontier(τ_w)
3: $\langle \phi_1, \mathcal{S}, \phi_2 \rangle$:= ExecuteSymbolic(τ_w, P)
4: **if** Edge$(S_{k-1}, S_k) \in$ CallReturn(E) **then**
5: let $\langle \Sigma, \sigma^I, \rightarrow \rangle$ = GetProc(Edge(S_{k-1}, S_k)) in
6: ϕ := InputConstraints($\mathcal{S}$)
7: ϕ' := $S_k[e/x]$
8: $\langle r, m \rangle$:= DASH($\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg\phi'$)
9: **if** r = "fail" **then**
10: $t := m$
11: $\rho := true$
12: **else**
13: ρ := ComputeRefinePred(m)
14: $t := \epsilon$
15: **end if**
16: **else**
17: t := IsSAT($\phi_1, \mathcal{S}, \phi_2, P$)
18: **if** $t = \epsilon$ **then**
19: ρ := RefinePred($\mathcal{S}, \tau_w$)
20: **else**
21: $\rho := true$
22: **end if**
23: **end if**
24: return $\langle t, \rho \rangle$

Fig. 14. The auxiliary function ExtendFrontier for interprocedural analysis.

The function ExecuteSymbolic (line 3) performs symbolic execution on the whole abstract error trace τ_w as described in Figure 11. 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$(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg\phi')$ 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. It can be easily seen that the predicate ϕ is a suitable predicate and this is analogous to using the strongest postcondition for refinement. On the other hand, a suitable predicate analogous to the intraprocedural WP$_\alpha$ predicate can be

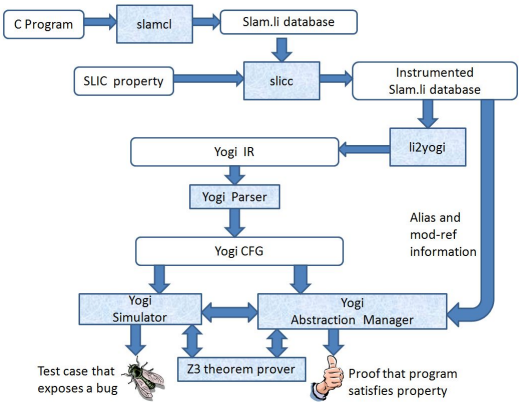

Fig. 15. Architecture of YOGI.

```
state {
   enum { Unlocked=0, Locked=1 }
     state = Unlocked;
}

KeAcquireSpinLock.return {
  if (state == Locked)
    abort;
  else
    state = Locked;
}

KeReleaseSpinLock.return {
  if (state == Unlocked)
    abort;
  else
    state = Unlocked;
}
```

Fig. 16. An example SLIC specification.

gleaned from the way DASH splits the initial region. This predicate is computed by the auxiliary function ComputeRefinePred in line 13 which takes the proof m computed by DASH and returns a suitable predicate ρ. Specifically, ComputeRefinePred(m) is defined as follows.

$$\text{ComputeRefinePred(m)} \stackrel{\text{def}}{=} \neg(\bigvee \rho_i)$$

where each ρ_i is a predicate in the proof m used to split the initial region $\sigma^I \wedge \phi$. It can be shown that ComputeRefinePred returns a suitable predicate.

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

5 EVALUATION

We have implemented DASH in a tool called YOGI which has been integrated with Microsoft's Static Driver Verifier (SDV) framework. As shown in Figure 15, YOGI takes two inputs: (1) a C program, and (2) a safety property specified in the SLIC specification language [24]. A sample SLIC specification for a locking protocol (`KeAcquireSpinLock` and `KeReleaseSpinLock` occur in strict alternation) is shown in Figure 16. YOGI uses SLAM's front-end (called slamcl) to parse C programs and SLAM's property instrumentor (called slicc) to instrument the property into the program. The resulting program with the property instrumented is in SLAM's internal binary format called li. We have developed a translator called LI2YOGI that converts the li format to YOGI's intermediate form called yogi-ir. The yogi-ir is a textual format that represents the program at the level of basic blocks with instructions. Each instruction is one of three types: an assignment, assume statement or a procedure call. Once a program has been converted to the yogi-ir format, it is read by the YOGIPARSER to produce an internal interprocedural control flow graph. SLAM's front end merges arrays to a single representative location – every access `a[i]` is approximated to `*a` ignoring the value of `i`. Consequently, YOGI also inherits this approximation.

The two main components of YOGI are: (1) YSIM, a simulator which can perform both concrete execution with concrete values and symbolic execution, and (2) YABSMAN, an abstraction manager, which manages proofs.

The YSIM simulator code is polymorphic over the type of the values it operates. Thus, the same simulation code does both concrete and symbolic execution. During concrete execution, the simulator uses a model of memory where concrete values of appropriate type are stored in locations. During symbolic execution, the simulator stores symbols and formulas in locations. It uses the Z3 theorem prover [25] to reason about consistency of the formulas and to generate test cases as satisfiable models of formulas. All predicates are propositional logic formulae defined over linear arithmetic and uninterpreted functions stored in Z3's internal representation. With these assumptions, the logic is decidable and therefore every satisfiability/validity query to the theorem prover results in either a "yes" or "no" answer. The YABSMAN abstraction manager maintains a region graph abstraction of the program.

The implementation of DASH is very close to the description in Section 4 with some additional optimizations described below.

(a) YSIM keeps track of which input locations are "touched" when symbolically executing an abstract path so that only these touched locations are initialized before running a test. This simple optimization greatly reduces the time to set up a test especially for large programs

(b) When faced with an if-branch in a program, DASH will perform an inexpensive test to see whether the WP_α of a weaker predicate, one that ignores the branch condition, still satisfies the template described in Figure 9. This can be done by evaluation, and does not require a theorem prover call. The effect of this optimization is that we avoid getting "stuck" in irrelevant

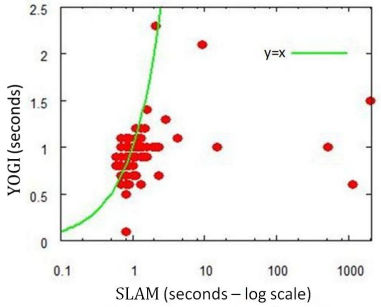

Fig. 17. Scatter plot of the relative runtimes of SLAM and YOGI on 95 C programs in SLAM's regression suite.

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 two sets of evaluations to compare YOGI and SLAM[3].

Device driver benchmarks. We have tested YOGI with SDV's integration pass suite which consists of 69 Windows Vista device drivers and 85 properties, a total of 5865 driver-property pairs. The largest driver in this suite has over 30K lines of code and the cumulative size of all drivers is over 300K lines of code. Currently, YOGI runs on 95% of the runs of the integration test pass and it is able to prove properties as well as find bugs in drivers. There are 129 runs where SLAM either times or spaces out where YOGI is able to give a result. The total time taken by YOGI on all the 5865 runs is about 32 hours on a 4 core machine, whereas SLAM takes around 69 hours.

A comparison of YOGI with SLAM on 16 representative drivers is shown in Table 1. Every row of this table shows the driver, its number of lines of code, the number of properties checked and the time (in minutes) taken by SLAM and YOGI along with the total number of iterations and the number of time-outs (set to 30 minutes). In many of the cases where SLAM either times out or gives up due to pointer aliasing, YOGI 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 YOGI using WP_α considers only the aliasing possibilites that occur along test executions. In fact, the simplified code snippet in Figure 1 in Section 2 was motivated by looking at device driver code and simplifying it for presentation. In the other runs where SLAM produces a result (proof/bug), YOGI also reports an identical result. As seen in the table, even though YOGI takes several more iterations when compared to SLAM, each iteration is very efficient, and the overall runtime of YOGI 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. There are also some checks where we observe that SLAM is faster than YOGI. It turns out that for all these checks, the proof can be computed immediately by abstraction and therefore SLAM is able to efficiently compute it. On the other hand, YOGI incurs additional overheads due to test execution.

SLAM regression suite. We ran YOGI on 95 C programs in SLAM's regression suite. These are simple C programs which contain common features of device drivers where the average size per program is 100 lines of code. A scatter plot of the relative runtimes of SLAM and YOGI can be seen in Figure 17. SLAM and YOGI 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 YOGI is faster than SLAM. The total time taken by SLAM for all the 95 programs (put together) is 20 minutes, whereas YOGI finishes all the 95 programs in 17 seconds. With test caching enabled (where tests are reused across runs of YOGI), YOGI finishes all the 95 programs in 4 seconds.

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 [26], and check termination properties [27]. 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.

We are continuing to improve scalability of YOGI in several ways. We have come up with notions of summarizing the results of analyzing a procedure using both concrete summaries ('must' summaries) and abstract summaries (or 'may' summaries). Initial empirical results show another order of magnitude improvement in YOGI's performance using such summaries [18]. In future work, we plan to extend YOGI to handle arrays (see Section 4.2.2). We also plan to improve scalability even more by using approximate symbolic execution,

3. We used SDV's latest version of SLAM that is shipped with Microsoft Windows Vista.

Program	Lines	Properties	SLAM			YOGI		
			Time-outs	Iterations	Time (min)	Time-outs	Iterations	Time (min)
parport	34196	19	1	577	**91.2**	0	8449	**26.1**
serial1	32385	21	3	288	**142.4**	0	16716	**21.5**
serial	31861	21	3	244	**203.9**	0	20135	**28.1**
fdc_fail	9251	50	0	676	**117.6**	0	9675	**8**
kbdclass1	7426	38	2	574	**124.9**	0	31846	**115**
kbdclass	7132	36	2	552	**125.5**	0	26572	**90.4**
serenum	6011	38	1	327	**95.6**	0	7198	**10.9**
pscr	5680	37	0	564	**55**	0	14126	**26.4**
modem	3467	19	0	246	**18**	0	9565	**22.3**
1394Vdev	2757	22	2	434	**90.7**	0	52240	**72.9**
1394Diag	2745	23	3	430	**121.4**	0	45031	**68.8**
diskperf	2351	31	0	224	**36.8**	1	16948	**100**
incomplete1	1558	29	0	120	**16**	0	6305	**6.3**
toastmon1	1539	32	0	119	**13.5**	0	7853	**8.4**
toastmon	1505	32	0	119	**16.6**	0	7619	**7.6**
daytona	565	29	1	514	**106.9**	0	14938	**77.4**

TABLE 1
Empirical evaluation of YOGI on 16 device drivers.

and by persisting may summaries and must summaries for each procedure and reusing them across multiple runs of YOGI.

ACKNOWLEDGMENTS

The authors would like to thank Thomas Ball, Patrice Godefroid, Akash Lal, James Larus, Rustan Leino, Kanika Nema, G. Ramalingam and Thomas Reps for comments on earlier drafts of this paper. They would also like to thank Nikolaj Bjorner and Leonardo de Moura for providing the Z3 theorem prover used by YOGI.

REFERENCES

[1] E. W. Dijkstra, "The humble programmer," *Commun. ACM*, vol. 15, no. 10, pp. 859–866, 1972.
[2] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing." in *PLDI '05: Programming Language Design and Implementation*. ACM Press, 2005, pp. 213–223.
[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *ACM CCS*, 2006.
[4] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *FSE '05: Foundations of Software Engineering*. ACM Press, 2005, pp. 263–272.
[5] 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*. ACM Press, 2006, pp. 117–127.
[6] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *PLDI '01: Programming Language Design and Implementation*. ACM Press, 2001, pp. 203–213.
[7] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL '02: Principles of Programming Languages*. ACM Press, 2002, pp. 58–70.
[8] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1997.
[9] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," in *SPIN '01: SPIN workshop on Model checking of Software*. Springer-Verlag New York, Inc., 2001, pp. 103–122.
[10] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *POPL '04: Principles of Programming Languages*. ACM Press, 2004, pp. 232–244.
[11] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons, "Proofs from tests," in *ISSTA '08: International Symposium on Software Testing and Analysis*. ACM Press, 2008, pp. 103–122.
[12] P. Godefroid and N. Klarlund, "Software model checking: Searching for computations in the abstract or the concrete," in *IFM '05: Integrated Formal Methods*, 2005, pp. 20–32.
[13] E. Gunter and D. Peled, "Model checking, testing and verification working together," *Form. Asp. Comput.*, vol. 17, no. 2, pp. 201–221, 2005.
[14] G. Yorsh, T. Ball, and M. Sagiv, "Testing, abstraction, theorem proving: better together!" in *ISSTA '06: International Symposium on Software Testing and Analysis*. ACM Press, 2006, pp. 145–156.
[15] D. Kroening, A. Groce, and E. M. Clarke, "Counterexample guided abstraction refinement via program execution," in *ICFEM '04: International Conference on Formal Engineering Methods*, ser. Lecture Notes in Computer Science, 2004, pp. 224–238.
[16] P. Godefroid, "Compositional dynamic test generation," in *POPL '07: Principles of Programming Languages*. ACM Press, 2007, pp. 47–54.
[17] R. Majumdar and K. Sen, "LATEST : Lazy dynamic test input generation," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-36, March 2007. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-36.html
[18] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali, "Compositional may-must program analysis: Unleashing the power of alternation," in *POPL '10: Principles of Programming Languages*. ACM Press, 2010.
[19] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV '00: Computer Aided Verification*. Springer-Verlag, 2000, pp. 154–169.
[20] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe, "Extended static checking," Compaq Systems Research Center, Tech. Rep. Research Report 159, December 1998.
[21] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, ""Boogie: A modular reusable verifier for object-oriented programs"," in *FMCO '05: Formal Methods for Components and Objects*, ser. LNCS 4111. Springer-Verlag, pp. 364–387.
[22] K. S. Namjoshi and R. P. Kurshan, "Syntactic program transformations for automatic abstraction," in *CAV '00: Computer Aided Verification*. Springer-Verlag, 2000, pp. 435–449.
[23] T.Ball, "Personal communication."
[24] T. Ball and S. K. Rajamani, "SLIC: A specification language for interface checking of C," Microsoft Research, Tech. Rep. MSR-TR-2001-21, 2001.
[25] 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.
[26] S. Qadeer and D. Wu, "KISS: Keep it simple and sequential," in *PLDI '04: Programming Language Design and Implementation*. ACM, 2004, pp. 14–24.
[27] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *PLDI '06: Programming Language Design and Implementation*. ACM, 2006, pp. 415–426.