

SYNERGY: A New Algorithm for Property Checking

Bhargav S. Gulavani
IIT Bombay
bhargav@cse.iitb.ac.in

Thomas A. Henzinger
EPFL
tah@epfl.ch

Yamini Kannan
Microsoft Research India
yaminik@microsoft.com

Aditya V. Nori
Microsoft Research India
adityan@microsoft.com

Sriram K. Rajamani
Microsoft Research India
sriram@microsoft.com

ABSTRACT

We consider the problem if a given program satisfies a specified safety property. Interesting programs have infinite state spaces, with inputs ranging over infinite domains, and for these programs the property checking problem is undecidable. Two broad approaches to property checking are *testing* and *verification*. Testing tries to find inputs and executions which demonstrate violations of the property. Verification tries to construct a formal proof which shows that all executions of the program satisfy the property. Testing works best when errors are easy to find, but it is often difficult to achieve sufficient coverage for correct programs. On the other hand, verification methods are most successful when proofs are easy to find, but they are often inefficient at discovering errors. We propose a new algorithm, SYNERGY, which combines testing and verification. SYNERGY unifies several ideas from the literature, including counterexample-guided model checking, directed testing, and partition refinement. This paper presents a description of the SYNERGY algorithm, its theoretical properties, a comparison with related algorithms, and a prototype implementation called YOGI.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification; D.2.5 [Software Engineering]: Testing and Debugging; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms

Testing, Verification

Keywords

Software model checking; Directed testing; Abstraction refinement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.
Copyright 2006 ACM 1-59593-468-5/06/0011 ...\$5.00.

1. INTRODUCTION

Automated tools for software verification have made great progress over the past few years. We can broadly classify these tools into two categories (the boundaries are not sharp, but we still find the classification useful). The first class of verification tools searches for bugs. These are tools that execute the program in one form or another. At one extreme, in program testing, the program is executed concretely on many possible inputs. Such test inputs may be either generated manually, by employing testers, or generated automatically using tools (see [9] for a survey on automatic test-case generation). At the other extreme, the program is executed abstractly, by tracking only a few facts during program execution [4, 10]. These tools are very efficient and therefore widely used. They have different strengths and weaknesses: testing finds real errors but it is difficult to achieve good coverage; abstract execution, if done statically, can cover all program paths but signals many false positives (potential errors that are not real). As a result, many intermediate solutions have been pursued, such as directed testing [13]. In this approach, the program is executed symbolically, by collecting all constraints along a path. This information is then used to drive a subsequent test into a desired branch off the original path. For concurrent programs, a practical technique to increase testing coverage is to systematically explore different interleavings by taking control of the scheduler [11].

The second class of verification tools searches for proof of the absence of bugs. These are tools that try to find a safe “envelope” of the program, which contains all possible program executions and is error-free. Also this class contains a diverse set of methods. At one extreme, in classical model checking [7], the exact envelope of a program is constructed as the reachable state space. At the other extreme, in deductive verification [24], a suitable program envelope is an inductive invariant. While computing the exact envelope proceeds automatically, the computation rarely terminates. On the other hand, user intervention is usually required to define a suitable overapproximate envelope, say, in the form of loop invariants, or in the form of abstraction functions. These inefficiencies, due to state explosion and the need for user guidance, have prevented the wide adoption of proof-based tools. Again, recent approaches try to address these issues. For instance, in counter-example guided abstraction refinement [3, 5, 6, 17, 22], the search for a safe program envelope is automated by iteratively refining a quotient (partition) of the reachable state space. Of course, the execution-based tools mentioned earlier may also produce proofs, for

example, if complete coverage can be ensured by a test-case generator, or if no potential errors (either real nor false) are reported by an abstract interpretation; but it is rather exceptional when this happens. Conversely, the proof-based tools may report bugs, but they generally do so only as a by-product of an expensive, failed search for proof. Not surprisingly, therefore, several recent papers have predicted that testing and verification can be combined in interesting ways [12, 15].

We present a new verification algorithm, called SYNERGY, which searches simultaneously for bugs and proof, and while doing so, tries to put the information obtained in one search to the best possible use in the other search. The search for bugs is guided by the proof under construction, and the search for proof is guided by the program executions that have already been performed. SYNERGY, thus, is a combination of underapproximate and overapproximate reasoning: program execution produces a successively more precise underapproximation of the reachability tree of the program, and partition refinement produces a successively more precise overapproximation. Specifically, SYNERGY guides testing towards errors; it can be viewed as “property-directed testing”: if some parts of a program can be proved safe, then SYNERGY focuses subsequent tests on other program parts. In partition refinement [20, 23], on the other hand, the biggest practical difficulty has been to decide where and how to refine an insufficiently precise abstraction. SYNERGY uses test information to make that decision. This is particularly effective during long deterministic stretches of program execution, such as *for* loops. While proof-based tools [3, 5, 17] may perform as many refinement steps as there are loop iterations, an inexpensive, concrete execution of the loop can immediately suggest the one necessary refinement (see Example 3 below). SYNERGY, therefore, performs better than the independent use of both execution-based and proof-based tools. We are not the first to combine concrete and abstract program execution. SYNERGY bears some resemblance to the LEE-YANNAKAKIS algorithm [23], which recently has been suggested also for software verification [27]. This algorithm constructs a bisimulation quotient of the reachable state space by simultaneous partition refinement and concrete execution, to see which abstract states (equivalence classes) are reachable. However, there are important theoretical and practical differences between SYNERGY and LEE-YANNAKAKIS. On the theoretical side, we show that SYNERGY constructs a *simulation* quotient of the program, not a bisimulation quotient. This is significant, because simulation is a coarser relation than bisimulation, and therefore proofs constructed by SYNERGY are smaller than proofs constructed by LEE-YANNAKAKIS. In fact, we give an example for which SYNERGY terminates with a proof, whereas LEE-YANNAKAKIS does not terminate, because the program has no finite bisimulation quotient. On the practical side, LEE-YANNAKAKIS performs concrete program executions only to avoid the refinement of unreachable abstract states; it neither guides concrete executions towards errors, nor does it use concrete executions to guide the refinement of reachable abstract states. SYNERGY, in contrast, typically collects many tests—even many that visit the same abstract states—between any two refinements of the partition. This is important, because tests are less expensive than refinement (which involves theorem prover calls), and they give valuable information for choosing the next refinement.

Yorsh, Ball, and Sagiv have recently proposed another approach that involves both abstraction and testing [29]. If there are abstract counterexamples, they fabricate new concrete states along the abstract counterexamples as a heuristic to increase the coverage of testing. They are also able to detect when the current program abstraction is a proof. Unlike in SYNERGY, no refinement algorithm is provided. Kroening, Groce, and Clarke have proposed using concrete program execution to perform abstraction refinement [21]. Their refinement technique is based on partial program simulation using SAT solvers. Unlike their approach, SYNERGY uses tests to choose the frontier of abstract counterexamples, and tries to either extend this frontier by directed testing, or refine the abstraction at the frontier.

This paper presents a description of the SYNERGY algorithm, its theoretical properties (soundness and termination), a comparison with related algorithms, and a prototype implementation in a tool called YOGI. The implementation works currently for single-procedure C programs with integer variables, and checks safety properties that are specified by invoking a special `error()` function. Even with this limited expressiveness, we are able to demonstrate the effectiveness of SYNERGY over existing algorithms for iterative partition refinement.

2. OVERVIEW

We informally present the SYNERGY algorithm on an example that is difficult for SLAM-like [3, 17] tools. Consider the program from Figure 1. Given an integer input `a`, the program executes the body of the while loop 1000 times, incrementing the loop counter `i` each time without modifying `a`. After the loop, the variable `a` is checked for being positive, and if the check fails, the error location `6` is entered. Clearly, the program reaches the error iff the input `a` is zero or negative. A counterexample-guided partition refinement tool based on predicate abstraction will, in this example, discover the 1000 predicates `(i==0)`, `(i==1)`, `(i==2)`, ..., `(i==999)` one by one before finding the path to the error. This is because every abstract error trace that executes the loop body less than 1000 times is infeasible (i.e., does not correspond to a concrete error trace), and to prove each infeasibility, a new predicate on the loop counter needs to be added. Directed testing, by contrast, performs well on this example. If a test input `a` with `(a>0)` is chosen, the test will not pass the assumption at location `5`. Then, a DART-like [13] tool will suggest a subsequent test with `(a<=0)` in order to pass that assumption. That test, of course, will hit the error.

We will see that, on this example, SYNERGY quickly finds the error by performing a DART-like underapproximate analysis. On other examples (such as the examples from Figures 3 and 8 below), where SLAM succeeds quickly in finding a proof, SYNERGY does so as well, by performing a SLAM-like overapproximate analysis. In particular, DART works well for deterministic loops (since there is only one path, which a test can cover easily, but a large number of abstract states along the path, which may take many iterations for iterative refinement to generate); and SLAM works well for sequences of branches (such as Figure 8, since there are a small number of abstract states for SLAM to generate, but a large number of paths for DART to enumerate). However, typical programs have both loops and sequences of branches, and SYNERGY works better on such programs than running SLAM or DART

```

void foo(int a)
{
    int i, c;
0:   i = 0;
1:   c = 0;
2:   while (i < 1000) {
3:       c = c + i;
4:       i = i + 1;
    }
5:   assume(a <= 0);
6:   error();
}

```

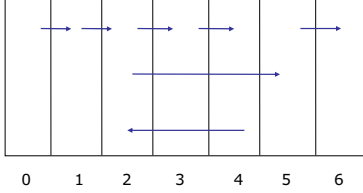


Figure 1: Example on which SLAM does not work well

in isolation. In fact, SYNERGY often performs better than running DART and SLAM independently in parallel, because in SYNERGY, the two analyses communicate information to each other.

SYNERGY maintains two data structures. For the underapproximate (concrete) analysis, SYNERGY collects the test runs it performs as a forest F . Each path in the forest F corresponds to a concrete execution of the program. The forest F is grown by performing new tests. As soon as an error location is added to F , a real error has been found. For the overapproximate (abstract) analysis, SYNERGY maintains a finite, relational abstraction A of the program. Each state of A is an equivalence class of concrete program states, and there is a transition from abstract state a to abstract state b iff some concrete state in a has a transition to some concrete state in b . Initially, A contains one abstract state per program location. For example, in Figure 1, there are 7 abstract states, one per program location, with location 6 being the error location. The partition A is repeatedly refined by splitting abstract states. As soon as the error location becomes unreachable in the abstract program A , a proof has been found (we will show in Section 5 that the abstract program A always simulates the concrete program).

SYNERGY grows the forest F by looking at the partition A , and it refines A by looking at F . Whenever there is an (abstract) error path in A , SYNERGY chooses an error path τ_{err} in A which has a prefix τ such that (1) τ corresponds to a (concrete) path in F , and (2) no abstract state in τ_{err} after the prefix τ is visited in F . Such an “ordered” path τ_{err} always exists. SYNERGY now tries to add to F a new test which follows the ordered path τ_{err} for at least one transition past the prefix τ . We use directed testing [13] to check if such a “suitable” test exists. If a suitable test exists, then it has a good chance of hitting the error if the error is indeed reachable along the ordered path. And even if the suitable test does not hit the error, it will indicate a longer feasible prefix of the ordered path. On the other hand, if a suitable test does not exist, then instead of growing the

forest F , SYNERGY refines the partition A by removing the first abstract transition after the prefix τ along the ordered path τ_{err} . This transition of τ_{err} from, say, a to b can always be removed by refining the abstract state a into $a \setminus \text{Pre}(b)$, where Pre is the preimage (weakest precondition) operator (defined in Section 3). Then SYNERGY continues by choosing a new ordered path, until either F finds a real program error or A provides a proof of program correctness.

On the example from Figure 1, the first ordered path found by SYNERGY is the abstract error trace corresponding to the program locations 0, 1, 2, 5, 6. Since the forest F is initially empty, SYNERGY adds some test that proceeds from location 0 to 1 along the ordered path (and possibly much further). Say the first such test input is $(a==45)$. This test produces a concrete path that executes the loop body 1000 times and then proceeds to location 5 (but not to 6). At this point, F contains this single path, and A still contains one abstract state per program location. This is now the point at which SYNERGY crucially deviates from previous approaches [12, 15, 21, 29]: the new ordered path that SYNERGY returns executes the loop body 1000 times before proceeding to locations 5 and 6. This is because all shorter abstract error paths (which contain fewer loop iterations) have no maximally corresponding prefix in F : consider an abstract path τ'_{err} to 5 and 6 with less than 1000 loop iterations; since 5 is visited by F , but no path in F corresponds to the prefix of τ'_{err} until 5, the path τ'_{err} is not ordered. Once the ordered path with 1000 loop iterations is chosen, the next suitable test is one that passes from 5 to 6. Say the second test input is $(a==5)$. This second test reaches the error, thus showing the program unsafe.

3. ALGORITHM

A program P is a triple $\langle \Sigma, \sigma^I, \rightarrow \rangle$, where Σ is a (possibly infinite) set of states, $\sigma^I \subseteq \Sigma$ is a set of initial states, and $\rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation. We use \rightarrow^* to denote the reflexive-transitive closure of \rightarrow . A property $\psi \subseteq \Sigma$ 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, ψ) . The answer to the property checking problem is “fail” if there is some initial state $s \in \sigma^I$ and some error state $t \in \psi$ such that $s \rightarrow^* t$, and “pass” otherwise.

We desire to produce witnesses for both “fail” and “pass” answers. A witness to “fail” is an *error trace*, that is, a finite sequence s_0, s_1, \dots, s_n of states such that (1) $s_0 \in \sigma^I$, and (2) $s_i \rightarrow s_{i+1}$ for $0 \leq i < n$, and (3) $s_n \in \psi$. A witness to “pass” is a finite-indexed partition Σ_{\simeq} of the state space Σ which proves the absence of error traces. Such proofs are constructed using abstract programs. Given an equivalence relation \simeq on Σ with finitely many equivalence classes, we define the abstract program $P_{\simeq} = \langle \Sigma_{\simeq}, \sigma^I_{\simeq}, \rightarrow_{\simeq} \rangle$ such that (1) Σ_{\simeq} is the set of equivalence classes of \simeq in Σ ; and (2) $\sigma^I_{\simeq} = \{S \in \Sigma_{\simeq} \mid S \cap \sigma^I \neq \emptyset\}$ is the set of equivalence classes that contain initial states; and (3) $S \rightarrow_{\simeq} T$, for $S, T \in \Sigma_{\simeq}$, iff there exist two states $s \in S$ and $t \in T$ such that $s \rightarrow t$. We use the term “regions” to denote the equivalence classes in Σ_{\simeq} . We use the notation ψ_{\simeq} to denote the regions that intersect with ψ ; formally, $\psi_{\simeq} = \{S \in \Sigma_{\simeq} \mid S \cap \psi \neq \emptyset\}$. An *abstract trace* is a sequence S_0, S_1, \dots, S_n of regions such that (1) $S_0 \in \sigma^I_{\simeq}$, and (2) $S_i \rightarrow_{\simeq} S_{i+1}$ for all $0 \leq i < n$. The abstract trace is an abstract error trace if, in addition, (3) $S_n \in \psi_{\simeq}$. The finite-indexed partition Σ_{\simeq} is a *proof*

SYNERGY($P = \langle \Sigma, \sigma^I, \rightarrow \rangle, \psi$)

Assumes: $\sigma^I \cap \psi = \emptyset$.

Returns:

(“fail”, t), where t is an error trace of P reaching ψ ; or
 (“pass”, Σ_{\simeq}), where Σ_{\simeq} is a proof that P cannot reach ψ .

```

1:  $F := \emptyset$ 
2:  $\Sigma_{\simeq} := \{\sigma^I, \psi, \Sigma \setminus (\sigma^I \cup \psi)\}$ 
3: loop
4:   for all  $S \in \Sigma_{\simeq}$  do
5:     if  $S \cap F \neq \emptyset$  and  $S \subseteq \psi$  then
6:       choose  $s \in S \cap F$ 
7:        $t := \text{TestFromWitness}(s)$ 
8:       return (“fail”,  $t$ )
9:     end if
10:  end for
11:   $\langle \Sigma_{\simeq}, \sigma^I_{\simeq}, \rightarrow_{\simeq} \rangle := \text{CreateAbstractProgram}(P, \Sigma_{\simeq})$ 
12:   $\tau = \text{GetAbstractTrace}(\langle \Sigma_{\simeq}, \sigma^I_{\simeq}, \rightarrow_{\simeq} \rangle, \psi)$ 
13:  if  $\tau = \epsilon$  then
14:    return (“pass”,  $\Sigma_{\simeq}$ )
15:  else
16:     $\langle \tau_{err}, k \rangle := \text{GetOrderedAbstractTrace}(\tau, F)$ 
17:     $t := \text{GenSuitableTest}(\tau_{err}, F)$ 
18:    let  $S_0, S_1, \dots, S_n = \tau_{err}$  in
19:    if  $t = \epsilon$  then
20:       $\Sigma_{\simeq} := (\Sigma_{\simeq} \setminus \{S_{k-1}\}) \cup$ 
21:         $\{S_{k-1} \cap \text{Pre}(S_k), S_{k-1} \setminus \text{Pre}(S_k)\}$ 
22:    else
23:      let  $s_0, s_1, \dots, s_m = t$  in
24:      for  $i = 0$  to  $m$  do
25:        if  $s_i \notin F$  then
26:           $F := F \cup \{s_i\}$ 
27:           $\text{parent}(s_i) := \text{if } i = 0 \text{ then } \epsilon \text{ else } s_{i-1}$ 
28:        end if
29:      end for
30:    end if
31:  end if
32:  /*
33:  The following code is commented out,
34:  and is explained in Section 5:
35:   $\Sigma_{\simeq} := \text{RefineWithGeneralization}(\Sigma_{\simeq}, tt)$ 
36:  */
37: end loop

```

Figure 2: The SYNERGY algorithm

that the program P cannot reach the error ψ if there is no abstract error trace.

The algorithm SYNERGY takes as inputs (1) a program $P = \langle \Sigma, \sigma^I, \rightarrow \rangle$, and (2) a property $\psi \subseteq \Sigma$. It can produce three types of results:

1. It may output “fail” together with a test generating an error trace of P to ψ .
2. It may output “pass” together with a finite-indexed partition Σ_{\simeq} proving that P cannot reach ψ .
3. It may not terminate.

The algorithm is shown in Figure 2. It maintains two core data structures: (1) a finite forest F of states, where for every state $s \in F$, either $s \notin \sigma^I$ and $\text{parent}(s) \in F$ is a concrete predecessor of s (that is, $\text{parent}(s) \rightarrow s$), or $s \in \sigma^I$

and $\text{parent}(s) = \epsilon$; and (2) a finite-indexed partition Σ_{\simeq} of the state space Σ . The regions of Σ_{\simeq} may be specified, for example, by program counter values and predicates over program variables. Initially, F is empty (line 1), and Σ_{\simeq} is the initial partition with three regions, namely, the initial states σ^I , the error states ψ , and all other states (line 2)¹. In each iteration of the main loop, the algorithm either expands the forest F to include more reachable states (in the hope that this expansion will help produce a “fail” answer), or refines the partition Σ_{\simeq} (in the hope that this refinement will help produce a “pass” answer). Intuitively, the expansion of the forest F is done by directed test-case generation in order to cover more regions, and the refinement of the partition Σ_{\simeq} is done at the boundary between a region that we know is reachable, and a region for which we cannot find a concrete test along an abstract error trace. Thus, abstract error traces are used to direct test-case generation, and the non-existence of certain kinds of test cases is used to guide partition refinement.

In each iteration of the loop, the algorithm first checks to see if it has already found a test case to the error region. This is checked by looking for a region S such that $S \cap F \neq \emptyset$ and $S \subseteq \psi$ (line 5). In that case, the algorithm chooses a state $s \in S \cap F$ and calls the auxiliary function **TestFromWitness** to compute a test case (input vector) that generates an error trace. Intuitively, **TestFromWitness** works by successively looking up the *parent* until it finds a root of the forest F . Formally, for a state $s \in F$, the function call **TestFromWitness**(s) returns the state sequence s_0, s_1, \dots, s_n such that $s_n = s$, and $\text{parent}(s_i) = s_{i-1}$ for all $0 < i \leq n$, and $\text{parent}(s_0) = \epsilon$. The initial state s_0 provides the desired test case.

If it is not able to find a test case leading to the error, the algorithm checks if the current partition Σ_{\simeq} provides a proof that P cannot reach ψ . It does this by first building the abstract program P_{\simeq} using the auxiliary function **CreateAbstractProgram** (line 11). Given a partition Σ_{\simeq} , the function **CreateAbstractProgram**(P, Σ_{\simeq}) returns the abstract program $P_{\simeq} = \langle \Sigma_{\simeq}, \sigma^I_{\simeq}, \rightarrow_{\simeq} \rangle$. The next step is to call to the auxiliary function **GetAbstractTrace** (line 12) in order to search for an abstract error trace. If there is no abstract error trace, then **GetAbstractTrace** returns the empty trace ϵ . In that case, the algorithm returns “pass” with the current partition Σ_{\simeq} . Otherwise, **GetAbstractTrace** returns an abstract trace S_0, S_1, \dots, S_n such that $S_n \subseteq \psi$. The next step is to convert this trace into an ordered abstract trace. The abstract trace S_0, S_1, \dots, S_n is *ordered* if the following two conditions hold:

1. There exists a *frontier* $k \stackrel{\text{def}}{=} \text{Frontier}(S_0, S_1, \dots, S_n)$ 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 **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

¹In the examples, the initial partition has a separate region for each program location.

τ_{err} . Intuitively, it works by finding the latest region in the trace that intersects with the forest F , choosing a state in this intersection, and following the *parent* pointers from the chosen state. The function `GetOrderedAbstractTrace` returns a pair $\langle \tau_{err}, k \rangle$, where τ_{err} is an ordered abstract error trace and $k = \text{Frontier}(\tau_{err})$.

The algorithm now tries to extend the forest F along the ordered abstract error trace τ_{err} . In particular, it tries to find a test case that extends F by at least one step at depth $k = \text{Frontier}(\tau_{err})$ along the abstract trace τ_{err} , but not necessarily all the way along τ_{err} ; such suitable tests can potentially deviate from the abstract trace after k steps. This flexibility is crucial: it allows the test to follow the concrete semantics of the program, and in doing so, avoid unnecessary refinements. We define suitable tests in two steps. First we define F -extensions, which are sequences that can be added to F while still maintaining the invariant that F is a forest (without adding cycles to F , or making some node in F have two parents). A finite sequence s_0, s_1, \dots, s_m of states is an F -extension if (1) $s_0 \in \sigma^I$, and (2) $s_i \rightarrow s_{i+1}$ for all $0 \leq i < m$, and (3) there exists k such that (a) $0 \leq k < m$ and (b) $s_i \in F$ for all $0 \leq i < k$ and (c) $s_j \notin F$ for all $k \leq j \leq m$. Given an abstract trace $\tau_{err} = S_0, S_1, \dots, S_n$ with $k = \text{Frontier}(\tau_{err})$, and the forest F , a sequence of states is *suitable* if it is (1) an F -extension and (2) follows the abstract trace τ_{err} at least for k steps. Formally, the auxiliary function `GenSuitableTest`(τ_{err}, F) takes as inputs an ordered abstract trace $\tau_{err} = S_0, S_1, \dots, S_n$ and the forest F , and either returns an F -extension $t = s_0, s_1, \dots, s_m$ such that (a) $m \geq \text{Frontier}(\tau_{err})$ and (b) $s_i \in S_i$ for all $0 \leq i \leq \text{Frontier}(\tau_{err})$, or returns ϵ if no such suitable sequence exists. We note that DART [13] can be used to generate such a suitable state sequence efficiently: for $k = \text{Frontier}(\tau_{err})$, by choosing a state $s \in S_{k-1} \cap F$, performing a symbolic execution along the path in the forest F up to state s , and conjoining the constraints that correspond to the transition to S_k , we can accumulate all constraints needed to drive a test case to the region S_k .

If we succeed in finding such a suitable test case, we simply add it to the forest F (lines 23–27), and continue. If no suitable test is found, then we know that there is no concrete program execution corresponding to the abstract trace S_0, S_1, \dots, S_k . However, we already have a concrete execution along the prefix S_0, S_1, \dots, S_{k-1} , because $S_{k-1} \cap F \neq \emptyset$. Thus, we split the region S_{k-1} using the preimage operator $\text{Pre}(S_k) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \exists s' \in S_k. s \rightarrow s'\}$ (lines 20–21), and thus eliminate the spurious (infeasible) abstract error trace from the abstract program. The call to the auxiliary function `RefineWithGeneralization` (line 35) has been commented out. This call is needed to help SYNERGY terminate on certain programs; we will discuss this in Section 5.

The distinguishing feature of the SYNERGY algorithm is the simultaneous search for a test case to witness an error, and a partition to witness a correctness proof. The two searches work in synergy (and hence the name). The search for the proof guides the test-case generation, because each new test case is generated with respect to an abstract error trace. The search for the test cases guides the proof, because the non-existence of a test case beyond the frontier in the forest of collected tests is used to decide where to refine the partition.

```

void foo(int y)
{
0:   lock.state = U;
1:   do {
2:     lock.state = L;
3:     x = y;
4:     if (*) {
5:       lock.state = U;
6:       y++;
    }
7:   } while (x != y)
8:   if (lock.state != L)
9:     error();
}

```

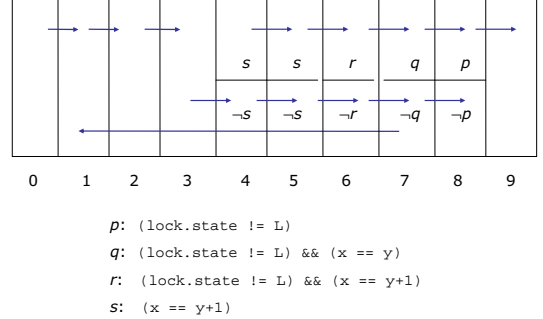


Figure 3: Example on which SLAM works well

4. DISCUSSION

In this section, we illustrate the SYNERGY algorithm on several examples, and compare it with other approaches. For the examples, we use a simple programming language with integer variables and standard control constructs — sequencing, conditionals (non-deterministic choice is denoted by “if (*)”), and loops. A state of such a program consists of a valuation for the variables. The program counter, pc , is a special (implicit) variable in such programs; the values of pc are specified as labels on the statements. We also treat pc in a special way, as is done in most software verification tools: we consider an initial partition, where each possible value of pc defines a separate region (this is a deviation from the description in Figure 2, which starts with three initial regions).

Comparison with counterexample-guided abstraction refinement. Consider the example in Figure 3 from [3]. This example is commonly used to illustrate how SLAM-like tools work [3, 17]. SLAM and BLAST are able to prove the property by discovering two predicates (`lock.state==U`) and (`x==y`) in two refinement steps.

SYNERGY starts with the initial partition $\{\{pc = i\} \mid 0 \leq i \leq 9\}$, and the initial abstract program is isomorphic to the concrete program’s control flow graph. In the first iteration, `GetOrderedAbstractTrace` returns an ordered abstract error trace that executes the loop body exactly once, namely, $\langle (\{pc = 0\}, \{pc = 1\}, \{pc = 2\}, \{pc = 3\}, \{pc = 4\}, \{pc = 7\}, \{pc = 8\}, \{pc = 9\}), 0 \rangle$. For brevity, we omit pc from such traces and simply write the trace as $\langle (0, 1, 2, 3, 4, 7, 8, 9), 0 \rangle$. The second component of the ordered abstract trace (in this case 0) is the frontier, which indicates the position of the first region in the trace that does not have a corresponding state (node) in the forest F main-

tained by the algorithm. In this case, the frontier 0 indicates that no region in the abstract trace has been visited by the forest F , which is initially empty. Thus, `GenSuitableTest` returns a test case (i.e., a value of the input variable y) which traverses the loop some number of times and visits all regions along the abstract trace up to $pc = 8$. The concrete trace generated by this test is added to the forest F .

In the second iteration, `GetOrderedAbstractTrace` returns the ordered abstract trace $\langle(0, 1, 2, 3, 4, 5, 6, 7, 8, 9), 7\rangle$, where the frontier 7 indicates that the region at position 7 in the trace (namely, $\{pc = 9\}$) is the first region that does not have a corresponding state in F . However, `GenSuitableTest` is unable to construct a test that follows this abstract trace and makes a transition from $\{pc = 8\}$ to $\{pc = 9\}$. Thus, `GenSuitableTest` returns ϵ , and the refinement step (lines 20–21 of the SYNERGY algorithm in Figure 2) splits the region $\{pc = 8\}$ into two regions—one satisfying the predicate $(\text{lock.state} \neq L)$, and the other one violating it. Let us call this predicate p , and denote the resulting regions as $\langle 8, p \rangle$ and $\langle 8, \neg p \rangle$. `GetOrderedAbstractTrace` now returns the ordered abstract trace $\langle(0, 1, 2, 3, 4, 5, 6, 7, \langle 8, p \rangle, 9), 6\rangle$, where the frontier 6 indicates that the region $\langle 8, p \rangle$ has not yet been visited by the forest. It is not possible to construct a test that follows this abstract trace and proceeds from region 7 to region $\langle 8, p \rangle$. Thus `GenSuitableTest` returns ϵ once again, and the refinement step splits the region $\{pc = 7\}$ with respect to the predicate $(\text{lock.state} \neq L) \ \&\& \ (x=y)$. Let us call this predicate q , and denote the resulting regions as $\langle 7, q \rangle$ and $\langle 7, \neg q \rangle$. `GetOrderedAbstractTrace` now returns the ordered abstract trace $\langle(0, 1, 2, 3, 4, 5, 6, \langle 7, q \rangle, \langle 8, p \rangle, 9), 7\rangle$. It is possible to construct tests that follow this abstract trace up to region 6, but it is not possible to construct a test that then proceeds from region 6 to region $\langle 7, q \rangle$. This results in another refinement step, this one splitting the region $\{pc = 6\}$ with respect to the predicate $(\text{lock.state} \neq L) \ \&\& \ (x=y+1)$. We call this predicate r . In subsequent iterations, the regions 4 and 5 are split with respect to the predicate $s = (x=y+1)$. This results in a proof of correctness, shown at the bottom of Figure 3.

In the above example, the spurious abstract error traces had each exactly one infeasibility, and refining that infeasibility in each iteration leads to a proof. If an abstract error trace has more than one infeasibility, then existing refinement techniques used by SLAM-like tools have difficulties in choosing the “right” infeasibility to refine. Deterministic loops with a fixed execution count are particularly difficult, and they make tools such as SLAM and BLAST spend as many iterations of the iterative refinement algorithm as there are executions of the loop body. (Although heuristics have been implemented to deal with deterministic loops—e.g., SLAM replaces most deterministic loop predicates with non-deterministic choice—the core difficulties remain.) Consider, for example, the program shown in Figure 1. In this program, the error region at $pc = 6$ is reachable. However, an abstract error trace such as $\langle 0, 1, 2, 5, 6 \rangle$ is spurious, because it exits the loop with 0 iterations. Refining this infeasibility leads to 1000 iterations of the refinement loop, resulting in the introduction of the predicates $(i=0)$, $(i=1)$, \dots , $(i=1000)$ one by one.

SYNERGY avoids these unnecessary refinements. The initial partition is $\{\{pc = i\} \mid 0 \leq i \leq 6\}$, and the initial abstract program is isomorphic to the control flow graph. Consider the ordered abstract trace $\langle(0, 1, 2, 5, 6), 0\rangle$ returned by

```

void foo(int a[])
{
    int i, j;
0:   i = 0; j = 1;
1:   a[j] = 0;
2:   while (i < 1000) {
3:       a[j] = a[j] + i;
4:       i = i + 1;
    }
5:   assume(a[0] <= 0);
6:   error();
}

```

Figure 4: Example on which path slicing does not work well

`GetOrderedAbstractTrace`. Since none of the abstract regions have concrete counterparts in the forest F (which is initially empty), a suitable test returned by `GenSuitableTest` is any test that visits the region $\{pc = 0\}$. Let the test case be $(a=45)$. This test case traverses the *while* loop 1000 times and visits all regions in the abstract trace except $\{pc = 6\}$. All states along the test sequence are added to the forest F . In the second iteration, the procedure `GetOrderedAbstractTrace` returns $\langle(0, 1, 2, (3, 4, 2)^{1000}, 5, 6), 3004\rangle$. This is because, even though `GetAbstractTrace` could have returned a shorter abstract trace, such as $(0, 1, 2, 5, 6)$, by following the parent pointers from the state in forest F that lies in the region $\{pc = 5\}$, the ordered abstract trace is forced to traverse the loop 1000 times. The frontier 3004 indicates that the region $\{pc = 6\}$ has not yet been reached by the forest F . Now, `GenSuitableTest` is able to generate the test case $(a=-5)$ that leads to the error.

Comparison with path slicing. The program from Figure 1 can be handled using path slicing [18]. This technique takes an abstract error trace π and returns a “path slice” π' , which is a projection of π such that (1) the infeasibility of π' implies the infeasibility of π , and (2) the feasibility of π' implies the existence of a concrete error trace. Given the abstract error trace $(0, 1, 2, 5, 6)$, path slicing removes the loop, resulting in the sliced path $(0, 1, 5, 6)$, which immediately leads to the identification of the error. Path slicing has to rely on other static analysis techniques such as pointer analysis. If we change the example so as to keep a two-element array, and replace the variables a and c by the array elements $a[0]$ and $a[1]$, respectively (see Figure 4), then path slicing is unable to slice the path $(0, 1, 2, 5, 6)$ to $(0, 1, 5, 6)$, because a typical alias analysis is not able to ascertain that the loop body does not affect the element $a[0]$. The SYNERGY algorithm, in contrast, finds this error in the same way as in the previous example.

Comparison with computing bisimilarity quotients. Partition refinement algorithms [20] are based on the notion of stability. They start from the same initial partition Σ_{\simeq}^0 as the SYNERGY algorithm (Σ_{\simeq}^0 contains three regions: the initial states, the error states, and all other states). An ordered pair $\langle P, Q \rangle$ of regions in a partition Σ_{\simeq} is *stable* if either $P \cap \text{Pre}(Q) = \emptyset$ or $P \subseteq \text{Pre}(Q)$. A stabilization step consists of choosing a pair $\langle P, Q \rangle$ of regions which is not stable, and splitting P into the two regions $P \cap \text{Pre}(Q)$ and $P \setminus \text{Pre}(Q)$. Partition refinement algorithms work by repeatedly performing stabilization steps until no unstable pair of regions can be found. For finite state spaces, the

LEE-YANNAKAKIS($P = \langle \Sigma, \sigma^I, \rightarrow \rangle, \psi$)

Assumes: $\sigma^I \cap \psi = \emptyset$.

Returns:

(“fail”, t), where t is an error trace of P reaching ψ ; or
 (“pass”, Σ_{\simeq}), where Σ_{\simeq} is a proof that P cannot reach ψ .

```

1:  $T := \sigma^I$ 
2:  $\Sigma_{\simeq} := \{\sigma^I, \psi, \Sigma \setminus (\sigma^I \cup \psi)\}$ 
3: loop
4:   for all  $S \in \Sigma_{\simeq}$  do
5:     if  $S \cap T \neq \emptyset$  and  $S \subseteq \psi$  then
6:       choose  $s \in S \cap T$ 
7:        $t := \text{TestFromWitness}(s)$ 
8:       return (“fail”,  $t$ )
9:     end if
10:  end for
11:  choose  $S \in \Sigma_{\simeq}$  such that  $S \cap T = \emptyset$  and
12:    there exist  $s \in S$  and  $t \in T$  with  $t \rightarrow s$ 
13:  if such  $S \in \Sigma_{\simeq}$  and  $s, t \in \Sigma$  exist then
14:     $T := T \cup \{s\}$ 
15:     $\text{parent}(s) := t$ 
16:  else
17:    choose  $P, Q \in \Sigma_{\simeq}$  such that  $P \cap T \neq \emptyset$  and
18:       $\text{Pre}(Q) \cap P \neq \emptyset$  and  $P \not\subseteq \text{Pre}(Q)$ 
19:    if such  $P, Q \in \Sigma_{\simeq}$  exist then
20:       $\Sigma_{\simeq} := (\Sigma_{\simeq} \setminus \{P\}) \cup \{P \cap \text{Pre}(Q), P \setminus \text{Pre}(Q)\}$ 
21:    else
22:      return (“pass”,  $\Sigma_{\simeq}$ )
23:    end if
24:  end if
25: end loop

```

Figure 5: The LEE-YANNAKAKIS algorithm

most efficient known partition refinement algorithm is due to [26]. Partition refinement algorithms terminate with the bisimilarity quotient of the original program, provided that the bisimilarity quotient has a finite index; if there are infinitely many bisimilarity classes, then partition refinement does not terminate. Formally, given a program P and a property ψ , a *simulation* \preceq is a binary relation on Σ such that for all states $s, t \in \Sigma$, if $s \preceq t$, then (1) s and t lie within the same region of the initial partition Σ_{\simeq}^0 ; and (2) for all states $s' \in \Sigma$ with $s \rightarrow s'$, there exists a state $t' \in \Sigma$ with $t \rightarrow t'$ such that $s' \preceq t'$ [25]. The symmetric simulation relations are called *bisimulations*, and the coarsest of those is called *bisimilarity*. If the program P cannot reach the error region ψ , then the bisimilarity quotient provides a proof for this fact.

Sometimes the reachable part of the bisimilarity quotient is finite, even though the bisimilarity quotient itself has an infinite index. Thus it is important to stabilize only reachable regions. The LEE-YANNAKAKIS algorithm [23] finds the reachable bisimilarity quotient of an infinite-state system. For comparison, we present the LEE-YANNAKAKIS algorithm in Figure 4 using notation similar to the SYNERGY algorithm. It starts from the initial partition and iterates two phases: (1) “search” tries to produce a witness (state) for the concrete reachability of each abstractly reachable region (the witnesses are maintained in a tree T); and (2) “split” tries to stabilize every reachable region with respect to all other regions (including the unreachable ones). The LEE-

```

void foo(int y)
{
0:   while (y > 0) {
1:     y = y - 1;
   }
2:   assume(false);
3:   error();
}

```

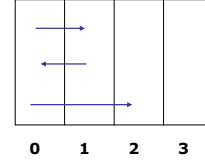


Figure 6: Example on which LEE-YANNAKAKIS does not work well

YANNAKAKIS algorithm iterates the search and split phases while giving priority to search as long as it makes progress. Recently this idea has re-appeared as underapproximation-guided abstraction refinement for model checking software [27]. There, a concrete search is performed with abstract matching (so as to keep at most one witness per region), and after the search is done, stability is checked on every pair $\langle P, Q \rangle$ of regions for which P has a witness.

The main difference between LEE-YANNAKAKIS and SYNERGY is that SYNERGY does not attempt to find a part of the bisimilarity quotient. Indeed, when SYNERGY terminates with a proof, the partition Σ_{\simeq} does not necessarily form (a part of) a bisimulation quotient. Instead, the resulting partition Σ_{\simeq} is guaranteed to *simulate* the program P with respect to the error region ψ ; that is, for all states $s \in \Sigma$, (1) if $s \in \sigma^I$, then $\text{Region}(s) \in \sigma^I_{\simeq}$; and (2) for all states $s' \in \Sigma$ with $s \rightarrow s'$, we have $\text{Region}(s) \rightarrow_{\simeq} \text{Region}(s')$; and (3) if $s \in \psi$, then $\text{Region}(s) \in \psi_{\simeq}$. Note that if Σ_{\simeq} simulates P with respect to ψ , then every concrete error trace can be matched step-by-step by an abstract error trace. The reachable bisimilarity quotient is guaranteed to simulate P with respect to ψ , but often SYNERGY finds a proof with fewer equivalence classes than the number of reachable bisimilarity classes. Thus, SYNERGY terminates in strictly more cases than LEE-YANNAKAKIS (see Section 5).

To illustrate this difference between SYNERGY and LEE-YANNAKAKIS, consider the program in Figure 6. We first explain how SYNERGY works on this example. The initial partition is $\{\{pc = i\} \mid 0 \leq i \leq 3\}$, and the abstract program is shown at the bottom of Figure 6. SYNERGY terminates immediately, because there is no abstract error trace. In contrast, LEE-YANNAKAKIS and [27] do not terminate on this example. This is because the reachable part of the initial partition is not stable. Refinements to stabilize the partition cause the introduction of the series $(y > 0)$, $(y > 1)$, $(y > 2)$, ... of predicates without terminating. This program does not have a finite reachable bisimulation quotient. However, it has a small abstraction which can prove the absence of errors, and SYNERGY finds that abstraction. Another difference between SYNERGY and reachable bisimulation quotient algorithms is that SYNERGY allows multiple concrete states to be explored within each abstract region

```

void foo(int x, int y)
{
0:   if (x != y)
1:     if (2*x == x + 10)
2:       error();
}

```

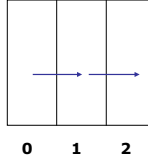


Figure 7: Example on which DART works well

during test generation (the reachable bisimulation quotient algorithms keep only one witness per region). This difference, though seemingly minor, allows SYNERGY to handle deterministic loops efficiently. Recall the example from Figure 1: LEE-YANNAKAKIS and [27] need to introduce the predicates $(i==0)$, $(i==1), \dots, (i==1000)$ before discovering that the error region is reachable.

Comparison with directed testing. DART [13] is an automated directed random testing tool which works by combining concrete and symbolic execution. DART starts with a randomly generated test input, and in addition to running the test concretely, it also executes the program with symbolic values for the input along the same control path as the test. The symbolic execution is then used to derive inputs for driving another test down another (closely related) path. By repeatedly choosing new paths, DART directs test case generation successively through all control paths of the program. In contrast to DART, SYNERGY does not attempt to cover all paths. Instead, it tries to cover all abstract states (regions).

Consider the example shown in Figure 7 from [13]. On this example, SYNERGY works very similar to DART. In the first iteration `GetOrderedAbstractTrace` returns the ordered abstract trace $\langle(0, 1, 2), 0\rangle$, where the frontier points to the region $\{pc = 0\}$. Thus, any test that visits $\{pc = 0\}$ is a suitable test. Say `GenSuitableTest` generates the test case $(x==10) \ \&\& \ (y==10)$ and adds the resulting concrete execution trace to the forest F . This test case covers only the region $\{pc = 0\}$. In the second iteration, `GetOrderedAbstractTrace` returns the ordered abstract trace $\langle(0, 1, 2), 1\rangle$, with the frontier pointing to $\{pc = 1\}$, because the test from the previous iteration already visited the region $\{pc = 0\}$. Thus, `GenSuitableTest` tries to find a test case that drives the program from $\{pc = 0\}$ to $\{pc = 1\}$, using the constraint $(x \neq y)$. Say `GenSuitableTest` generates the new test case $(x==50) \ \&\& \ (y==255)$, which visits the region $\{pc = 1\}$. In the third iteration, `GetOrderedAbstractTrace` returns the ordered abstract trace $\langle(0, 1, 2), 2\rangle$, with the frontier pointing to $\{pc = 2\}$. Now `GenSuitableTest` tries to find a test case that drives the program from $\{pc = 0\}$ to $\{pc = 1\}$ and further to $\{pc = 2\}$, using the constraints $(x \neq y)$ and $(2*x == x + 10)$. Say the third generated test case is $(x==10) \ \&\& \ (y==25)$. This test follows the abstract trace $(0, 1, 2)$ and finds the error.

```

void foo()
{
0:   lock.state = L;
1:   if (*) {
2:     x0 = x0 + 1;
}
3:   else {
4:     x0 = x0 - 1;
}
5:   if (*) {
6:     x1 = x1 + 1;
}
7:   else {
8:     x1 = x1 - 1;
}
...

m:   if (*) {
m+1:   xn = xn + 1;
}
m+2:  else {
m+3:   xn = xn - 1;
}
m+4:  if (lock.state != L)
m+5:   error();
}

```

Figure 8: Example on which DART does not work well

Unlike DART, we note that SYNERGY works by trying to exercise abstract traces that lead to the error region. To clarify this difference, consider the example shown in Figure 8. SYNERGY is able to prove this program correct in $O(n)$ iterations by refining each abstract state with the predicate $(\text{lock.state}==L)$. However, DART requires $O(2^n)$ test cases in order to cover all control paths of the program. This is also an illustration of the difference between the goals of DART and SYNERGY. DART’s goal is to cover all control paths in a program; SYNERGY’s goal is to either prove a given property, or find a test that violates the property. Real-world programs have combinations of the “diamond” structure of *if-then-else* statements (as in Figure 8) and loops (as in Figure 3 or Figure 1). In these cases, SYNERGY performs better than running both SLAM and DART in parallel independently, because its SLAM-like proof search moves through the “diamond” structure quickly without enumerating an exponential number of paths, and its DART-like directed tests cover the loop quickly without any iterative abstraction refinements.

5. SOUNDNESS AND TERMINATION

In this section we present some theorems that characterize the SYNERGY algorithm. The first theorem states that SYNERGY is sound, in that every error and every proof found by SYNERGY is valid.

THEOREM 1. *Suppose that we run the SYNERGY algorithm on a program P and property ψ .*

- *If SYNERGY returns (“pass”, Σ_{\simeq}), then the partition Σ_{\simeq} simulates P with respect to ψ , and thus is a proof that P cannot reach ψ .*
- *If SYNERGY returns (“fail”, t), then t is an error trace.*

```

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

```

Figure 9: Example on which SYNERGY fails to terminate

Since the property verification problem is undecidable in general, and Theorem 1 guarantees soundness on termination, it necessarily has to be the case that SYNERGY cannot terminate on all inputs. However, we can prove that it terminates in strictly more cases than algorithms that find reachable bisimulation quotients. In order to show this, we will use the following lemma.

LEMMA 1. *Suppose that the LEE-YANNAKAKIS algorithm terminates on input (P, ψ) and returns (“pass”, Σ_{\sim}^*). If we run SYNERGY on the same input (P, ψ) , then every iteration of the main loop (lines 3–37) computes a partition Σ_{\sim} that is coarser than Σ_{\sim}^* (that is, every region in Σ_{\sim}^* is contained in some region in Σ_{\sim}).*

PROOF. We prove this by induction on the number of iterations of SYNERGY’s main loop. The base case is immediate. So assume that after iteration $i = n$ the partition Σ_{\sim} is coarser than Σ_{\sim}^* . We need to show that this claim still holds after iteration $i = n + 1$. Since LEE-YANNAKAKIS has returned “pass” (by assumption), by Theorem 1 (soundness), SYNERGY will not return “fail” in iteration $i = n + 1$. Therefore a local stabilization takes place (lines 20–21 of Figure 2). Since the region that is being split is reachable, this split would also be performed by LEE-YANNAKAKIS during the split phase. Therefore, SYNERGY maintains the invariant that the partition Σ_{\sim} after iteration $i = n + 1$ is coarser than Σ_{\sim}^* , and the lemma follows. \square

THEOREM 2. *If the LEE-YANNAKAKIS algorithm terminates on input $\langle P, \psi \rangle$, then the SYNERGY algorithm terminates on input $\langle P, \psi \rangle$ as well. Furthermore, there exist inputs on which SYNERGY terminates, but LEE-YANNAKAKIS does not.*

PROOF. The first part of the theorem follows from Theorem 1 and Lemma 1. The second part of the theorem follows from the example of Figure 6. \square

The SYNERGY algorithm fails to terminate in cases where the refinement step on lines 20–21 of Figure 2 is unable to find the “right” split. Consider the example shown in Figure 9. On this example, SYNERGY loops by repeatedly splitting the region $\{pc = 3\}$ with the predicates $(y < 0)$, $(y + x < 0)$, $(y + 2x < 0)$, \dots , thereby generating longer and longer test sequences. The abstract trace to the error region merely gets longer and longer in each iteration. The algorithm fails to discover the invariant $(y >= 0) \ \&\& \ (x >= 0)$. In order to cope with such situations, we could add a call to procedure `RefineWithGeneralization` on line 35 to discover a necessary generalization. Examples of such generalization procedures

include widening (see, for example, [8, 14]) and interpolation (see, for example, [16, 19]). However, the need for generalization is orthogonal to the need for discovering the “right” place where to perform refinement. Even predicate discovery algorithms that are able to generalize (such as [14] or [19]) have difficulty with examples such as the deterministic loop from Figure 1. The core contribution of SYNERGY is to use testing in order to help in finding the right place for refinement, both for the purpose of finding errors and the purpose of finding proofs.

6. EVALUATION

We have implemented the SYNERGY algorithm in a tool called YOGI². The implementation is written in the programming language F# [1], and the input to the tool is an x86 program binary for any single-procedure C program with only integer variables. Pointers are currently not supported. Function calls are not supported either, with the exception of two special functions:

- `int nondet()`: This function is used to model non-deterministic input—it takes no inputs and returns an arbitrary integer.
- `void error()`: This function is used to model error states.

Given a C program P with calls to `nondet()` and `error()`, YOGI answers the following question: *Are there possible integer values returned from invocations of the function `nondet()` such that the program P invokes the function `error()`?*

YOGI uses VULCAN [28] to parse the input x86 program binary. The ZAP theorem prover [2] is used to answer validity and satisfiability queries. Since our programs contain only integers, we use only the theory of linear arithmetic. Also, the model generation capability of ZAP is used to implement the function `GenSuitableTest` (see Figure 2). Our implementation is very close to the algorithm described in Figure 2, with a few optimizations. In particular, we sort the concrete states in the forest F that are associated with each region in the order in which they were added to F . This leads to shorter traces from `GetOrderedAbstractTrace`, and makes the tool faster. We allow the suitable test generated by `GenSuitableTest` to run to completion, but use a time-out to abort if a test gets stuck in an infinite loop. We use the technique described in [14] to discover generalization predicates if the algorithm fails to terminate without generalization (function `RefineWithGeneralization`).

Table 1 shows our empirical results. We compare each test program on three algorithms—SYNERGY, SLAM, and LEE-YANNAKAKIS. For each algorithm we give the number of iterations taken, and the actual run time (in seconds). The SLAM tool is the latest version run with default options (and with the `-a0` option, where no apriori abstraction is performed). LEE-YANNAKAKIS is our own implementation of the Lee-Yannakakis algorithm [23], built using the same code infrastructure as SYNERGY. All test programs are small examples such as the ones presented in Section 4. However, they represent code patterns that we see commonly occurring in real-world programs such as device drivers.

²Named for its ability to mix the abstract and concrete, in peace and harmony.

Program	SYNERGY		SLAM		LEE-YANNAKAKIS	
	iters	time	iters	time	iters	time
test1.c	9	3.92	4	1.70	*	*
test2.c	6	7.88	4	1.55	*	*
test3.c	5	2.19	13	8.032	*	*
test4.c	2	2.67	12	3.52	22	8.08
test5.c	2	1.28	1	0.90	*	*
test6.c	1	1.45	1	1.27	1	1.75
test7.c	6	2.11	4	1.11	6	2.06
test8.c	2	1.28	2	1.19	*	*
test9.c	3	1.39	1	1.19	3	1.42
test10.c	3	1.52	1	1.25	3	1.52
test11.c	2	1.30	13	5.03	*	*
test12.c	7	2.30	13	10.25	*	*
test13.c	12	3.17	2	1.31	12	3.18
test14.c	1	1.0625	12	3.453	*	*
test15.c	3	5.98	*	*	3	5.65
test16.c	3	9.20	*	*	*	*
test17.c	2	2.28	*	*	*	*
test18.c	24	13.41	*	*	*	*
test19.c	24	10.84	*	*	*	*
test20.c	22	9.42	*	*	*	*

Table 1: Experimental results (“*” indicates that YOGI does not terminate within 20 minutes)

The programs `test1.c` and `test2.c` are similar to the program from Figure 3. Both SLAM and SYNERGY terminate on these examples. SLAM terminates with fewer iterations on both programs due to the following reason: once a predicate is discovered, SLAM uses it at all program locations, but our current implementation for SYNERGY introduces the predicate only in the regions that result from splitting. LEE-YANNAKAKIS does not terminate on these programs, because they do not have finite reachable bisimulation quotients. The programs `test3.c`, `test4.c`, `test5.c`, `test6.c`, and `test7.c` are similar to the program from Figure 1. SLAM takes longer to prove `test3.c` and `test4.c`—these programs have deterministic loops, and SLAM will discover many predicates (corresponding to iterating the loop) before proving the property. LEE-YANNAKAKIS does not terminate on `test5.c`, because it does not have a finite reachable bisimulation quotient. The program `test8.c` is similar to the program from Figure 6, and therefore LEE-YANNAKAKIS does not terminate. The programs `test9.c` and `test10.c` are similar to the program from Figure 7. The programs `test11.c` and `test12.c` have the following structure: they have a deterministic loop with nondeterministic branching within the loop. SYNERGY discovers the loop invariant, because it is checked after the loop. However, since SLAM discovers predicates by forward symbolic execution, it introduces many predicates (corresponding to iterating the loop), and therefore takes longer to prove the property. Since `test11.c` and `test12.c` do not have a finite reachable bisimulation quotient, LEE-YANNAKAKIS does not terminate on these programs. The program `test13.c` is similar to the example from Figure 8. All three algorithms terminate on this example. SLAM terminates with fewer iterations due to the same reason as for `test1.c` and `test2.c`—once a predicate is found, SLAM adds it at all program locations, whereas our implementations of SYNERGY and LEE-

YANNAKAKIS add it only to the regions resulting from the split. The program `test14.c` combines loops and diamond-shaped branching. The program `test15.c` has a loop with a check for error inside the loop. Interestingly, the program is correct, and has a finite reachable bisimulation quotient. Both SYNERGY and LEE-YANNAKAKIS terminate on this example with three iterations, but SLAM does not terminate. The program `test16.c` is a correct program with an infinite loop, and a check for error outside the loop. SYNERGY proves this program correct, whereas SLAM does not terminate, because it fails to discover the loop invariant. LEE-YANNAKAKIS does not terminate on this program either, because it does not have a finite reachable bisimulation quotient. For `test17.c`, `test18.c`, `test19.c`, and `test20.c`, the SYNERGY implementation uses predicates generated by the function `RefineWithGeneralization` (line 35). Of all the examples presented, these are the only ones that use predicates discovered by from generalization. SLAM and LEE-YANNAKAKIS do not terminate on these examples, but this comparison is not fair, because their implementations do not invoke any predicate generalization functions.

7. CONCLUSION

Over the past few years, systematic testing tools and formal verification tools have been a very active area of research. Several recent papers have predicted that testing and verification can be combined in deep ways [12, 15]. We presented a new algorithm, called SYNERGY, which combines directed testing and abstraction refinement based verification algorithms in a novel way. The algorithm has both theoretical and practical benefits. It is theoretically interesting, because it can be viewed as a relaxation of reachable partition refinement algorithms [23, 27] in order to compute a simulation quotient, rather than a bisimulation quotient. It

is practically interesting, because it combines the ability of SLAM-like tools [3, 17] to handle a large number of program paths using a small number of abstract states, with the ability of DART-like tools [13] to avoid unnecessary refinements through concrete execution. Our current implementation, which is called YOGI, handles a small subset of C (no pointers; no procedure calls). We are extending YOGI to support these features as well, in order to enable a more thorough understanding and evaluation of the SYNERGY algorithm.

8. ACKNOWLEDGMENTS

We thank Stefan Schwoon for his comments on an early draft of this paper. We thank Shuvendu Lahiri and Nikolai Tillman for their help with the ZAP theorem prover, and Shankar Shastri for his help with VULCAN. We thank Rakesh K. for his help with performing the experimental comparison with SLAM. We thank the Swiss National Science Foundation for partial support of this research.

9. REFERENCES

- [1] F#: <http://research.microsoft.com/fsharp/fsharp.aspx>.
- [2] T. Ball, S. Lahiri, M. Musuvathi. ZAP: Automated theorem proving for software analysis. Tech. Rep. MSR-TR-2005-137, Microsoft Research, 2005.
- [3] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking of Software (SPIN)*, LNCS 2057, pp. 103–122. Springer, 2001.
- [4] W.R. Bush, J.D. Pincus, D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30:775–802, 2000.
- [5] S. Chaki, E.M. Clarke, A. Groce, S. Jha, H. Veith. Modular verification of software components in C. *IEEE Trans. Software Engineering*, 30:388–402, 2004.
- [6] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, LNCS 1855, pp. 154–169. Springer, 2000.
- [7] E.M. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press, 1999.
- [8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pp. 238–252. ACM, 1977.
- [9] J. Edvardsson. A survey on automatic test data generation. In *Computer Science and Engineering in Linköping*, pp. 21–28. ECSEL, 1999.
- [10] D. Engler, B. Chelf, A. Chou, S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Operating System Design and Implementation*, pp. 1–16. Usenix, 2000.
- [11] P. Godefroid. Model checking for programming languages using VERISOFT. In *Principles of Programming Languages*, pp. 174–186. ACM, 1997.
- [12] P. Godefroid and N. Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In *Integrated Formal Methods*, LNCS 3771, pp. 20–32. Springer, 2005.
- [13] P. Godefroid, N. Klarlund, K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation*, pp. 213–223. ACM, 2005.
- [14] B.S. Gulavani and S.K. Rajamani. Counterexample-driven refinement for abstract interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 3920, pp. 474–488. Springer, 2006.
- [15] E.L. Gunter and D. Peled. Model checking, testing, and verification working together. *Formal Aspects of Computing*, 17:201–221, 2005.
- [16] T.A. Henzinger, R. Jhala, R. Majumdar, K.L. McMillan. Abstractions from proofs. In *Principles of Programming Languages*, pp. 232–244. ACM, 2004.
- [17] T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, pp. 58–70. ACM, 2002.
- [18] R. Jhala and R. Majumdar. Path slicing. In *Programming Language Design and Implementation*, pp. 38–47. ACM, 2005.
- [19] R. Jhala and K.L. McMillan. A practical and complete approach to predicate refinement. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 3920, pp. 459–473. Springer, 2006.
- [20] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite-state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.
- [21] D. Kröning, A. Groce, E.M. Clarke. Counterexample-guided abstraction refinement via program execution. In *Formal Engineering Methods (ICFEM)*, LNCS 3308, pp. 224–238. Springer, 2004.
- [22] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
- [23] D. Lee and M. Yannakakis. On-line minimization of transition systems. In *Theory of Computing (STOC)*, pp. 264–274. ACM, 1992.
- [24] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems*. Springer, 1995.
- [25] R. Milner. An algebraic definition of simulation between programs. In *Artificial Intelligence (IJCAI)*, pp. 481–489. British Computer Society, 1971.
- [26] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM J. Computing*, 16:973–989, 1987.
- [27] C.S. Pasareanu, R. Pelánek, W. Visser. Concrete model checking with abstract matching and refinement. In *Computer-Aided Verification*, LNCS 3576, pp. 52–66. Springer, 2005.
- [28] A. Srivastava, A. Edwards, H. Vo. VULCAN: Binary transformation in a distributed environment. Tech. Rep. MSR-TR-2001-50, Microsoft Research, 2001.
- [29] G. Yorsh, T. Ball, M. Sagiv. Testing, abstraction, theorem proving: Better together! In *Software Testing and Analysis (ISSTA)*, pp. 145–156. ACM, 2006.