SYNERGY : A New Algorithm for Property Checking

Bhargav S. Gulavani* bhargav@cse.iitb.ernet.in
Thomas A. Henzinger† tah@epfl.ch
Yamini Kannan‡ yaminik@microsoft.com
Aditya V. Nori‡ adityan@microsoft.com
Sriram K. Rajamani‡ sriram@microsoft.com

*IIT Bombay †EPFL ‡Microsoft Research India

June 2006
Technical Report
MSR-TR-2006-76

The property checking problem is to check if a 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 have been proposed: (1) testing and (2) verification. Testing tries to find inputs and executions that demonstrate violations to the property. Verification tries to find proofs that all executions of the program satisfy the property. Testing works when errors are easy to find, and verification works when proofs are easy to find. Testing is typically inefficient for correct programs, and verification methods are typically inefficient at finding errors. We propose a new algorithm, SYNERGY, that combines testing and verification. SYNERGY generalizes several disparate algorithms in the literature including: (1) counterexample driven refinement approaches for verification (such as SLAM [2], BLAST [15], MAGIC [5]), (2) directed testing approaches (such as DART [13]) and (3) partition refinement approaches (such as Paige-Tarjan [22] and Lee-Yannakakis [19]) algorithms). This paper presents a description of the SYNERGY algorithm, its theoretical properties, detailed comparison with related algorithms, and a prototype implementation in a tool —YOGI.

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

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. (These 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 guided testing [13]. In this approach, the program is executed symbolically, by collecting all constraints along a path. This information is then be 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 [6,8], the exact envelope of a program is constructed as the reachable state space. At the other extreme, in deductive verification [20], 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 intermediate approaches try to address these issues. For instance, in counter-example guided abstraction refinement [2, 5, 7, 15, 18], 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 proof, for example, if complete coverage can be ensured by a test-case generator, or if no potential errors (neither 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 byproduct of an expensive, failed search for proof.

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 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. Suppose we are partially on the way towards constructing a proof, that is, we have constructed a partition of the reachable state space which not completely safe (it contains abstract paths to errors, real or false). Where is the most promising place to look for real errors? It is precisely along the abstract error paths. So we drive concrete program executions as far as possible along these paths, using the technology of [13]. Conversely, suppose we have performed many program executions without finding a bug. Where is the most promising place to refine our partial proof? It is precisely at the last points on abstract error paths which have been visited by concrete program executions. At such a point, all performed program executions part from the abstract error path, which is thus likely to become infeasible exactly at that point.

Synergy, thus, is a combination of underapproximate and overapproximate reasoning: program execution produces a successively more accurate underapproximation of the reachability tree of the program, and partition refinement produces produces a successively more accurate overapproximation. The overapproximation is used to guide the underapproximation to real errors as quickly as possible; the underapproximation is used to guide the overapproximation to a proof as quickly as possible. In other words, Synergy guides testing towards errors; it can be viewed as "property-directed testing." When the property (error region) changes, different tests will be favored. If some parts of a program can easily be proved safe, then Synergy quickly focuses the tests on the other program parts. In partition refinement, on the other hand, the most difficult problem has been to decide where to refine an infeasible abstract error trace. 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 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- and proof-based tools.

Synergy bears some resemblance to the Lee-Yannakakis algorithm [19], which recently has also been suggested for software verification [23]. 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 important, 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, where Synergy terminates with a proof, but 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 the error, 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 because tests (program execution) are less expensive than refinement (which involves theorem prover calls), and they give valuable information for choosing the next refinement step.

This paper presents a description of the SYNERGY algorithm, its theoretical properties (soundness and termination), a detailed comparison with related algorithms, and a prototype implementation in a tool —YOGI. YOGI currently works for single procedure C programs with integer variables, and checks only safety properties 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 refinement and partition refinement.

2 Overview

We informally present the algorithm SYNERGY on an example that is difficult for SLAM-like tools. Consider the program from Figure 3. 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. (Without loss of generality, we specify safety properties by invoking a special `error()` function.) Clearly, the program reaches the error iff the input `a` is zero or negative. A 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. Guided testing, by contrast, performs well on this example. If a first test input `a` with `(a>0)` is chosen, the test will not pass the assumption at location `5`. Then, a DART-like 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 Figure 2 and Figure 8 below), where SLAM succeeds quickly in finding a proof, SYNERGY does so as well, by performing a SLAM-like overapproximate analysis. 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. In particular, DART works very well with deterministic loops (since there is only one path that a test can cover quite easily, but a large number of abstract states that take several iterations for iterative refinement to enumerate), and SLAM works very well with sequences of branches (such as Figure 8, since there are a small number of abstract states, 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 in isolation.

SYNERGY keeps 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 is 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. The partition A is repeatedly refined by splitting abstract states. As soon as the error location becomes unreachable in the abstraction A, a proof has been found. (Note that at this point, the partition A does not need to be stable, i.e., it is not necessarily a bisimulation of the program.)

SYNERGY grows the forest F by looking at the partition A, and it refines P by looking at F. Whenever there is an (abstract) error path in A, SYNERGY chooses an error path tt in A such that (i) tt has a prefix t which corresponds to a (concrete) path in F, and (ii) no abstract state in tt after the prefix t is visited in F. Such a "nice" path tt always exists. SYNERGY now tries to add to F a new test which follows the nice path tt for at least one transition past the prefix t. 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 nice path. And even if the suitable test does not hit the error, it will indicate a longer feasible prefix of the nice 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 prefix t along the nice path tt. This transition of tt from, say, a to b can always be removed by refining the abstract state a into $a \backslash \mathbf{Pre}(b)$, where $\mathbf{Pre}$ is the weakest-precondition operator. Then SYNERGY continues by choosing a new nice path, until either F finds a real program error or A provides a proof of program correctness.

On the example from Figure 3, the first nice path found by SYNERGY is the abstract error trace `0, 1, 2, 3, 4, 5, 6`. Since the forest F is initially empty, SYNERGY adds some test that proceeds from location `0` to `1` along the nice 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: the new nice path that SYNERGY returns executes the loop body 1000 times before proceeding to locations `5` and `6`. This is because any shorter abstract error path (which contains fewer loop iterations) has no maximally corresponding prefix in F: consider an abstract path tt' 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 tt' until `5`, the path tt' is not "nice." Once the nice 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 $\xrightarrow{*}$ to denote the 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 $(P = \langle \Sigma, \sigma^I, \rightarrow \rangle, \psi)$. The answer to the property checking problem is "fail" if there is some $s \in \sigma^I$ and $t \in \psi$ such that $s \xrightarrow{*} t$, and "pass" otherwise.

We desire to produce witnesses for both the "fail" and "pass" answers. The witness to "fail" is a finite sequence of states $s_0, s_1, s_2, \ldots, s_n$ such that $s_0 \in \sigma^I$, and $s_i \rightarrow s_{i+1}$ for $0 \le i \le n-1$, and $s_n \in \psi$. The witness to "pass" is a finite-indexed partition $\Sigma_\simeq$ of Σ that proves the absence of paths leading from initial states to bad states. Such proofs are constructed using *abstract programs*. We consider equivalence relations $\simeq$ of Σ which induce partitions with a finite index. For any such relation $\simeq$ of Σ, we define an abstract program $P_\simeq = \langle \Sigma_\simeq, {\sigma^I}_\simeq, \rightarrow_\simeq \rangle$ where (1) $\Sigma_\simeq$ are the equivalence classes of Σ under the equivalence $\simeq$, (2) ${\sigma^I}_\simeq = \{S \in \Sigma_\simeq \mid S \cap \sigma^I \neq \Phi\}$ is the set of classes of Σ that have some state from σ^I, and (3) $S \rightarrow_\simeq T$, for $S, T \in \Sigma_\simeq$ iff there exists $s \in S$ and $t \in T$ such that $s \rightarrow t$. We use the term *regions* to denote the equivalence classes in $\Sigma_\simeq$. We use the notation $\psi_\simeq$ to denote the equivalence classes that intersect with ψ. Formally, $\psi_\simeq = \{S \in \Sigma_\simeq \mid S \cap \psi \neq \Phi\}$. Any path starting at the initial region ${\sigma^I}_\simeq$ in the abstract program $P_\simeq$ is called an *abstract trace* in this program.

A finite-indexed partition $\Sigma_\simeq$ is said to be a *proof* for the "pass" answer if there is no abstract trace in the abstract program $P_\simeq$ leading to a region in $\psi_\simeq$.

Our algorithm SYNERGY takes two inputs (1) a program $P = \langle \Sigma, \sigma^I, \rightarrow \rangle$, and (2) a property $\psi \subseteq \Sigma$. It can produce 3 types of results:

1. It may produce "fail" with a test $s_0, s_1, s_2, \ldots, s_n$ such that $s_0 \in \sigma^I$, and $s_i \rightarrow s_{i+1}$ for $0 \le i \le n-1$, and $s_n \in \psi$.
2. It may produce "pass" with a finite partition $\Sigma_\simeq$ that is a proof for the "pass" answer.
3. It may not terminate.

The SYNERGY algorithm is shown in Figure 3. It maintains two core data structures (1) a forest F of concrete states, where for every $s \in F$, $\mathit{parent}(s) \in F \cup \{\epsilon\}$, and (2) a finite-indexed partition $\Sigma_\simeq$ of Σ. Initially, F is empty (line 1) and $\Sigma_\simeq$ is the initial partition with 3 regions, namely, the initial states σ^I, the error states ψ, and all other states that are neither in σ^I nor in ψ (line 2). In each iteration, 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

```
SYNERGY($P = \langle \Sigma, \sigma^I, \rightarrow \rangle$, $\psi$)
Assume: $\sigma^I \cap \psi = \emptyset$
Returns: (1) (“fail”, $t$), where $t$ is a test that reaches $\psi$
         (2) (“pass”, $\simeq$), where $\simeq$ is a proof for unreachability of $\psi$

$F := \emptyset$
$\Sigma_{\simeq} := \{\sigma^I, \psi, \Sigma \setminus (\sigma^I \cup \psi)\}$

loop
  for all ( $S \in \Sigma_{\simeq}$) do
    if $(S \cap F \neq \Phi)$ and $(S \subseteq \psi)$ then
      pick $s \in S \cap F$;
      $t :=$ TestFromWitness($s$);
      return (“fail”, $t$)
    end if
  end for

  $\langle \Sigma_{\simeq}, {\sigma^I}_{\simeq}, \rightarrow_{\simeq} \rangle :=$ CreateAbstractProgram($P, \Sigma_{\simeq}$);
  $at =$ GetAbstractTrace($\langle \Sigma_{\simeq}, {\sigma^I}_{\simeq}, \rightarrow_{\simeq} \rangle, \psi$);
  if $(at = \epsilon)$ then
    return (“pass”, $\Sigma_{\simeq}$)
  else
    $\langle tt, k \rangle :=$ GetNiceAbstractTrace($at, F$);
    let $S_0, S_1, \ldots, S_n = tt$ in
    $t :=$ GenSuitableTest($tt, F$)
    if $(t = \epsilon)$ then
      $\Sigma_{\simeq} :=$
        $(\Sigma_{\simeq} \setminus \{S_{k-1}\}) \cup \{S_{k-1} \cap \mathbf{Pre}(S_k), S_{k-1} \setminus \mathbf{Pre}(S_k)\}$
    else
      let $s_0, s_1, \ldots, s_m = t$ in
      for $(i = 0$ to m) do
        if $(s_i \notin F)$ then
          $F := F \cup \{s_i\}$;
          $parent(s_i) :=$ if $(i = 0)$ then $\epsilon$ else $s_{i-1}$
        end if
      end for
    end if
  end if
  /*
  The following code is commented out,
  and is explained in Section 6.
  $\simeq :=$ RefineWithGeneralization($\simeq, tt$)
  */
end loop
```

Figure 1: The SYNERGY algorithm

produce a "pass" answer). Intuitively, the expansion of the forest F is done by directed test case generation (similar to DART [13]) to cover more regions, and the refinement of the partition $\Sigma_{\simeq}$ 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 trace. Thus, abstract traces that lead to an error region in the abstract program $P_{\simeq}$ are used to direct test case generation, and the non-existence of certain kinds of test cases are used to guide partition refinement.

In each iteration of the loop, the algorithm first checks to see if we have already found a test case to the error region. This is checked by looking for a region S such that $S \cap F \neq \Phi$ and $S \subseteq \psi$ (line 6). In that case, the algorithm picks a state $s \in S \cap F$ and calls the auxiliary function **TestFromWitness** to compute a test sequence that leads to the error. Intuitively, **TestFromWitness** works by successively looking up the *parent* starting with its argument until it finds a root of the forest F. Formally, for a state $s \in F$, the function call **TestFromWitness**(s) returns a test sequence $s_0, s_1, \ldots, s_n$ such that $s_n = s$, and $\mathit{parent}(s_i) = s_{i-1}$, for all $0 < i \leq n$, and $\mathit{parent}(s_0) = \epsilon$.

If we have not been able to find a test case leading to the error, the algorithm checks if the current partition $\Sigma_{\simeq}$ is a proof for unreachability of ψ. It does this by first building the abstract program $P_{\simeq}$ using the auxiliary function **CreateAbstractProgram** (line 13). Given a partition $\Sigma_{\simeq}$, the function **CreateAbstractProgram**$(P, \Sigma_{\simeq})$ returns the abstract program $P_{\simeq} = \langle \Sigma_{\simeq}, {\sigma^I}_{\simeq}, \rightarrow_{\simeq} \rangle$. For every pair of regions $S, T \in \Sigma_{\simeq}$, we have that $S \rightarrow_{\simeq} T$ iff there exist $s \in S$ and $t \in T$ such that $s \rightarrow t$. Thus, the abstract program $P_{\simeq}$ simulates the concrete program P by construction.

The next step in the algorithm is the call to the function **GetAbstractTrace** (line 14) to search for an abstract trace that leads to an error region. If there is no such trace, then **GetAbstractTrace** returns an empty trace ϵ. In that case, the algorithm returns "pass" with the current partition $\Sigma_{\simeq}$. Otherwise, **GetAbstractTrace** returns an *abstract trace* $S_0, S_1, \ldots, S_n$ such that:

1. $S_0 \subseteq \sigma^I$.
2. $S_n \subseteq \psi$.
3. For all $0 \leq i \leq j - 1$, we have that $S_i \rightarrow_{\simeq} S_{i+1}$.

The next step is to convert this trace into a *nice* abstract trace. An abstract trace $S_0, S_1, \ldots, S_n$ is said to be *nice* if in addition to the above three conditions, the following hold:

1. There exists $k =$**Frontier**$(S_0, S_1, \ldots, S_n)$ where $0 \leq k \leq n$, such that for all j such that $k \leq j \leq n$ we have $S_i \cap F = \Phi$, and for all i such that $0 \leq i < k$ we have $S_i \cap F \neq \Phi$.
2. There exists $s \in S_{k-1} \cap F$ such that for all $0 \leq i < k$, we have that $S_i =$ **GetRegion**$(\mathit{parent}^{k-1-i}(s))$ (this function maps every concrete state in F to its corresponding abstract region).

We note that whenever there is an abstract trace that leads to the error, there must exist a nice abstract trace. The auxiliary function **GetNiceAbstractTrace** (line 18) converts an abstract trace at to a nice abstract trace tt. Intuitively, it works by finding the latest region in the trace that intersects with the forest F, finding a state that intersects with F in that region, and following the *parent* pointers from this state. **GetNiceAbstractTrace** returns a pair $\langle tt, k\rangle$ where tt is a nice abstract trace and $k = \mathbf{Frontier}(tt)$.

The algorithm now tries to extend the forest F along the nice abstract trace tt. In particular, it tries to find a *suitable* test case that extends F by at least by one step at depth $\mathbf{Frontier}(tt)$ along the abstract trace tt, but not necessarily all the way along tt. Suitable tests can potentially deviate from the abstract trace after $\mathbf{Frontier}(tt)$. This flexibility is crucial, and allows the test to follow the concrete semantics of the program, and avoid un-necessary refinements. We define suitable tests in two steps. First we define F-extensions, which are intuitively 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 sequence of states $s_0, s_1, \ldots, s_m$ is an *F-extension* if (1) $s_0 \in \sigma^I$, (2) for all $0 \le i \le m-1$ we have that $s_i \rightarrow s_{i+1}$, and (3) there exists $0 \le k < m$ such that, for all i where $0 \le i < k$ we have that $s_i \in F$ and for all j where $k \le j \le m$ we have that $s_j \notin F$. Given an abstract trace $tt = S_0, S_1, \ldots, S_n$ with $k = \mathbf{Frontier}(tt)$, and the forest F, a sequence of states is *suitable* if it is (1) an F-extension, and (2) follows the abstract trace tt at least for k steps. Formally, the function $\mathbf{GenSuitableTest}(tt, F)$ takes as inputs a nice abstract trace $tt = S_0, S_1, \ldots, S_n$ and the forest F, and returns a suitable sequence $t = s_0, s_1, \ldots, s_m$ such that $m \ge \mathbf{Frontier}(tt)$ and for all $0 \le i \le \mathbf{Frontier}(tt)$ we have that $s_i \in S_i$, or it returns ϵ if no such suitable sequence exists. We note that DART [13] can be used to generate such a suitable sequence efficiently. Suppose $k = \mathbf{Frontier}(tt)$. Then, by picking $s \in S_{k-1} \cap F$, doing a symbolic execution along the path in the forest F up to step $k-1$, and by conjoining the constraints corresponding to S_k, we can accumulate the constraints needed to drive a test case to S_k, similar to DART [13]. If we succeed in finding such a test case, we simply add this test case to the forest (lines 25-31) and continue. If finding such a test is not possible, then we know that there is no concrete execution corresponding to the abstract trace $S_0, S_1, \ldots, S_{k-1}, S_k$, but we know that there is a concrete execution for the prefix $S_0, S_1, \ldots, S_{k-1}$ since $S_{k-1} \cap F \neq \Phi$. Thus, we split the region S_{k-1} using $\mathbf{Pre}(S_k)$ (lines 22-23), and thus eliminate this false abstract trace in the abstract program.

The call to function **RefineWithGeneralization** (line 37) has been commented out. However, this call is needed to make SYNERGY work on certain programs. We discuss this in Section 6.

The distinguishing feature of the SYNERGY algorithm is the simultaneous search for a test case to witness the error, and a partition to witness the proof. The two searches work in synergy (and hence the name). The search for the proof guides the test case generation, since the test case is always generated with respect to an abstract trace that leads to an error region. The search for the test guides the proof, since the non-existence of a test case at the frontier of the forest of concrete states (obtained by executing tests) is used to decide where to perform the refinement for the proof. In the next section, we show by way of examples and arguments, how SYNERGY compares with existing algorithms in the literature.

4 Discussion

In this section, we illustrate the SYNERGY algorithm on examples, and compare it with prior work. For the examples, we use a simple programming language with integer variables, and usual control constructs –sequencing, conditionals, and loops. A state of such a program consists of a valuation to all the variables. The program counter, *pc*, is a special (implicit) variable in such programs. The values of the *pc* are specified as labels on the statements. We also treat the *pc* in a special way, as done in most tools (for example [2]). We consider an initial partition where each possible value of the *pc* defines a separate partition (this is a deviation from the description in Figure 3, which starts with three initial partitions, specified in line 2 of the algorithm).

```
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:  assert(false);
```

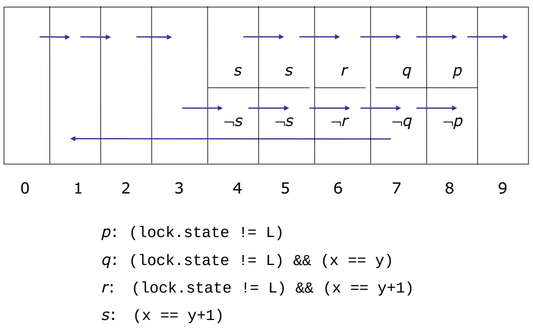
s
s
r
q
p
¬s
¬s
¬r
¬q
¬p
0
1
2
3
4
5
6
7
8
9
p: (lock.state != L)
q: (lock.state != L) && (x == y)
r: (lock.state != L) && (x == y+1)
s: (x == y+1)

Figure 2: Example program on which SLAM works well

Comparison with SLAM. Consider the example in Figure 2 from [2]. This has been a quite popular example, commonly used to illustrate how SLAM works.

SLAM is able to prove the property by discovering two predicates `(lock.state == U)` and `(x == y)`.

We illustrate how SYNERGY works on this example. The initial partition we start with is $\{pc = i\}_{0 \leq i \leq 9}$, and the initial abstract program is isomorphic to the program's control flow graph. In the first iteration, **GetNiceAbstractTrace** returns some nice abstract trace to the error – one that iterates 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 elide 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 abstract trace (in this case 0) is its frontier, which indicates the least index in the trace which does not have an element in the forest F maintained by the algorithm. In this case, the frontier 0 indicates that no region in the abstract trace has been visited by elements in the forest F, which is initially empty. Thus, **GenSuitableTest** returns some test (with some value of the input variable `y`), which traverses traverses the loop a certain number of times, and visits all the regions in the abstract trace up to $pc = 8$. All the elements of this test are added to the forest F.

In the second iteration, **GetNiceAbstractTrace** returns the abstract trace $\langle (0, 1, 2, 3, 4, 7, 8, 9), 7 \rangle$, where the frontier 7 indicates that the region with index 7 in the trace (namely, $pc = 9$) is the first region that does not have a state in the forest F. However, **GenSuitableTest** is unable to construct a test that goes along this abstract trace, and makes a transition from $pc = 8$ to $pc = 9$. Thus **GenSuitableTest** returns ϵ, and the refinement step in lines 22-23 of the SYNERGY algorithm (see Figure 3) splits the region pc=8 with the predicate `(lock.state != L)`. Let us call this predicate p. Denote the resultant regions as $\langle 8, p \rangle$ and $\langle 8, \neg p \rangle$. **GetNiceAbstractTrace** now returns the abstract trace $\langle (0, 1, 2, 3, 4, 7, \langle 8, p \rangle, 9), 6 \rangle$, 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 goes from region 7 to region $\langle 8, p \rangle$. Thus **GenSuitableTest** returns ϵ again, and the refinement step (Lines 22-23 of the SYNERGY algorithm) splits the region pc=7 with the predicate `(lock.state != L ) && (x == y)`. Let us call this predicate q and the resultant split regions as $\langle 7, q \rangle$ and $\langle 7, \neg q \rangle$. **GetNiceAbstractTrace** now returns the 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 test cases that go up to region 6, but it is not possible to construct a test case that transitions from region 6 to region $\langle 7, q \rangle$. This results in the refinement step splitting region 6 using the predicate `(lock.state != L) && (x == y+1)`. Let us call this predicate r. In subsequent iterations, the regions 4 and 5 are split with the predicate $s =$ `(x == y+1)`. This results in a proof of correctness, shown in the bottom of Figure 2.

In the above example, the infeasible abstract traces had exactly one infeasibility, and refining that infeasibility in each iteration leads to a proof. However, if the abstract trace has more than one feasibility, then existing refinement techniques in tools such as SLAM [3] and BLAST [15] have difficulty picking 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

```
0: i = 0;
1: c = 0;
2: while (i < 1000){
3:    c = c + i;
4:    i = i + 1 ;
   }
5: assume(a <= 0);
6: assert(false);
```

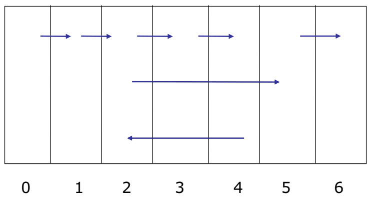

Figure 3: Example program on which SLAM does not work well

many iterations of the iterative refinement algorithm as the execution count of the loop. Though heuristics have been implemented in SLAM to deal with deterministic loops, the main one being replacing most deterministic loop predicates with non-deterministic choice, the core difficulties remain.

For example, consider the program shown in Figure 3. In this program, the assertion failure at pc=6 is reachable. However, an abstract trace such as $\langle 0, 1, 2, 5, 6\rangle$ is infeasible since it exits the loop with 0 iterations. Refining this infeasibility leads to 1000 iterations of the refinement loop, with the introduction of predicates `(i==0), (i==1), ..., (i==1000)`. Below, we show how SYNERGY avoids these unnecessary refinements.

Let the initial partition be $\{pc = i\}_{0\leq i\leq 6}$. The abstract graph is isomorphic to the CFG of the program. Consider the abstract trace $\langle(0, 1, 2, 5, 6), 0\rangle$ returned by **GetNiceAbstractTrace**. Since none of the abstract regions have concrete states that belong to the forest F, a suitable test case returned by **GenSuitableTest** could be 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 these states in the test sequence are added to the forest F.

In the second iteration, the procedure **GetNiceAbstractTrace** 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, say for example, $(0, 1, 2, 5, 6)$, by following the parent pointers from the state in forest F that intersects with $pc = 5$, the nice abstract trace is forced to traverse the loop

1000 times. The frontier 3004 indicates that the region $pc = 6$ has not been reached yet by the forest F. Now, **GenSuitableTest** is able to generate the test case $a = -5$ that leads to the error.

```
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: assert(false);
```

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

Comparison with path slicing. The program in Figure 3 can be handled using the "path slicing" technique presented in [16]. The path slicing takes an abstract trace (leading to an error region) π and returns a "slice" π', which is a projection of π, such that (1) infeasibility of π' implies infeasibility of π, and (2) feasibility of π' implies existence of a concrete path to an error state. Given the abstract trace, $(0, 1, 2, 5, 6)$ the path slicing technique is able to slice away the loop, resulting in the sliced path $(0, 1, 5, 6)$, which immediately leads to the identification of the error. The path slicing algorithm 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 variables `a` and `c` with 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 $(0, 1, 5, 6)$, since a typical alias analysis is not able to ascertain that the loop body does not affect the element `a[0]`. The SYNERGY algorithm finds this error in the same way as in the previous example.
Comparison with reachable bisimulation quotient algorithms. Partition refinement algorithms are all based on the notion of *stability*. An ordered pair of regions $\langle P, Q\rangle$ is *stable* if either $P \cap \mathbf{Pre}(Q) = \Phi$ or $P \subseteq \mathbf{Pre}(Q)$. A stabilization step consists of taking a pair of regions $\langle P, Q\rangle$ that is not stable, and splitting P into two regions – $P \cap \mathbf{Pre}(Q)$ and $P \setminus \mathbf{Pre}(Q)$. Partition refinement algorithms work by repeatedly picking a pairs of regions that are not stable and stabilizing them until no such pair can be found. The naive algorithm is quadratic, and a sophisticated algorithm has been proposed to improve the complexity to $O(n \log n)$ [22].

Partition refinement algorithms terminate with a bisimulation quotient of the original program. Thus, if the bisimulation quotient of a program is infinite, then partition refinement algorithms do not terminate on that program. Sometimes, the reachable portion of the bisimulation quotient is finite, even though the bisimulation quotient itself is infinite. The Lee-Yannakakis algorithm [19] finds the reachable bisimulation quotient of an infinite-state system. For comparison,

```
Lee-Yannakakis($P = \langle \Sigma, \sigma^I, \rightarrow \rangle$, $\psi$)
**Assume:** $\sigma^I \cap \psi = \emptyset$
**Returns:** (1) (“fail”, $t$), where $t$ is a test that reaches $\psi$
             (2) (“pass”, $\simeq$), where $\simeq$ is a proof for unreachability of $\psi$

$T := s \mid s \in \sigma^I$
$\Sigma_{\simeq} := \{\sigma^I, \psi, \Sigma \setminus (\sigma^I \cup \psi)\}$

**loop**
  **for all** ( $S \in \Sigma_{\simeq}$) **do**
    **if** ($S \cap T \neq \Phi$) and ($S \subseteq \psi$) **then**
      pick $s \in S \cap T$;
      $t :=$ **TestFromWitness**($s$);
      return (“fail”, $t$)
    **end if**
  **end for**

  pick $S \in \Sigma_{\simeq}$ such that $S \cap T = \Phi$ and there exists $s \in S$ and $t \in T$ such
  that $t \rightarrow s$.

  **if** such $S \in \Sigma_{\simeq}$, and $s, t \in \Sigma$ exist **then**
    $T := T \cup \{s\}$;
    $parent(s) := t$;
  **else**
    pick $P, Q \in \Sigma_{\simeq}$ such that $P \cap T \neq \emptyset$ and $\mathbf{Pre}(Q) \cap P \neq \Phi$ and $\mathbf{Pre}(Q) \not\subseteq$
    $P$.
    **if** such $P, Q \in \Sigma_{\simeq}$ exist **then**
      $\Sigma_{\simeq} := (\Sigma_{\simeq} \setminus P\}) \cup \{P \cap \mathbf{Pre}(Q), P \setminus \mathbf{Pre}(Q)\}$
    **else**
      return (“pass”,$\Sigma_{\simeq}$)
    **end if**
  **end if**
**end loop**
```

Figure 5: The Lee-Yannakakis algorithm

we have presented the Lee-Yannakakis algorithm in Figure 4 using notations similar to the SYNERGY algorithm. It starts with an initial partition and has two basic steps: (1) search, where it tries to produce a concrete witness to reach each partition (the concrete witnesses are maintained in a tree T), and (2) split, where it tries to stabilize every reachable partition with respect to every other partition (including unreachable ones). The Lee-Yannakakis algorithm iterates the search and split steps, while giving priority to the search step as long as it makes progress. More recently, this idea has re-appeared in the context of underapproximation driven refinement for model checking software [23]. Here, a concrete search is done with abstract matching (so as to ensure reaching at most one concrete state per region), and after the search is done, stability is checked on every pair of partitions $\langle P, Q\rangle$ such that P has a concrete witness.

```
0: while(y > 0){
1:   y = y - 1;
   }
2: assume(false);
3: assert(false);
```

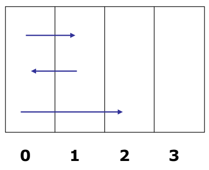

Figure 6: Comparison with reachable bisimulation quotient algorithms

The main difference between SYNERGY and the Lee-Yannakakis algorithm is that SYNERGY does not attempt to find a reachable bisimulation quotient. When SYNERGY stops with a proof, the partitions do not necessarily form a bisimulation quotient. Instead, they simulate the concrete system. Thus, the SYNERGY algorithm terminates in every case where the Lee-Yannakakis algorithm terminates. In addition, the SYNERGY algorithm terminates even in cases where the Lee-Yannakakis algorithms do not terminate due to the reachable bisimulation quotients being infinite (see Section 5 for a formal treatment).

To illustrate the difference between SYNERGY and algorithms for reachable bisimulation quotients, consider the program in Figure 6.

We first explain how SYNERGY works on this example. Let the initial partition be $\{pc = i\}_{0\leq i\leq 3}$. The abstract program is shown in the bottom of Figure 6. The SYNERGY algorithm terminates with this partition immediately, since there is no abstract trace that leads to the error.

In contrast, the algorithms for computing reachable bisimulation quotients are unable to terminate on this example. This is because, the partition shown in the abstract graph is not stable. Refinements to stabilize the partition lead these algorithms to introduce a series of predicates `(y>0)`, `(y>1)`, `(y>2)`,... without terminating. This program does not have a finite reachable bisimulation quotient. However, it has a small abstraction which can prove the absence of the error, and SYNERGY finds that abstraction.

Another minor difference between SYNERGY and reachable bisimulation quotient algorithms is that SYNERGY allows multiple concrete states to be explored per partition during test generation, whereas the reachable bisimulation quotient algorithms allow only one concrete state as a witness per partition. This difference, though minor, allows SYNERGY to handle deterministic loops (recall example from Figure 3) more efficiently, without introducing one predicate for each iteration of the deterministic loop. In contrast, algorithms for computing reachable bisimulation quotients need to introduce the predicates `(i==0)`, `(i==1),...,` `(i==1000)` before discovering that the assertion failure is reachable.

```
0: if(x != y)
1:   if(2*x = x + 10)
2:     assert(false);
```

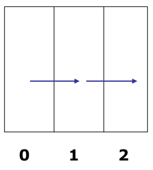

Figure 7: Example program from DART paper

Comparison with DART. DART [13] is an automated directed random testing tool that works by combining concrete and symbolic execution. DART starts with a randomly generated test input, and in addition to running the test concretely, also executes it 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 the different control paths in the program. In contrast to DART, SYNERGY does not attempt to cover all the paths. Instead, it tries to cover all the abstract states.

Consider the example shown in Figure 7 from [13]. In this example, SYNERGY works very similar to DART. In the first iteration **GetNiceAbstractTrace** returns the abstract trace $\langle (0,1,2),0 \rangle$ where the frontier points to the region $pc{=}0$. Thus, any test that visits the region $pc{=}0$ is a suitable test case and **GenSuitableTest** generates a test case, say $x = 10$, $y = 10$, and adds the states in the execution of this test to the forest F. This test case covers only the region $pc{=}0$. In the second iteration **GetNiceAbstractTrace** returns the abstract trace $\langle (0,1,2),1 \rangle$ where the frontier points to $pc{=}1$, since the test in the previous iteration has already visited the region $pc{=}0$. Thus, **GenSuitableTest** tries to come up with test case that goes from $pc{=}0$ to $pc{=}1$, and uses the constraint `(x != y)` to come up with the test $x = 50$, $y = 255$, which visits the abstract state $pc{=}1$. In the third iteration **GetNiceAbstractTrace** returns the abstract trace $\langle (0,1,2),2 \rangle$ where the frontier points to the region $pc{=}2$. Thus, **GenSuitableTest** tries to come up with a test case that goes from $pc{=}0$ to $pc{=}1$, and $pc{=}2$, and uses the constraints `(x !== y) && (2*x == x + 10)` to come up with the test case $x = 10$, $y = 25$, which follows the abstract trace $(0,1,2)$ and finds the error.

```
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;i
   }

   ...

m:   if(*) {
m+1:   xn = xn + 1;
     }
m+2: else {
m+3:   xn = xn - 1;
     }
m+4: assert(lock.state == ’L’);
```

Figure 8: Comparison with DART

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 where it refines each abstract state with the predicate `lock.state == ’L’`. However, DART takes $O(2^n)$ iterations to cover all the different paths in the program. This is also an illustration of the difference between the goals of DART and SYNERGY. DART’s goal is to cover all the control paths in a program. SYNERGY’s goal is to prove a given property, and find tests that violate the given property.

Typical programs have combinations of the “diamond” structure of if-then-else statements (as in Figure 8) and loops (as in Figure 2 or Figure 3). In thse cases, SYNERGY performs better than running SLAM and DART in parallel independently, since the SLAM-like proof searches through the “diamond” structure quickly without enumerating an exponential number of paths, and DART-like directed testing covers the loop directly with concrete execution without doing any iterative refinements.

5 Theorems

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 we run* SYNERGY *on any program $P = \langle \Sigma, \sigma^I, \rightarrow \rangle$ and property ψ.*

- *If* SYNERGY *returns ("pass", $\Sigma_{\simeq}$), then the abstract program $P_{\simeq} = \langle \Sigma_{\simeq}, {\sigma^I}_{\simeq}, \rightarrow_{\simeq} \rangle$ simulates the program P, and thus is a proof that P does not reach ψ.*
- *If* SYNERGY *returns ("fail", t), then the test case t witnesses the violation of property ψ by P.*

Proof: By construction (see Section 3), the abstract program $P_{\simeq}$ simulates P. Therefore when SYNERGY returns ("pass", $\Sigma_{\simeq}$), $P_{\simeq}$ is a witness to the fact that P does not reach ψ. If SYNERGY returns ("fail", t), then the test case t is a concrete witness to the fact that the property ψ is violated by P. ∎

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 input programs. However, we can prove that it terminates on strictly more cases than algorithms that find reachable bisimulation quotients. In order to show this, we will require the following lemma.

Lemma 1 *Every iteration in* SYNERGY*'s main loop (lines 4 – 39) computes a partition $\Sigma_{\simeq}$ that is coarser than the final stable partition computed by the Lee-Yannakakis algorithm.*

Proof: We prove this by induction on the number of iterations i of SYNERGY's main loop.
Basis: Since SYNERGY starts with an overapproximation $P_{\simeq}$ of the program P, the lemma is true for $i = 0$.
Inductive hypothesis: Let the lemma be true for $i = n$. We need to show that the lemma holds for $i = n + 1$.
If $i = n$ and SYNERGY returns ("fail",t), then by the inductive hypothesis, SYNERGY terminates with a partition $\simeq$ that is coarser than Lee-Yannakakis partition. Therefore, assume that $i = n + 1$ and a local stabilization takes place (as shown in line 22). Since the abstract region that is being partitioned (induced by this stabilization) is reachable, this split would also be performed by the Lee-Yannakakis algorithm in its "stabilize" phase. Therefore, SYNERGY maintains the invariant that the computed partition after iteration $i = n + 1$ is coarser than the Lee-Yannakakis partition, and the lemma follows. ∎

Theorem 2 *If the Lee-Yannakakis algorithm [19] terminates on input $\langle P, \psi \rangle$, then* SYNERGY *terminates on $\langle P, \psi \rangle$ as well. Further, there exist inputs $\langle Q, \varphi \rangle$ where* SYNERGY *terminates on $\langle Q, \varphi \rangle$ and Lee-Yannakakis algorithm [19] does not terminate on $\langle Q, \varphi \rangle$.*

Proof: SYNERGY does not terminate only if there are an unbounded number of local refinements (stabilizations) performed on line 22. Therefore, if the Lee-Yannakakis algorithm terminates (the bisimulation quotient is finite), from Lemma 5, it follows that SYNERGY also terminates. The second part of the theorem follows from the example illustrated in Figure 6. ■

6 Non-termination and generalization

The SYNERGY algorithm fails to terminate in cases where the refinement step in Line 22 is unable to find the "right" partition.

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

Figure 9: Example where SYNERGY loops with longer and longer abstract traces

Consider the example in Figure 9. Here, the SYNERGY algorithm loops by repeatedly partitioning the region $pc = 3$ with predicates `(y<0)`, `(y+x<0)`, `(y+2x<0)`, `...` without generating any tests. 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)`.

To cope with these situations, we could add a call to procedure **RefineWithGeneralization** in line 36 to discover such generalizations. Examples of such procedures include widening (see, for example [14]) or interpolation (see, for example [17]).

However, the need for generalization is orthogonal to the need to discover the "right" place to do refinement. Even predicate discovery algorithms that are able to do generalization (such as [14] or [17]) have difficulty with examples such as the deterministic loop in Figure 3, and the core contribution of SYNERGY is to use testing to help with finding the right place in the abstract counterexample to do refinement, both for the purposes of finding errors and finding proofs.

Program	SYNERGY		SLAM		LEE-YANNAKAKIS	
	iters	time	iters	time	iters	time
Test1.c	12	39	4	1.56	*	*
Test2.c	2	0.58	*	*	*	*
Test3.c	1	6.4	2	1.125	*	*
Test4.c	1	0.812	1	1.64	4	0.84
Test5.c	2	0.23	12	5.13	3	0.56
Test6.c	5	0.98	2	1.10	5	1.06
Test7.c	2	11.40	*	*	*	*
Test8.c	15	1.9	*	*	15	2.47
Test9.c	4	0.64	*	*	*	*

Table 1: Experimental results

7 Implementation

We have implemented the SYNERGY algorithm in a tool we call YOGI[1]. Our implementation is in OCAML [24] and the input to the tool is any single procedure C program with only integer variables. Pointers are not supported currently. Function calls are not supported, with the exception of two special functions:

- `int nondet()`, which returns an arbitrary integer and takes no inputs is supported to model nondeterministic input.
- `void error()`, is used to model error.

Given a C program P with calls to `nondet()` and `error()` the YOGI tool answers the question: For all possible integer values returned during from invocations to `nondet()`, does the program P ever invoke `error()`?

YOGI uses CIL [21] to parse the input C program. The ZAP theorem prover [1] is used to answer validity and satisfiability queries. Since our programs have only integers we use only linear arithmetic theory. Also, the model generation capability of ZAP is used to implement the function **GenSuitableTest**(see Figure 3).

Our implementation is very close to the algorithm described in Figure 3, with a few optimizations. In particular, we sort the concrete states in forest F associated with each region, by the order in which they were added. This leads to shorter traces from **GetNiceAbstractTrace**, and makes the tool faster. We allow the suitable test generated by **GenSuitableTest** to run to completion, but use a timeout to abort if the 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.

Table 1 shows our empirical results. We compare each test program on 3 algorithms —SYNERGY, SLAM and LEE-YANNAKAKIS. For each algorithm we

[1]Named so, in reverence to the tool's ability to mix the abstract and concrete, with peace and harmony.

give the number of iterations taken, and the actual run time. The SLAM tool is the latest version run with default options, and LEE-YANNAKAKIS is our own implementation of the Lee-Yannakis algorithm (built using the same code infrastructure as SYNERGY). Run times are in seconds, and a "*" entry indicates that the tool did not terminate in 10 minutes.

All our tests 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.

`Test1.c` is similar to the program in Figure 2. Both SLAM and SYNERGY terminate on this example. SLAM terminates with fewer iterations due to the following reason: once a predicate is discovered, SLAM uses it all program counters, but our current implementation for SYNERGY introduces the predicate only in the current *pc*during splitting. LEE-YANNAKAKIS does not terminate on the first three tests since these programs do not have a finite bisimulation quotients. `Test2.c` is similar to the program in Figure 3. SYNERGY finds a test case that leads to the error in the second iteration, whereas SLAM does not terminate. `Test3.c` is similar to the program in Figure 6. LEE-YANNAKAKIS does not terminate in the first 3 tests since the reachable bisimulation quotient is not finite for these examples. `Test4.c` is similar to the program in Figure 7. `Test5.c` is a program with a deterministic loop (which executes 10 times, and maintains the loop invariant `x == i`, where the variable x is a global, and the loop index is i). SYNERGY and LEE-YANNAKAKIS discover this invariant since it is checked after the loop. However, since SLAM discovers predicates by forward symbolic simulation, it introduces predicates `(i==0),(x==0),(i==1),(x==1),...` before it finds the error. `Test6.c` is similar to the example in Figure 8. All three algorithms terminate on this example. SLAM terminates with fewer iterations due to the same reason as in `Test1.c` —once a predicate is found, the SLAM implementation adds it for all program counters whereas our implementations of SYNERGY and LEE-YANNAKAKIS add it only to the region where the split was done. `Test7.c` combines loops and diamond-shaped branches. `Test8.c` has a loop with a check for error inside the loop. Interestingly, the program is correct, and has a finite bisimulation quotient. Both SYNERGY and LEE-YANNAKAKIS terminate on this example with 15 iterations, but SLAM does not terminate. For `Test9.c`, the SYNERGY implementation uses predicates from the function **RefineWithGeneralization** from line 37. Of all the examples presented, this is the only one that uses predicates from generalization. SLAM and LEE-YANNAKAKIS do not terminate on this example, although this comparison is not fair since their implementations do not invoke any predicate generalization algorithms.

8 Conclusion

Over the past few years, systematic testing tools and 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 (see for example [12]).

We presented a new algorithm SYNERGY, which combines directed testing and verification algorithms in a novel way. The algorithm has both theoretical and practical benefits. It is theoretically interesting, since it appears to be a generalization of reachable partition refinement algorithms [19, 23] to compute simulation quotients, rather than bisimulation quotients. It is practically interesting, since it appears to combine the ability of SLAM-like tools to handle large number of paths using a small number of "abstract" states, with the ability of DART-like tools to avoid refinements through concrete execution. Our current implementation YOGI handles a very restricted subset of C (no pointers, no procedure calls). We are currently extending our implementation to allow these features as well, to allow a more thorough understanding and evaluation of the algorithm.

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. We thank Rakesh K for his help with performing the experimental comparisons with SLAM.

References

[1] T. Ball, S. Lahiri, and M. Musuvathi. Zap: Automated theorem proving for software analysis. Technical Report MSR-TR-2005-137, Microsoft Research, 2005.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, LNCS 2057. Springer-Verlag, 2001.

[3] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.

[5] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[6] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.

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

[9] J. Edvardsson. A survey on automatic test data generation. In *CSE 99: Computer Science and Engineering*, ECSEL, pages 21–28, 1999.

[10] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation.* Usenix Association, 2000.

[11] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186, 1997.

[12] P. Godefroid and N. Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In *IFM 05: Integrated Formal Methods*, LNCS 3771, pages 20–32. Springer-Verlag, 2005.

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

[14] B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS 06: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 3920. Springer-Verlag, 2006.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM, January 2002.

[16] R. Jhala and R. Majumdar. Path slicing. In *PLDI 05: Programming Language Design and Implementation*, 2005.

[17] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS 06: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 3920. Springer-Verlag, 2006.

[18] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[19] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *STOC 92: ACM Symposium on Theory of Computing*, pages 264–274. ACM Press, 1992.

[20] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems*. Springer-Verlag, 1995.

[21] G. C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC 02: Compiler Construction*, LNCS 2304, pages 213–228. Springer-Verlag, 2002.

[22] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[23] C. S. Pasareanu, R. Pelánek, and W. Visser. Concrete model checking with abstract matching and refinement. In *CAV 05: Computer-Aided Verification*, pages 52–66, 2005.

[24] Objective caml – http://caml.inria.fr/ocaml/index.en.html.