Compositional May-Must Program Analysis: Unleashing the Power of Alternation

Patrice Godefroid Aditya V. Nori Sriram K. Rajamani Sai Deep Tetali
Microsoft Research
{pg, adityan, sriram, v-saitet}@microsoft.com

Abstract

Program analysis tools typically compute two types of information: (1) *may* information that is true of *all* program executions and is used to prove the absence of bugs in the program, and (2) *must* information that is true of *some* program executions and is used to prove the existence of bugs in the program. In this paper, we propose a new algorithm, dubbed SMASH, which computes both may and must information *compositionally*. At each procedure boundary, may and must information is represented and stored as may and must summaries, respectively. Those summaries are computed in a demand-driven manner and possibly using summaries of the opposite type. We have implemented SMASH using predicate abstraction (as in SLAM) for the may part and using dynamic test generation (as in DART) for the must part. Results of experiments with 69 Microsoft Windows Vista device drivers show that SMASH can significantly outperform may-only, must-only and non-compositional may-must algorithms. Indeed, our empirical results indicate that most complex code fragments in large programs are actually often either easy to prove irrelevant to the specific property of interest using may analysis *or* easy to traverse using directed testing. The fine-grained coupling and *alternation* of may (universal) and must (existential) summaries allows SMASH to easily navigate through these code fragments while traditional may-only, must-only or non-compositional may-must algorithms are stuck in their specific analyses.

1. Introduction

The use of static analysis and dynamic analysis to find bugs and prove properties of programs has received a lot of attention in the past decade. Tools that came out of research in this area [7, 11, 22, 4, 16, 18] are now routinely used for ensuring quality control in industrial-strength software development projects.

Program analysis tools typically compute two types of information. May information captures facts that are true about *all executions* of the program. For example, static whole-program may-alias analysis is used to bound all possible pointer aliases that can occur during a program execution. For efficiency and scalability reasons, the may information computed is typically over-approximate. Thus, if a may-alias analysis says that pointer `p` may alias pointer `q`, it is possible that no execution exists where `p` and `q` actually alias. Because it is over-approximate, may information can be used to prove properties of programs. For example, consider the following code snippet that is a part of some larger program:

```
1: *p = 4;
2: *q = 5;
3: assert(*p == 4);
```

If the may-alias set of pointer `p` does *not* include pointer `q`, then we know for sure that `p` and `q` never alias, and we can use this information to prove that the assertion in line 3 always holds.

Dually, must information captures facts that are guaranteed to hold on *particular executions* of the program. For example, dynamic must-alias analysis can detect aliases that occur during specific executions of the program. For large programs, the must information computed is typically under-approximate, since it is infeasible to cover all executions. Thus, in the previous example, if a must-alias analysis determines that `p` and `q` must alias in some execution, then we can use this information to prove that the assertion `assert(*p == 4)` can be violated.

In this paper, we focus on the problem of checking whether a sequential program $\mathcal{P}$ satisfies an assertion φ, called the property checking problem. May and must analyses offer complementary approaches to solve this problem. A may analysis can be used to prove that all executions of the program satisfy assertion φ, while a must analysis can be used to prove the existence of some program execution that violates the assertion φ.

Compositional approaches to property checking involve decomposing the whole-program analysis into several sub-analyses of individual components (such as program blocks or procedures), summarizing the results of these sub-analyses, and memoizing (caching) those summaries for possible later re-use in other calling contexts. Summarizing at procedure boundaries is indispensable for scalability. A may summary of a procedure P is of the form $\langle \varphi_1 \stackrel{\mathbf{may}}{\Longrightarrow}_P \varphi_2 \rangle$, where φ_1 and φ_2 are predicates over program states. The may summary $\langle \varphi_1 \stackrel{\mathbf{may}}{\Longrightarrow}_P \varphi_2 \rangle$ means that, if we invoke procedure P from any state satisfying φ_1, the set of all possible states of the program on termination of P is over-approximated by the set of states φ_2. This implies that no states satisfying $\neg\varphi_2$ are reachable from states satisfying φ_1 by executing P. Dually, a must summary of a procedure P is of the form $\langle \varphi_1 \stackrel{\mathbf{must}}{\Longrightarrow}_P \varphi_2 \rangle$, which means that, if we invoke procedure P from any state satisfying φ_1, the set of all possible states of the program on termination of P is under-approximated by the set of states φ_2. This implies that any state satisfying φ_2 is guaranteed to be reachable from some state satisfying φ_1 by executing P.

Intuitively, a may summary of a procedure represents a property that is guaranteed to be true about all executions of the procedure, and a must summary of a procedure represents witness executions of the procedure that are guaranteed to exist. May summaries pro-

vide obvious benefits to improving the efficiency of may analysis: when a compositional may analysis requires a sub-query for a procedure P, a previously-computed may summary for P can potentially be re-used to answer that query without re-analyzing the procedure. Similarly, must summaries can also considerably speed-up must analysis: when a compositional must analysis requires a sub-query for a procedure P, an existing must summary for P can potentially be re-used to answer that query.

In this paper, we present a new algorithm, named SMASH, that performs both may analysis and must analysis *simultaneously*, and uses both may summaries and must summaries to improve the *effectiveness* as well as the *efficiency* of the analysis. The key novel feature of SMASH is its inherent use of alternation: *both* may analysis and must analysis in SMASH use *both* may summaries and must summaries. Surprisingly, this feature of the algorithm enables SMASH to often significantly outperform compositional may-only, compositional must-only and non-compositional may-must algorithms, and hence to handle larger as well as complex programs. From our experiments, we observed that the gain in efficiency is due to alternation in may-must reasoning in SMASH and can be explained as follows: most complex code fragments in large programs are actually often *either* easy to prove irrelevant to the specific property of interest using may analysis *or* easy to traverse using directed testing. Those fragments that are easy to prove irrelevant can often cause a must analysis to needlessly explore a large number of paths searching for a path that violates the property, whereas a may analysis can inexpensively conclude that no such path exists. Dually, those fragments that are easy to traverse using directed testing often cause may analyses to needlessly refine program may-abstractions and get stuck while trying to discover complex loop invariants. This fined-grained coupling and *alternation* of may (universal) and must (existential) summaries allow SMASH to easily navigate through these code fragments while traditional may-only, must-only or non-compositional may-must algorithms are stuck in their specific analyses.

We support the above intuition with significant empirical evidence from checking 85 properties on 69 Windows Vista drivers. With a non-compositional may-must analysis, all these checks take 117 hours (these include 61 checks timing out after 30 minutes). With SMASH, the same checks take only 44 hours, with only 9 time-outs. To drill down into the gains, we also implemented compositional may-only and compositional must-only algorithms, and compare them with SMASH in detail. Our data clearly indicates and quantifies the gains due to the *interplay* between may summaries and must summaries in SMASH.

Though the main goal of this paper is to describe the design, implementation and evaluation of SMASH, we have another auxiliary goal of placing SMASH in the context of the large and diverse amount of existing work in this area. Over the past 10 years, there have been a variety of analysis techniques proposed to perform may analysis, such as SLAM [4], BLAST [23] and ESP [11], and a variety of analysis techniques to perform must analysis, such as DART [16], EXE [8] and SMART [14], and some combinations of the two, such as SYNERGY [19] and DASH [5]. Even if the reader is unfamiliar with this prior work, this paper includes a detailed overview of the entire landscape of such analyses, and places SMASH in the context of all these analyses in a very precise manner. We consider this unified framework another valuable contribution of this paper, given the breadth and depth of these tools.

The remainder of the paper is organized as follows. Section 2 motivates and explains the SMASH algorithm using examples. Section 3 reviews may-must analysis for single-procedure programs. Section 4 presents the SMASH algorithm formally as a set of declarative rules. Section 5 describes our implementation of SMASH and empirical results from running SMASH on several examples. Section 6 surveys related work.

2. Overview

We illustrate using examples the benefit of may summaries, must summaries, and the interplay between may and must summaries in the SMASH algorithm.

The input to SMASH is a sequential program P and an assertion in P. The goal of SMASH is either to prove that the assertion holds for all executions of P, or to find an execution of P that violates the assertion. The verification question "do all program executions satisfy the assertion?" is reduced to a dual reachability question, or *query*, "can some program execution lead to an assertion violation?" SMASH performs a modular interprocedural analysis and incrementally decomposes this reachability query into several sub-queries that are generated in a demand-driven manner. Each sub-query is of the form of $\langle \varphi_1 \stackrel{?}{\Longrightarrow}_f \varphi_2 \rangle$, where φ_1 and φ_2 are state predicates representing respectively a precondition (calling context) and postcondition (return context) for a procedure f (or block) in P. The answer to such a query is "yes" if there exists an execution of f starting in some state $\sigma_1 \in \varphi_1$ and terminating in some state $\sigma_2 \in \varphi_2$, "no" if such an execution does not exist, and "unknown" ("maybe") if the algorithm is unable to decisively conclude either way (the last option is needed since program verification is undecidable in general). SMASH uses may and must summaries to answer queries.

A may summary $\langle \psi_1 \stackrel{\mathbf{may}}{\Longrightarrow}_f \psi_2 \rangle$ implies that, for any state $x \in \psi_1$, for any state y such that the execution of f starting in state x terminates in state y, we have $y \in \psi_2$. For technical convenience (explained in Section 3.1), SMASH maintains negated may summaries, called *not-may* summaries, where the postcondition is complemented. Thus, a not-may summary $\langle \psi_1 \stackrel{\neg\mathbf{may}}{\Longrightarrow}_f \psi_2 \rangle$ implies that for any state $x \in \psi_1$, there does not exist a state $y \in \psi_2$ such that the execution of f starting in state x terminates in state y. Clearly, a not-may summary $\langle \psi_1 \stackrel{\neg\mathbf{may}}{\Longrightarrow}_f \psi_2 \rangle$ can be used to give a "no" answer to a query $\langle \varphi_1 \stackrel{?}{\Longrightarrow}_f \varphi_2 \rangle$ for f provided that $\varphi_1 \subseteq \psi_1$ and $\varphi_2 \subseteq \psi_2$.

A must summary $\langle \psi_1 \stackrel{\mathbf{must}}{\Longrightarrow}_f \psi_2 \rangle$ implies that, for every state $y \in \psi_2$, there exists a state $x \in \psi_1$ such that the execution of f starting in state $x \in \psi_1$ terminates in state $y \in \psi_2$ (this is also called $must^-$ in the literature [2]). Thus, a must summary $\langle \psi_1 \stackrel{\mathbf{must}}{\Longrightarrow}_f \psi_2 \rangle$ can be used to give a "yes" answer to a query $\langle \varphi_1 \stackrel{?}{\Longrightarrow}_f \varphi_2 \rangle$ provided that $\psi_1 \subseteq \varphi_1$ and $\psi_2 \cap \varphi_2 \neq \{\}$.

SMASH computes not-may summaries on abstract state sets, called regions, using predicate abstraction and automatic partition refinement (as in SLAM [4]), and it computes must summaries using symbolic execution along whole-program paths (as in DART [16]). SMASH starts with an empty set of summaries for each function. As the SMASH algorithm proceeds, it progressively refines the not-may summaries and must summaries of each function on demand, in order to prove that the assertion is never violated, or to find an execution that violates the assertion. The exact algorithm is described using declarative rules in Section 4. Here, we illustrate SMASH using small examples.

Example 1. Consider the example in Figure 1. The inputs of this program are the arguments passed to the function `main`. Since function `g` always returns non-negative values, the assertion at line `5` of function `main` can never be reached. This can be proved using a not-may summary for the function `g`. Given the assertion in line `5` of function `main` as a goal, SMASH first tries to find an execution along the path `1, 2, 3, 4, 5`. After some analysis, SMASH generates the query $\langle \mathbf{true} \stackrel{?}{\Longrightarrow}_{\mathtt{g}} (retval < 0) \rangle$. Since all

```
void main(int i1,i2,i3)                    int g(int i)
{                                          {
0: int x1,x2,x3;                           11: if (i > 0)
1: x1 = g(i1);                             12:   return i;
2: x2 = g(i2);                             13: else
3: x3 = g(i3);                             14:   return -i;
4: if ((x1 < 0)||(x2 < 0)||(x3 < 0))       }
5:   assert(false);
}
```

Figure 1. Example to illustrate benefits of not-may summaries.

```
void main(int i1,i2,i3)          int f(int i)
{                                {
0: int x1,x2,x3;                 11: if (i > 0)
1: x1 = f(i1);                   12:   return i;
2: x2 = f(i2);                       else
3: x3 = f(i3);                   13:   return h(i);
4: if (x1 > 0)                    // h(i) is a complex function
5:   if (x2 > 0)                 // such as a hash function
6:     if (x3 > 0)               }
7:       assert(false);
}
```

Figure 2. Example to illustrate benefits of must summaries.

```
void  main(int j)                int foo(int i,j)
{                                {
1: int i = 0;                    11: if (j > 0)
2: x = foo(i,j);                 12:   return bar(i)+1;
3: if (x == 1)                   13: else
4:   assert(false);              14:   return i+1;
}                                }
```

Figure 3. Example of must summary benefiting from not-may summary.

paths in g result in a return value greater than or equal to 0, this result is encoded as a not-may summary $\langle \mathbf{true} \stackrel{\neg\mathbf{may}}{\Longrightarrow}_{\mathtt{g}} (retval < 0)\rangle$. Once this not-may summary is computed, it can be used at all the call-sites of g in function `main` (at lines 1,2 and 3) to show that the assertion failure in line 5 can never be reached. □

Example 2. Consider the example program in Figure 2. Function `f` has two branches, one of which (the else branch) is hard to analyze since it invokes a complicated or even unknown hash function `h`. As before, SMASH first tries to find an execution along the path `1, 2, 3, 4, 5, 6, 7`. From the conditions in lines 4, 5 and 6, it incrementally collects the constraints $x1 > 0$, $x2 > 0$, and $x3 > 0$, and generates a query $\langle \mathbf{true} \stackrel{?}{\Longrightarrow}_{\mathtt{f}} (retval > 0)\rangle$ for function f where $retval$ denotes the return value of that function. SMASH now searches for an execution path in `f` that satisfies the postcondition $(retval > 0)$ and computes the must summary $\langle (i > 0) \stackrel{\mathbf{must}}{\Longrightarrow}_{\mathtt{f}} (retval > 0)\rangle$ by exploring only the "if" branch of the conditional at line 11 and avoiding the exploration of the complex function `h` in the "else" branch. Once this must summary is computed for `f`, a symbolic execution along the path `1, 2, 3, 4, 5, 6, 7` can reuse this summary (as in SMART [14, 1]) at the call sites at lines 1,2 and 3 without descending into `f` any further. Next, SMASH generates a test input for `main` satisfying the constraints $i1 > 0 \wedge i2 > 0 \wedge i3 > 0$ to prove that the assertion violation in line 7 can be reached. □

Next, we illustrate the interplay between not-may summaries and must summaries using simple examples.

Example 3. Consider the example in Figure 3. In this example, suppose `bar` is a complex function with nested function calls and a large number of paths, but one which already has a not-may summary $\langle (i = 0) \stackrel{\neg\mathbf{may}}{\Longrightarrow}_{\mathtt{bar}} (retval = 0)\rangle$. We show how this not-may summary can help with computing a must summary of `foo`.

During the analysis of `main`, SMASH tries to generate an execution that goes along the path `1,2,3,4` in function `main`. This

```
void main(int i,j)                 int foo(int i,j)
{                                  {
0: int x;                          10: int r,y;
1: if (i > 2 && j > 2) {           11: y = bar(i);
2:   x = foo(i,j);                 12: if (y > 10)
3:   if (x < 0)                    13:   r = g(j);
4:     assert(false);              14: else
   }                               15:   r = y;
}                                  16: return r;
                                   }
int g(int j)
{
20: if (j > 0)
21:   return j;
22: else
23:   return -j;
}
```

Figure 4. Example of not-may summary benefiting from must summary.

results in the query $\langle (i = 0) \stackrel{?}{\Longrightarrow}_{\mathtt{foo}} (retval = 1)\rangle$. This query results in SMASH searching through paths in `foo` for an execution satisfying the query. However, due to the not-may summary $\langle (i = 0) \stackrel{\neg\mathbf{may}}{\Longrightarrow}_{\mathtt{bar}} (retval = 0)\rangle$, SMASH is immediately able to conclude that none of the paths through `bar` can result in the desired post-condition $(retval = 1)$ for `foo` given that `i=0`. Thus, it explores only the "else" branch of the `if` statement in line 11 and generates the must summary $\langle (i = 0 \wedge j \leq 0) \stackrel{\mathbf{must}}{\Longrightarrow}_{\mathtt{foo}} (retval = 1)\rangle$. Note that while computing such a must summary, the SMASH algorithm uses the not-may summary of `bar` to conclude that no path through `bar` would result in the desired post condition, hence avoiding a wasteful search through the large number of paths in `bar`. Once the must summary for $\langle (i = 0 \wedge j \leq 0) \stackrel{\mathbf{must}}{\Longrightarrow}_{\mathtt{foo}} (retval = 1)\rangle$ is computed, SMASH uses it to analyze `main` and establish that any test case $j \leq 0$ violates the assertion in line 4. □

Example 4. Consider the example in Figure 4. In this example, suppose `bar` is a complex function (with loops etc.) whose set of possible return values is hard to characterize precisely (perhaps because the integers returned are prime numbers). Assume that a prior analysis of `bar` results in the following must summaries $\langle (i > 5) \stackrel{\mathbf{must}}{\Longrightarrow}_{\mathtt{bar}} (retval > 20)\rangle$ and $\langle (i < 5) \stackrel{\mathbf{must}}{\Longrightarrow}_{\mathtt{bar}} (retval < 5)\rangle$, obtained by symbolically executing specific paths (tests) in `bar`, as well as the not-may summary $\langle (true) \stackrel{\neg\mathbf{may}}{\Longrightarrow}_{\mathtt{bar}} (retval < 0)\rangle$. We now show how SMASH uses these must and not-may summaries for `bar` in order to compute a not-may summary for `foo` proving that the assertion in line 4 of `main` can never be reached.

SMASH first tries to find an execution along the path `0,1,2,3,4` in `main`. In order to do this, it generates a query $\langle (i > 2 \wedge j > 2) \stackrel{?}{\Longrightarrow}_{\mathtt{foo}} (x < 0)\rangle$ which leads to an analysis of `foo`. While analyzing `foo`, SMASH uses the available must-summaries $\langle (i > 5) \stackrel{\mathbf{must}}{\Longrightarrow}_{\mathtt{bar}} (retval > 20)\rangle$ and $\langle (i < 5) \stackrel{\mathbf{must}}{\Longrightarrow}_{\mathtt{bar}} (retval < 5)\rangle$ in order to prove that lines 13 and 15, respectively, are reachable (and therefore prevent a possibly expensive and hopeless not-may proof that one of those two branches is not feasible). Next, SMASH generates a query $\langle (j > 2) \stackrel{?}{\Longrightarrow}_{\mathtt{g}} (retval < 0)\rangle$. While analyzing the body of g with precondition $j > 2$, SMASH concludes that the return value $retval$ of g is always greater than 0, thus generating the not-may summary $\langle (j > 2) \stackrel{\neg\mathbf{may}}{\Longrightarrow}_{\mathtt{g}} (retval < 0)\rangle$. Subsequently, while analyzing the path `10,11,12,14,15,16`, SMASH generates the query $\langle (i > 2) \stackrel{?}{\Longrightarrow}_{\mathtt{bar}} (retval < 0)\rangle$ and uses the not-may summary $\langle (true) \stackrel{\neg\mathbf{may}}{\Longrightarrow}_{\mathtt{bar}} (retval < 0)\rangle$ to answer this query. This results in the not-may summary $\langle (i > 2 \wedge j > 2) \stackrel{\neg\mathbf{may}}{\Longrightarrow}_{\mathtt{foo}} (retval < 0)\rangle$ for `foo`. Using this not-may summary for `foo`, SMASH is able to prove that the assertion in line 4 of `main` cannot be reached. □

While it is intuitive that building not-may summaries improves may analysis, and must summaries improve must-analysis, our empirical results (see Section 5) revealed that using both not-may and must summaries together scaled better than just using may summaries or must summaries individually. We discovered the power of alternation in the process of understanding these empirical results. In Example 3, a not-may summary of `bar` was used to compute a must-summary of `foo`. In Example 4, a must-summary of `bar` was used to avoid a may-analysis over the return value of `bar` while still being able to build a not-may analysis of `foo`. Even though these examples are simple, they illustrate common patterns exhibited in large programs (see Section 5): some parts of large programs are more amenable to may analysis while other parts are more amenable to must analysis. In particular, for functions with many paths due to nested calls, conditionals and loops, but simple postconditions established in a post-dominator of all paths (such as the last statement before return), not-may summaries can be easy to compute. Such not-may summaries can be used to avoid an expensive search of (possibly infinitely many) explicit paths during must analysis, as we saw in Example 3. On the other hand, if a function has complex loops with complex loop invariants, a may analysis tends not to converge, while a must analysis of the same code can easily identify be a few feasible paths that traverse entirely the function and generate usable must summaries for those paths. Such must summaries can be used to avoid an expensive search for proofs in these parts of the code, as illustrated in Example 4. The tight integration between may and must summaries allows SMASH to alternate between both in a flexible and unprecedented manner. Further, our empirical results (see Figure 12 in Section 5) even quantify the amount of interaction between may analysis and must-analysis for our data set. On an average (1) to generate a proof, 68% of summaries used were not-may summaries and 32% of the summaries used were must summaries, and (2) to identify a bug, 64% of summaries used were must summaries and 36% of the summaries used were not-may summaries.

Note that must summaries are existential while not-may summaries are universal. Higher-level summaries of one type (existential or universal) are built up from lower-level summaries of the same type. The role played by lower-level summaries of one type when computing higher-level summaries of the other type is "only" to prove the non-existence of lower-level summaries of that other type, since any query cannot simultaneously have both a matching must-summary and a matching not-may summary. For instance, lower-level existential summaries help prove the non-existence of matching not-may lower-level summaries, hence avoiding unnecessary not-may analyses, and vice versa. With this in mind, using $must^-$ (backward) or $must$ (forward) summaries does not make much difference, as both summaries imply the non-existence of a matching not-may summary. In this paper, we use $must^-$ summaries for technical convenience. These considerations are formalized and discussed in detail in Section 4.

3. Single-Procedure Programs and May-Must Analysis

A program $\mathcal{P}$ has a finite set of variables $V_\mathcal{P}$. Each of these variables take values from an infinite domain $\mathcal{D}$ (such as integers or pointers). Informally, a program is specified by its variables and its control flow graph, which may contain cycles representing program loops. Each edge of the control flow graph maps to a statement. Initially, we consider only two types of statements: assignments and assume statements. Later we will add procedure calls and returns, when we consider multi-procedure programs.

Formally, a sequential program $\mathcal{P}$ is defined as a 6-tuple $\langle V_\mathcal{P}, N_\mathcal{P}, E_\mathcal{P}, n^0_\mathcal{P}, n^x_\mathcal{P}, \lambda_\mathcal{P}\rangle$ where

1. $V_\mathcal{P}$ is a finite set of variables that the program manipulates (each variable takes values from an infinite domain $\mathcal{D}$),
2. $N_\mathcal{P}$ is a finite set of nodes (or program locations),
3. $E_\mathcal{P} \subseteq N_\mathcal{P} \times N_\mathcal{P}$ is a finite set of edges,
4. $n^0_\mathcal{P} \in N_\mathcal{P}$ is a distinguished entry node,
5. $n^x_\mathcal{P} \in N_\mathcal{P}$ is a distinguished exit node,
6. $\lambda_\mathcal{P} : E_\mathcal{P} \rightarrow \mathsf{Stmts}$, maps each edge to a statement in the program,

We consider two types of statements: (1) *assignment* statements are of the form `x := e` where `x` is a variable and `e` is a side-effect free expression over variables and constants, and (2) *assume* (or conditional) statements are of the form `assume(e)`, where `e` is a side-effect free expression. A configuration of a program $\mathcal{P}$ is a pair $\langle n, \sigma\rangle$ where $n \in N_\mathcal{P}$ and σ is a *state*, defined below. A state of a program $\mathcal{P}$ is a valuation to the program variables $V_\mathcal{P}$. The set of all states of $\mathcal{P}$ is denoted by $\Sigma_\mathcal{P}$.

An assignment statement is a function $\Sigma_\mathcal{P} \rightarrow \Sigma_\mathcal{P}$ since it maps every state σ to a state obtained by executing the assignment. An assume statement `assume(e)` is a partial function over $\Sigma_\mathcal{P}$. If e evaluates to true in a state σ, then the function maps σ to itself. Otherwise, it is undefined over σ.

Thus, every edge $e \in E_\mathcal{P}$ can be thought of as a relation $\Gamma_e \subseteq \Sigma_\mathcal{P} \times \Sigma_\mathcal{P}$. So far, Γ_e is actually a partial function, but it is convenient to think of it as a relation and the generality will help when we consider multi-procedure programs.

Note that though each statement of the program is deterministic (i.e, Γ_e is a partial function), the program $\mathcal{P}$ has control nondeterminism, since nodes in $N_\mathcal{P}$ can have multiple outgoing edges in $E_\mathcal{P}$.

The initial configurations of a program $\mathcal{P}$ are given by the set $\{\langle n^0_\mathcal{P}, \sigma\rangle \mid \sigma \in \Sigma_\mathcal{P}\}$. That is, the node $n^0_\mathcal{P}$ is the entry node of the program, and the state σ is any possible state where variables can take any values. From any configuration $\langle n, \sigma\rangle$, the program can execute a *step* by taking any outgoing edge of $e = \langle n, n'\rangle$ out of n and transitioning to a state obtained by computing the image of the relation Γ_e with respect to the state σ. That is, if there exists σ' such that $\Gamma_e(\sigma, \sigma')$, then the program can transition to the configuration $\langle n', \sigma'\rangle$. Starting from an initial configuration, a program can execute several steps producing several configurations.

A verification question for program $\mathcal{P}$ is formalized as a reachability query of the form $\langle \varphi_1 \stackrel{?}{\Longrightarrow}_\mathcal{P} \varphi_2\rangle$, where φ_1 represents a set of initial states at entry node $n^0_\mathcal{P}$, and φ_2 represents a set of states at a exit node $n^x_\mathcal{P}$. The reachability query evaluates to "yes" (meaning the program is incorrect) if there exists an execution which starts at a configuration $\langle n^0_\mathcal{P}, \sigma_1\rangle$ with $\sigma_1 \in \varphi_1$ and ends at a configuration $\langle n^x_\mathcal{P}, \sigma_2\rangle$ with $\sigma_2 \in \varphi_2$, it evaluates to "no" if such an execution provably does not exist, or it evaluates to "unknown" otherwise. Indeed, the verification question is undecidable in general, particularly if the domain $\mathcal{D}$ of values taken by variables is infinite. Even if $\mathcal{D}$ is finite, the number of configurations is exponential in the number of variables $V_\mathcal{P}$ making an exact answer to this question computationally difficult to ascertain. Thus, in practice, approximate methods are used to answer the verification question.

Note that specifications in the style of Hoare triples such as $\{\varphi_1\}\ P\ \{\varphi_2\}$, where we want all executions starting from states in φ_1 to end in states from φ_2, can be expressed in our notation as the query $\langle \varphi_1 \stackrel{?}{\Longrightarrow}_\mathcal{P} \neg\varphi_2\rangle$. Assertions in the form used in the example programs from Section 2 can also be expressed in this form by adding a special boolean variable $error$ which is set to **false** initially and set to **true** if an assertion fails, enabling us to state an equivalent specification $\langle \Sigma_\mathcal{P} \stackrel{?}{\Longrightarrow}_\mathcal{P} error\rangle$.

$$\frac{\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle}{\Omega_{n^0_{\mathcal{P}}} := \hat{\varphi}_1 \quad \forall n \in N_{\mathcal{P}} \setminus \{n^0_{\mathcal{P}}\}.\Omega_n := \{\}} [\text{ INIT-OMEGA }]$$

$$\frac{e = \langle n_1, n_2 \rangle \in E_{\mathcal{P}} \quad \theta \subseteq \mathsf{Post}(\Gamma_e, \Omega_{n_1})}{\Omega_{n_2} := \Omega_{n_2} \cup \theta} [\text{ MUST-POST }]$$

$$\frac{\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle \quad \Omega_{n^x_{\mathcal{P}}} \cap \hat{\varphi}_2 \neq \{\}}{\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle \;=\; \text{yes}} [\text{ BUGFOUND }]$$

Figure 6. Must analysis.

We recall the formal definitions of the preimage Pre and postcondition Post operators that will be used later. Suppose $\Gamma_e \subseteq \Sigma_{\mathcal{P}} \times \Sigma_{\mathcal{P}}$, and $\varphi \subseteq \Sigma_{\mathcal{P}}$. The *precondition* of φ with respect to Γ_e (denoted $\mathsf{Pre}(\Gamma_e, \varphi)$) is the set of all predecessors of states in φ given by

$$\mathsf{Pre}(\Gamma_e, \varphi) \overset{\text{def}}{=} \{\sigma \in \Sigma_{\mathcal{P}} \mid \exists \sigma' \in \varphi. \Gamma_e(\sigma, \sigma')\}$$

In a dual way, the postcondition of φ with respect to Γ_e (denoted $\mathsf{Post}(\Gamma_e, \varphi)$) is given by

$$\mathsf{Post}(\Gamma_e, \varphi) \overset{\text{def}}{=} \{\sigma \in \Sigma_{\mathcal{P}} \mid \exists \sigma' \in \varphi. \Gamma_e(\sigma', \sigma)\}$$

3.1 May Analysis

A *may analysis* of a program is used to prove that the program never reaches an error node during any execution. Formally, a *may analysis* of a program $\mathcal{P}$ associates every node n in $N_{\mathcal{P}}$ with a finite partition Π_n of $\Sigma_{\mathcal{P}}$, and every edge $e = \langle n_1, n_2 \rangle \in E_{\mathcal{P}}$ with a set of edges $\Pi_e \subseteq \Pi_{n_1} \times \Pi_{n_2}$, such that, for any region π_1 in Π_{n_1}, and any region π_2 in Π_{n_2}, if $\exists \sigma_1 \in \pi_1. \exists \sigma_2 \in \pi_2. \Gamma_e(\sigma_1, \sigma_2)$ then $\langle \pi_1, \pi_2 \rangle \in \Pi_e$.

By construction, the edges Π_e *over-approximate* the transition relation of the program, and are called *may* edges or transitions. A partition Π_n for a node n is defined as a set of regions. Associated with every program edge $e = \langle n_1, n_2 \rangle$, initially there is a may edge from every region of the partition Π_{n_1} to every region of the partition Π_{n_2}. As the algorithm proceeds, partitions get refined and may edges get deleted. Instead of deleting edges, we find it notationally convenient to maintain the complement of may edges, called N_e edges below. The set of N_e edges grows monotonically as the algorithm proceeds.

In response to a query $\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle$, if there is no path through the edges Π_e from every region φ_1 associated program's entry node $n^0_{\mathcal{P}}$ such that $\varphi_1 \cap \hat{\varphi}_1 \neq \{\}$ to every region $\hat{\varphi}_2$ associated with the exit node $n^x_{\mathcal{P}}$ such that $\varphi_2 \cap \hat{\varphi}_2 \neq \{\}$, then the may analysis proves that none of the states in $\hat{\varphi}_2$ can be reached at the exit node of $\mathcal{P}$ by starting the program with states from $\hat{\varphi}_1$. Figure 5 gives a set of declarative rules to perform a may analysis which automatically refines partitions in a demand-driven manner.

In response to a query $\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle$, the rule INIT-PI-NE initializes the exit node $n^x_{\mathcal{P}}$ with a partition consisting of two regions $\hat{\varphi}_2$ and $\Sigma_{\mathcal{P}} \setminus \hat{\varphi}_2$. All other nodes are initialized to have a single partition with all possible states $\Sigma_{\mathcal{P}}$. The current set of rules performs *backward* may analysis, using the Pre operator. Thus, to allow maximum flexibility, we do not partition the initial node with the precondition $\hat{\varphi}_1$ from the query. The precondition $\hat{\varphi}_1$ is used in the last rule VERIFIED described below. The rule also initializes an empty relation N_e associated with each program edge e. Relation N_e is the complement of Π_e and is used to keep track of may edges that we know for sure do *not* exist.

Abstract partition refinement is performed using the rule NOTMAY-PRE. The rule NOTMAY-PRE chooses two nodes n_1 and n_2, a region φ_1 in the partition Π_{n_1} of n_1, and a region φ_2 in the partition Π_{n_2} of n_2, and then *splits* φ_1 using the precondition θ of φ_2 with respect to the transition relation on the edge $e = \langle n_1, n_2 \rangle$. We denote splitting the region φ_1 in partition Π_{n_1} into two sub-regions $\varphi_1 \cap \theta$ and $\varphi_1 \cap \neg\theta$ by $\Pi_{n_1} := (\Pi_{n_1} \setminus \{\varphi_1\}) \cup \{\varphi_1 \cap \theta, \varphi_1 \cap \neg\theta\}$. After the split, we know by construction and the definition of Pre that any state in the new region $\varphi_1 \cap \neg\,\theta$ *cannot* possibly lead to a state in region φ_2, and we record this in relation N_e accordingly. Note that a superset of the precondition can be used as valid approximations to do splits. Such approximations are necessary in practice whenever Pre cannot be computed precisely.

A dual rule NOTMAY-POST for splitting regions using the operator Post in the forward direction exists but is omitted here.

The IMPL-RIGHT rule allows N_e edges of the form (φ_1, φ_2) to be maintained as the post-regions φ_2 get refined. The rules IMPL-LEFT rule allows N_e edges of the form (φ_1, φ_2) to be maintained as the pre-regions φ_1 get refined.

The VERIFIED rule says that as soon as *all* paths from the entry node regions that intersect with the precondition of the query $\hat{\varphi}_1$ to the exit node region that intersects with the postcondition of the query $\hat{\varphi}_2$ have at least one step where the may analysis shows that the edge does *not* exist, then we have verified that the answer to the query $\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle$ is "no" and hence that the program is correct.

It is easy to show that, at every refinement step, the transition system defined over the partitioned regions simulates the program $\mathcal{P}$. Thus, the incremental inference of relation N_e made by the may analysis is always guaranteed to be sound with respect to $\mathcal{P}$. However, there is no guarantee that, if the program is correct, the refinement process will ever converge to such an answer. For finite-state programs, the process is guaranteed to terminate.

The partition splits specified in the rules of Figure 5 are nondeterministic. Specific abstraction-refinement tools such as SLAM [4] can be viewed as instantiating this framework by refining regions only along abstract counterexamples that lead from the starting node to the error node.

3.2 Must Analysis

A *must analysis* of a program is used to prove that the program reaches a given set of states during some execution. While a may analysis over-approximates reachability information in order to prove the absence of errors, a must analysis under-approximates reachability information in order to prove the existence of execution paths leading to an error. A must analysis based on successive partition refinements can be defined by dualizing the rules presented in the previous section. However, splitting must partitions is no longer guaranteed to preserve previously-computed must transitions and other techniques (such as hyper-must transitions or cartesian abstraction) are needed to restore the monotonicity of must-partition refinement [15].

In this paper, we consider a specific type of must abstractions which avoid the issues above. Specifically, a *must analysis* of a program $\mathcal{P}$ associates every node n in $N_{\mathcal{P}}$ with a set Ω_n of states that are all guaranteed to be reachable from the initial state of the program. The set Ω_n of states associated with a node n increases monotonically during the must analysis and is never partitioned.

Figure 6 gives a set of declarative rules to perform a must analysis using sets Ω_n.

In response to a query $\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle$, we initialize $\Omega_{n^0_{\mathcal{P}}}$ with $\hat{\varphi}_1$ and for all other nodes n we initialize Ω_n to be the empty set (rule INIT-OMEGA).

The rule MUST-POST specifies how to perform a forward must-analysis using the postcondition operator Post: the postcondition θ of Ω_{n_1} with respect to the transition relation associated with an edge e from node n_1 to node n_2 can be safely added to Ω_{n_2}. If the postcondition Post cannot be computed precisely, any subset is a

$$\frac{\langle \hat{\varphi}_1 \stackrel{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle}{\Pi_{n^x_{\mathcal{P}}} := \{\hat{\varphi}_2, \Sigma_{\mathcal{P}} \setminus \hat{\varphi}_2\} \quad \forall n \in N_{\mathcal{P}} \setminus \{n^x_{\mathcal{P}}\}.\Pi_n := \{\Sigma_{\mathcal{P}}\} \quad \forall e \in E_{\mathcal{P}}.N_e := \{\}} [\text{ INIT-PI-NE }]$$

$$\frac{\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad e = \langle n_1, n_2 \rangle \in E_{\mathcal{P}} \quad \theta \supseteq \mathsf{Pre}(\Gamma_e, \varphi_2)}{\Pi_{n_1} := (\Pi_{n_1} \setminus \{\varphi_1\}) \cup \{\varphi_1 \cap \theta, \varphi_1 \cap \neg\theta\} \quad N_e := N_e \cup \{(\varphi_1 \cap \neg\,\theta, \varphi_2)\}} [\text{ NOTMAY-PRE }]$$

$$\frac{(\varphi_1, \varphi_2) \in N_e \quad \varphi_1' \subseteq \varphi_1}{N_e := N_e \cup \{(\varphi_1', \varphi_2)\}} [\text{ IMPL-LEFT }] \qquad \frac{(\varphi_1, \varphi_2) \in N_e \quad \varphi_2' \subseteq \varphi_2}{N_e := N_e \cup \{(\varphi_1, \varphi_2')\}} [\text{ IMPL-RIGHT }]$$

$$\frac{\begin{array}{c}\langle \hat{\varphi}_1 \stackrel{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle \\ \forall n_0, \ldots, n_k. \forall \varphi_0, \ldots, \varphi_k. \quad n_0 = n^0_{\mathcal{P}} \wedge n_k = n^x_{\mathcal{P}} \wedge \varphi_0 \in \Pi_{n_0} \wedge \varphi_1 \in \Pi_{n_1} \cdots \varphi_k \in \Pi_{n_k} \wedge \varphi_0 \cap \hat{\varphi}_1 \neq \{\} \wedge \varphi_k \cap \hat{\varphi}_2 \neq \{\} \\ \Rightarrow \exists 0 \leq i \leq k-1. e = \langle n_i, n_{i+1} \rangle \in E_{\mathcal{P}} \Rightarrow (\varphi_i, \varphi_{i+1}) \in N_e\end{array}}{\langle \hat{\varphi}_1 \stackrel{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle \;=\; \text{no}} [\text{ VERIFIED }]$$

Figure 5. May analysis.

$$\frac{\begin{array}{c}\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad e = \langle n_1, n_2 \rangle \in E_{\mathcal{P}} \\ \Omega_{n_1} \cap \varphi_1 \neq \{\} \quad \Omega_{n_2} \cap \varphi_2 = \{\} \quad \theta \subseteq \mathsf{Post}(\Gamma_e, \Omega_{n_1} \cap \varphi_1) \quad \varphi_2 \cap \theta \neq \{\}\end{array}}{\Omega_{n_2} := \Omega_{n_2} \cup \theta} [\text{ MUST-POST }]$$

$$\frac{\begin{array}{c}\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad e = \langle n_1, n_2 \rangle \in E_{\mathcal{P}} \\ \Omega_{n_1} \cap \varphi_1 \neq \{\} \quad \Omega_{n_2} \cap \varphi_2 = \{\} \quad \beta \supseteq \mathsf{Pre}(\Gamma_e, \varphi_2) \quad \beta \cap \Omega_{n_1} = \{\}\end{array}}{\Pi_{n_1} := (\Pi_{n_1} \setminus \{\varphi_1\}) \cup \{\varphi_1 \cap \beta, \varphi_1 \cap \neg\beta\} \quad N_e := N_e \cup \{(\varphi_1 \cap \neg\beta, \varphi_2)\}} [\text{ NOTMAY-PRE }]$$

Figure 7. May-must analysis. Rules INIT-PI-NE, IMPL-LEFT, IMPL-RIGHT and VERIFIED are assumed to be included from Figure 5, and rules INIT-OMEGA and BUGFOUND are assumed to be included from Figure 6.

valid approximation. This formalization includes as a specific case DART [16], where complex constraints are simplified by substituting symbolic expressions with concrete values observed dynamically during testing.

A rule MUST-PRE for backward must-analysis using the precondition operator Pre can be written as the dual of rule MUST-POST, but is omitted here.

During the analysis of a query $\langle \hat{\varphi}_1 \stackrel{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2 \rangle$, if $\Omega_{n^x_{\mathcal{P}}}$ ever intersects with $\hat{\varphi}_2$, then we can conclude that the answer to the query is "yes" (rule BUGFOUND).

3.3 May-Must Analysis

May and must analysis have complementary properties – the former is used for verification and the latter for bug finding. Figure 7 shows a set of rules that combine may and must rules in order to perform both a may and a must analysis simultaneously. The rules INIT-PI-NE, IMPL-LEFT, IMPL-RIGHT and VERIFIED are included "as is" from Figure 5, and rules INIT-OMEGA and BUGFOUND are included "as is" from Figure 6, so we do not repeat them here.

The interesting rules are MUST-POST and NOTMAY-PRE. Suppose edge $e = \langle n_1, n_2 \rangle$ is such that $\varphi_1 \in \Pi_{n_1}$ and $\varphi_2 \in \Pi_{n_2}$. Furthermore, suppose that $\varphi_1 \cap \Omega_{n_1} \neq \{\}$, and $\varphi_2 \cap \Omega_{n_2} = \{\}$. This means that we know that the partition $\varphi_1 \in \Pi_{n_1}$ is indeed reachable (since it intersects with the region Ω_{n_1}), and we do not know if $\varphi_2 \in \Pi_{n_2}$ is reachable (since it does not intersect with the region Ω_{n_2}). If we can compute a subset of the postcondition $\theta \subseteq \mathsf{Post}(\Gamma_e, \varphi_1 \cap \Omega_{n_1})$ such that θ intersects with φ_2, then we simply augment Ω_{n_2} with θ, as specified in the rule MUST-POST. Dually, if we can compute a superset of the precondition $\beta \supset \mathsf{Pre}(\Gamma_e, \varphi_2)$ such that β does not intersect with Ω_{n_1}, then we split the region $\varphi_1 \in \Pi_{n_1}$ using β and we know that region φ_2 is not reachable from region $\varphi_1 \cap \neg\beta$, as specified in the rule NOTMAY-PRE. Note that any interpolant (see [26]) between the sets $\mathsf{Pre}(\Gamma_e, \varphi_2)$ and Ω_{n_1} satisfies the conditions to be used as β, and can be used to split the region φ_1.

These rules can be instantiated to obtain the SYNERGY [19] and DASH [5] algorithms. These algorithms have the specific property that, whenever one can compute precisely Post, equivalently Pre, then either rule MUST-POST or rule NOTMAY-PRE must be enabled, and a single call to a theorem prover to compute Post/Pre is then sufficient to refine either the current must or may (respectively) program abstraction.

4. Multi-Procedure Programs and Compositional May-Must Analysis

In programs with multiple procedures, *compositional* analyses, which analyze one procedure at a time and build summaries for each procedure for possible re-use in other calling contexts, are desired for scalability. We describe how to do compositional analysis with may abstractions, must abstractions and by combining may-must abstractions. First, we extend our program notation to allow programs with multiple procedures.

A multi-procedure program $\mathcal{P}$ is a set of single-procedure programs $\{\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n\}$. Each of the single-procedure programs follows the notation described earlier, with the following modifications. There are *global variables* common to all procedures, and *local variables* private to each procedure (more precisely, private to each invocation of a procedure). Thus, for any single-procedure program $\mathcal{P}_i$ (also called procedure $\mathcal{P}_i$), we have that $\mathcal{P}_i = \langle V_{\mathcal{P}_i}, N_{\mathcal{P}_i}, E_{\mathcal{P}_i}, n^0_{\mathcal{P}_i}, n^x_{\mathcal{P}_i}, \lambda_{\mathcal{P}_i} \rangle$ where $V_{\mathcal{P}_i}$ is the disjoint union of global variables V^G and local variables $V^L_{\mathcal{P}_i}$. Parameters and return values can be simulated using global variables, so we do not explicitly model these. Without loss of generality, we assume each procedure has a single entry node $n^0_{\mathcal{P}_i}$ and a single exit node $n^x_{\mathcal{P}_i}$. For any two distinct procedures $\mathcal{P}_i$ and $\mathcal{P}_j$, we assume that $N_{\mathcal{P}_i}$ and $N_{\mathcal{P}_j}$ are disjoint. Thus a node unambiguously identifies which procedure it is in.

In addition to the assignment and assume statements, we add a new statement `call p` where `p` is the name of the procedure being called. Procedure $\mathcal{P}_0$ is the "main" procedure where the program starts executing. As before, all global variables initialize nondeterministically to any value in the initial configuration. Whenever a procedure is entered, its local variables are initialized nondeterministically. Unlike local variables, global variables get initialized

$$\frac{\text{procedure } \mathcal{P}_i \in \mathcal{P}}{\overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i} := \{\}} [\text{ INIT-NOTMAYSUM }]$$

$$\frac{\begin{array}{c}\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad e = \langle n_1, n_2\rangle \in E_{\mathcal{P}_i} \text{ is a call to procedure } \mathcal{P}_j \\ (\hat{\varphi}_1, \hat{\varphi}_2) \in \overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_j} \quad \varphi_2 \subseteq \hat{\varphi}_2 \quad \theta \subseteq \hat{\varphi}_1\end{array}}{\Pi_{n_1} := (\Pi_{n_1} \setminus \{\varphi_1\}) \cup \{\varphi_1 \cap \theta, \varphi_1 \cap \neg\theta\} \quad N_e := N_e \cup \{(\varphi_1 \cap \theta, \varphi_2)\}} [\text{ NOTMAY-PRE-USESUMMARY }]$$

$$\frac{\begin{array}{c}\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad e = \langle n_1, n_2\rangle \in E_{\mathcal{P}_i} \text{ is a call to procedure } \mathcal{P}_j \\ \psi_1 = \exists V^L_{\mathcal{P}_i}.\varphi_1 \quad \psi_2 = \exists V^L_{\mathcal{P}_i}.\varphi_2\end{array}}{\langle \psi_1 \overset{?}{\Longrightarrow}_{\mathcal{P}_j} \psi_2\rangle} [\text{ MAY-CALL }]$$

$$\frac{\begin{array}{c}\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}_i} \hat{\varphi}_2\rangle \\ \forall n_0, \ldots, n_k. \forall \varphi_0, \ldots, \varphi_k. \quad n_0 = n^0_{\mathcal{P}_i} \wedge n_k = n^x_{\mathcal{P}_i} \wedge \varphi_0 \in \Pi_{n_0} \wedge \varphi_1 \in \Pi_{n_1} \cdots \varphi_k \in \Pi_{n_k} \wedge \varphi_0 \cap \hat{\varphi}_1 \neq \{\} \wedge \varphi_k \cap \hat{\varphi}_2 \neq \{\} \\ \Rightarrow \exists 0 \leq i \leq k-1. e = \langle n_i, n_{i+1}\rangle \in E_{\mathcal{P}} \Rightarrow (\varphi_i, \varphi_{i+1}) \in N_e\end{array}}{\overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i} := \overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i} \cup (\exists V^L_{\mathcal{P}_i}.\varphi_0, \exists V^L_{\mathcal{P}_i}.\varphi_k)} [\text{ CREATE-NOTMAYSUMMARY }]$$

$$\frac{(\varphi_1, \varphi) \in \overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i} \quad (\varphi_2, \varphi) \in \overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i}}{\overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i} := \overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i} \cup \{(\varphi_1 \cup \varphi_2, \varphi)\}} [\text{ MERGE-MAYSUMMARY }]$$

Figure 8. Compositional may analysis. Rules from Figure 5 are assumed to be included, but not shown.

only once at the beginning of execution, and can be used for communication between procedures.

We use the query notation to specify verification questions for multi-procedure programs as well. In particular, for multi-procedure programs, we assume that the query is asked in terms of the main procedure $\mathcal{P}_0$, and is of the form $\langle \varphi_1 \overset{?}{\Longrightarrow}_{\mathcal{P}_0} \varphi_2\rangle$. Our compositional analyses solve the verification question by formulating various sub-queries to procedures that are called from $\mathcal{P}_0$, and then sub-queries to procedures called by those procedures and so on. Each sub-query is defined in the context of a particular procedure, and is solved by analyzing the code of that procedure in combination with summaries for other called procedures.

We have presented the rules in Section 3 in such a way that they can easily be extended to work with summaries. In particular, the edges N_e from Figure 5 can be used to build not-may summaries, and the sets Ω_n from Figure 6 can be used to build must summaries.

Given a set $S \subseteq \Sigma_{\mathcal{P}_i}$, we use $\exists V^L_{\mathcal{P}_i}.S$ to denote the set of global states obtained by considering only the values of global variables of each state in S.

4.1 Compositional May Analysis

Figure 8 gives the rules for compositional may analysis. The rules for intraprocedural may analysis from Figure 5 are assumed to be included and are not repeated.

The rules maintain a set $\overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i}$ of not-may summaries for each procedure $\mathcal{P}_i$. The rule INIT-NOTMAYSUM initializes the set of not-may summaries of all procedures to empty sets. The rule NOTMAY-PRE-USESUMMARY uses an existing not-may summary to split a region $\varphi \in \Pi_{n_1}$ along the lines of the NOTMAY-PRE rule. Recall that if $\langle \hat{\varphi}_1 \overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i} \hat{\varphi}_2\rangle$ is a not-may summary, then there is no transition from any state in $\hat{\varphi}_1$ to any state in $\hat{\varphi}_2$. Thus, for any subset θ of $\hat{\varphi}_1$ there exists no transition from any state in θ to any state in $\varphi_2 \subseteq \hat{\varphi}_2$. This justifies the N_e edge $\{(\varphi_1 \cap \theta, \varphi_2)\}$ in the consequent of the rule.

The rule MAY-CALL generates a query in the context of a called procedure after existentially quantifying local variables of the caller since those are not in the scope of the callee. For notational convenience, we assume that the partitions Π_n for each node n and the edges N_e are computed afresh for each invocation of a query. (Note that intraprocedural inference steps could themselves be summarized and re-used across multiple query invocations.)

The rule CREATE-NOTMAYSUMMARY is used to generate a not-may summary from N_e edges. If all paths from some region φ_0 in $\Pi_{n^0_{\mathcal{P}_i}}$ to some region φ_k in $\Pi_{n^x_{\mathcal{P}_i}}$ pass through at least one N_e edge, we can conclude that there are no paths from states in φ_0 to states in φ_k. Thus, we can add a not-may summary between these two sets of states after quantifying out the local variables that are irrelevant to the calling contexts.

The rule MERGE-MAYSUMMARY allows not-may summaries to be merged. The correctness of this rule follows from the definition of not-may summaries. If there are no paths from states in φ_1 to states in φ, and there are no paths from states in φ_2 to states in φ, we can conclude that there are no paths from states in $\varphi_1 \cup \varphi_2$ to states in φ. Merged summaries can contribute to larger sets θ used to split regions in the NOTMAY-PRE-USESUMMARY rule.

4.2 Compositional Must Analysis

The rules in Figure 9 along with the intraprocedural rules described in Figure 6 define compositional must analysis.

The set of must-summaries for each procedure $\mathcal{P}_i$ is denoted by $\overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_i}$. This set is initialized to the empty set in the rule INIT-NOTMAYSUM, and it monotonically increases as the analysis proceeds.

The rule MUST-POST-USESUMMARY is very similar to the rule MUST-POST in the intraprocedural must analysis. The only difference here is that the node n_1 is a call node, representing a call to another procedure $\mathcal{P}_j$. If a suitable must summary (φ_1, φ_2) exists, the rule uses the must summary to compute an underapproximation to the postcondition.

The rule MUST-CALL creates a new sub-query for a called procedure $\mathcal{P}_j$, which can result in new summaries created for $\mathcal{P}_j$. As in the compositional may analysis case, we assume again for notational simplicity that the sets Ω_n for each node n are computed afresh for each query.

The rule CREATE-MUSTSUMMARY creates must summaries in the context of the current query for the procedure. Suppose $\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}_i} \hat{\varphi}_2\rangle$ is the current query for the procedure $\mathcal{P}_i$. Suppose $\theta = \exists V^L_{\mathcal{P}_i}.\Omega_{n^x_{\mathcal{P}_i}}$ represents the global state reached at the exit point of the procedure $\mathcal{P}_i$ by using must analysis. Further suppose that θ intersects with the post-state of the query φ_2. Then, we add the must summary $\{(\hat{\varphi}_1, \theta)\}$ since every state in θ can be obtained by executing the procedure starting at some state in φ_1.

Finally the rule MERGE-MUSTSUMMARY allows merging must summaries. The correctness of this rule follows from the definition of must summaries. If every state in φ_1 can be reached from some state in φ, and every state in φ_2 can be reached from

$$\frac{\text{procedure } \mathcal{P}_i \in \mathcal{P}}{\overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_i} := \{\}} [\text{ INIT-MUSTSUMMARY }]$$

$$\frac{e = \langle n_1, n_2\rangle \in E_{\mathcal{P}_i} \quad \text{is a call to procedure } \mathcal{P}_j \quad (\varphi_1, \varphi_2) \in \overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_j} \quad \Omega_{n_1} \supseteq \varphi_1 \quad \theta \subseteq \varphi_2}{\Omega_{n_2} := \Omega_{n_2} \cup \theta} [\text{ MUST-POST-USESUMMARY }]$$

$$\frac{e = \langle n_1, n_2\rangle \in E_{\mathcal{P}_i} \quad \text{is a call to procedure } \mathcal{P}_j \quad \theta = \exists V^L_{\mathcal{P}_i}.\Omega_{n_1}}{\langle \theta \overset{?}{\Longrightarrow}_{\mathcal{P}_j} \Sigma_{\mathcal{P}}\rangle} [\text{ MUST-CALL }]$$

$$\frac{\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}_i} \hat{\varphi}_2\rangle \quad \theta = \exists V^L_{\mathcal{P}_i}.\Omega_{n^x_{\mathcal{P}_i}} \quad \theta \cap \hat{\varphi}_2 \neq \{\}}{\overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_i} := \overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_i} \cup \{(\hat{\varphi}_1, \theta)\}} [\text{ CREATE-MUSTSUMMARY }]$$

$$\frac{(\varphi, \varphi_1) \in \overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_i} \quad (\varphi, \varphi_2) \in \overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_i}}{\overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_i} := \overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_i} \cup \{(\varphi, \varphi_1 \cup \varphi_2)\}} [\text{ MERGE-MUSTSUMMARY }]$$

Figure 9. Compositional must analysis. Rules from Figure 6 are assumed to be included, but not shown.

some state in φ, it follows that every state in $\varphi_1 \cup \varphi_2$ can be reached from some state in φ.

These set of rules can be instantiated to obtain a variant of the SMART algorithm [14], a compositional version of the DART algorithm. Indeed, our rules compute summaries for specific calling contexts (see θ in rule MUST-CALL) and record those in the preconditions of summaries. In contrast, preconditions of summaries in SMART are expressed exclusively in terms of the procedure's input variables and calling contexts themselves are not recorded. The summaries of SMART can therefore be more general (hence more re-usable), while our summaries record must-reachability information more precisely, which in turns simplifies the formalization and the combination with not-may summaries, as is discussed next.

4.3 SMASH: Compositional May-Must Analysis

The rules for compositional may-must analysis combine the may rules and the must rules in the same way as in the single procedure case. The set of rules is shown in Figure 10. All rules from compositional may analysis (Figure 8), compositional must analysis (Figure 9), and intraprocedural may-must analysis (Figure 7) are included but not shown, except the rules MAY-CALL and MUST-CALL which are replaced by the new rule MAYMUST-CALL, and except the rules MUST-POST-USESUMMARY and NOTMAY-PRE-USESUMMARY which are modified as shown in Figure 10.

The succinctness of these rules hides how not-may summaries and must summaries interact, so we explain this in more detail. Each query made to SMASH can be answered with either a not-may summary or a must summary, but not both. Whenever SMASH analyzes a procedure $\mathcal{P}_i$ and encounters an edge e in $\mathcal{P}_i$ calling another procedure $\mathcal{P}_j$, SMASH can either use an existing must summary for procedure $\mathcal{P}_j$ (using the rule MUST-POST-USESUMMARY), or an existing not-may summary for $\mathcal{P}_j$ (using the rule NOTMAY-PRE-USESUMMARY), or initiate a fresh query to $\mathcal{P}_j$ (using the rule MAY-MUST-CALL). Note that when the rule MAY-MUST-CALL generates a query for the called procedure $\mathcal{P}_j$, we do not know if the query will result in creating not-may summaries or must summaries. If the body of $\mathcal{P}_j$ calls another procedure $\mathcal{P}_k$, another query can be potentially generated for that procedure as well. Eventually, some of the calls could be handled using must summaries (using the rule MOST-POST-USESUMMARY), and some of the calls using not-may summaries (using the rule NOTMAY-PRE-USESUMMARY). Thus, not-may summaries can be used to create must summaries and vice versa, as illustrated in Section 2 with Examples 3 and 4.

The rule MUST-POST-USESUMMARY is similar to the rule MUST-POST in Figure 7. The main difference is that the edge e is a call to another procedure $\mathcal{P}_j$. Assuming a suitable must summary exists, the rule uses the must summary as the transition relation of the procedure call in order to perform the equivalent of a Post computation. Suppose edge $e = \langle n_1, n_2\rangle$ is such that $\varphi_1 \in \Pi_{n_1}$ and $\varphi_2 \in \Pi_{n_2}$. Furthermore, suppose that $\varphi_1 \cap \Omega_{n_1} \neq \{\}$, $\varphi_2 \cap \Omega_{n_2} = \{\}$, and $(\hat{\varphi}_1, \hat{\varphi}_2)$ is a must-summary for $\mathcal{P}_j$, with $\Omega_{n_1} \supseteq \hat{\varphi}_1$. Since $(\hat{\varphi}_1, \hat{\varphi}_2)$ is a must-summary, we know that all the states in $\hat{\varphi}_2$ are reachable by executions of $\mathcal{P}_j$ from states in $\hat{\varphi}_1$. Since $\Omega_{n_1} \supseteq \hat{\varphi}_1$, this implies that any subset $\theta \subseteq \hat{\varphi}_2$ can be reached from the set of sates Ω_{n_1} by executing the procedure $\mathcal{P}_j$. Thus, we can add θ to the set of states Ω_{n_2}, which are the set of states that are guaranteed to be reachable at node n_2 during this execution of procedure $\mathcal{P}_j$.

Similarly, the rule NOTMAY-PRE-USESUMMARY is like the rule NOTMAY-PRE in Figure 7. The main difference is again that the edge e is a call to another procedure $\mathcal{P}_j$. Assuming a suitable not-may summary exists, the rule uses the not-may summary as the transition relation of the procedure call to perform the equivalent of a Pre computation. Suppose e, φ_1 and φ_2 satisfy the same assumptions as above, and there is a not-may summary $(\hat{\varphi}_1, \hat{\varphi}_2)$ with $\varphi_2 \subseteq \hat{\varphi}_2$. Since $(\hat{\varphi}_1, \hat{\varphi}_2)$ is a not-may summary, we know that there are no executions of the procedure $\mathcal{P}_j$ starting at any subset $\theta \subseteq \hat{\varphi}_1$ resulting in states in $\hat{\varphi}_2$. Thus, we can partition the region $\varphi_1 \in \Pi_{n_1}$ with the guarantee that there are no transitions from $\varphi_1 \cap \theta$ to φ_2, and hence we can add $(\varphi_1 \cap \theta, \varphi_2)$ to N_e. We can use interpolants (as discussed in Section 3.3) to choose possible values for θ.

Given $e, \varphi_1, \varphi_2, \Omega_{n_1}$, we can show that both rules MUST-POST-SUMMARY and NOTMAY-PRE-USESUMMARY can never be simultaneously enabled since, by construction, it is not possible to both have a must summary $(\hat{\varphi}_1, \hat{\varphi}_2)$ and a not-may summary (ψ_1, ψ_2) for a procedure such that $\psi_1 \supseteq \hat{\varphi}_1$ and $\psi_2 \supseteq \hat{\varphi}_2$.

Recursion. To avoid clutter, the rules for our compositional algorithms (Figures 8, 9 and 10) have been written for non-recursive programs. To handle recursive programs, we need to keep track for each function for which queries are in progress, and also constrain the order in which rules are applied. When a query $\langle \hat{\varphi}_1 \overset{?}{\Longrightarrow}_{\mathcal{P}} \hat{\varphi}_2\rangle$ is made, in addition to checking whether existing not-may or must summaries can be used to answer the queries (using rules MUST-POST-SUMMARY and NOTMAY-PRE-USESUMMARY), we need to also check if this query can be answered by another "in-progress" query $\langle \varphi_1 \overset{?}{\Longrightarrow}_{\mathcal{P}} \varphi_2\rangle$, and start computations for the current query only if no such in-progress query exists. Such a check would guarantee termination of the SMASH algorithm for recursive programs where data types are over a finite domain and no dynamic alloca-

$$\frac{\begin{array}{c}\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad \varphi_1 \cap \Omega_{n_1} \neq \{\} \quad \varphi_2 \cap \Omega_{n_2} = \{\} \\ e = \langle n_1, n_2 \rangle \in E_{\mathcal{P}_i} \text{ is a call to procedure } \mathcal{P}_j \\ (\hat{\varphi}_1, \hat{\varphi}_2) \in \overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_j} \quad \Omega_{n_1} \supseteq \hat{\varphi}_1 \quad \theta \subseteq \hat{\varphi}_2 \quad \varphi_2 \cap \theta \neq \{\}\end{array}}{\Omega_{n_2} := \Omega_{n_2} \cup \theta} [\text{ MUST-POST-USESUMMARY }]$$

$$\frac{\begin{array}{c}\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad \varphi_1 \cap \Omega_{n_1} \neq \{\} \quad \varphi_2 \cap \Omega_{n_2} = \{\} \\ e = \langle n_1, n_2 \rangle \in E_{\mathcal{P}_i} \text{ is a call to procedure } \mathcal{P}_j \\ (\hat{\varphi}_1, \hat{\varphi}_2) \in \overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_j} \quad \varphi_2 \subseteq \hat{\varphi}_2 \quad \theta \subseteq \hat{\varphi}_1 \quad \neg\theta \cap \Omega_{n_1} = \{\}\end{array}}{\Pi_{n_1} := (\Pi_{n_1} \setminus \{\varphi_1\}) \cup \{\varphi_1 \cap \theta, \varphi_1 \cap \neg\theta\} \quad N_e := N_e \cup \{(\varphi_1 \cap \theta, \varphi_2)\}} [\text{ NOTMAY-PRE-USESUMMARY }]$$

$$\frac{\begin{array}{c}\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad \varphi_1 \cap \Omega_{n_1} \neq \{\} \quad \varphi_2 \cap \Omega_{n_2} = \{\} \\ e = \langle n_1, n_2 \rangle \in E_{\mathcal{P}_i} \text{ is a call to procedure } \mathcal{P}_j \quad \psi_1 = \exists V^L_{\mathcal{P}_i}.(\varphi_1 \cap \Omega_{n_1}) \quad \psi_2 = \exists V^L_{\mathcal{P}_i}.\varphi_2\end{array}}{\langle \psi_1 \overset{?}{\Longrightarrow}_{\mathcal{P}_j} \psi_2 \rangle} [\text{ MAYMUST-CALL }]$$

Figure 10. SMASH: Compositional may-must analysis. All rules from compositional may analysis (Figure 8), compositional must analysis (Figure 9), and intraprocedural may-must analysis (Figure 7) are included but not shown, except the rules MAY-CALL and MUST-CALL which are replaced by the new rule MAYMUST-CALL, and except the rules MUST-POST-USESUMMARY and NOTMAY-PRE-USESUMMARY which are modified as shown above.

tion is allowed (for instance, as in boolean programs [3]). However, if data types are unbounded or if dynamic allocation is allowed, checking a query is undecidable and SMASH is not guaranteed to terminate.

Soundness. To establish soundness, we assert the following five invariants over the data structures of the SMASH algorithm:

I1. For every node $n \in N_{\mathcal{P}_i}$, Π_n is a partition of $\Sigma_{\mathcal{P}_i}$.

I2. For every node $n \in N_{\mathcal{P}_i}$ and every state $\sigma \in \Omega_n$, σ is reachable by some program execution at node n starting from initial state of the program.

I3. For every $(\varphi_1, \varphi_2) \in N_e$, and any $\sigma_1 \in \varphi_1$, it is not possible to execute the statement at edge e starting from state σ_1 and reach a state $\sigma_2 \in \varphi_2$.

I4. For every $(\varphi_1, \varphi_2) \in \overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i}$, for any state $\sigma_1 \in \varphi_1$, it is not possible to execute procedure $\mathcal{P}_i$ starting at state σ_1 and reach a state $\sigma_2 \in \varphi_2$ after completing the execution of $\mathcal{P}_i$.

I5. For every $(\varphi_1, \varphi_2) \in \overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_i}$, for any state $\sigma_2 \in \varphi_2$, there exists a state $\sigma_1 \in \varphi_1$ such that executing the procedure $\mathcal{P}_i$ starting at state σ_1 leads to the state σ_2.

The correctness of these invariants is established by induction over each of the rules of the SMASH algorithm. As an example, we show the induction step for the rule NOTMAY-PRE-USESUMMARY from Figure 10. We need to establish that the refinement to Π_{n_1} respects invariant **I1**, and the new edge added to N_e respects the invariant **I3**. Due to the induction hypothesis, we can assume that Π_{n_1} before execution of the rule is a partition. For any $\varphi_1 \in \Pi_{n_1}$, and any value of θ, since Π_{n_1} is a partition, we have that $(\Pi_{n_1} \setminus \{\varphi_1\}) \cup \{\varphi_1 \cap \theta, \varphi_1 \cap \neg\theta\}$ is a partition, hence preserving invariant **I1**. Next, consider the edge $(\varphi_1 \cap \theta, \varphi_2)$ that is added to N_e. Due to the antecedent of the rule, since $(\hat{\varphi}_1, \hat{\varphi}_2) \in \overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_j}$, due to induction hypothesis **I4** we know that no state in $\hat{\varphi}_1$ can reach any state in $\hat{\varphi}_2$ by executing procedure $\mathcal{P}_j$. Since $\theta \subseteq \hat{\varphi}_1$, we have that $\varphi_1 \cap \theta \subseteq \hat{\varphi}_1$. Consequently, we have that no state in $\varphi_1 \cap \theta$ can reach any state in $\varphi_2 \subseteq \hat{\varphi}_2$ by executing $\mathcal{P}_j$, and it follows that the updated N_e satisfies the invariant **I3**. The soundness of all the other rules is proved in a similar way.

5. Evaluation

In this section, we describe our implementation of SMASH and present results of experiments with several Microsoft Windows Vista device drivers.

Statistics	DASH (Figure 7)	SMASH (Figure 10)
Average Not-May Summaries/driver	0	39
Average Must Summaries/driver	0	12
Number of proofs	2176	2228
Number of bugs	64	64
Time-outs	61	9
Time (hours)	117	44

Table 1. SMASH vs. DASH on 69 drivers (342000 LOC) and 85 properties.

5.1 Implementation

Our tool SMASH is a deterministic implementation of the declarative rules of Figure 10, developed in the F# programming language and using the Z3 theorem prover [12] . Recall that every reachability query $\langle \varphi_1 \overset{?}{\Longrightarrow}_{\mathcal{P}_i} \varphi_2 \rangle$ for a procedure $\mathcal{P}_i$ can be answered *exclusively* by either a must summary or a not-may summary. Therefore, the SMASH implementation makes the rules given in Figure 10 deterministic as shown below:

1. If some previously computed must summary $\langle \hat{\varphi}_1 \overset{\mathbf{must}}{\Longrightarrow}_{\mathcal{P}_i} \hat{\varphi}_2 \rangle$ is applicable, return "yes".
2. If some previously computed not-may summary $\langle \hat{\varphi}_1 \overset{\neg\mathbf{may}}{\Longrightarrow}_{\mathcal{P}_i} \hat{\varphi}_2 \rangle$ is applicable, return "no".
3. Otherwise, analyze the procedure $\mathcal{P}_i$ using the rule MAY-MUST-CALL. The result of this analysis could be itself either a "yes" (computation of a must summary for $\mathcal{P}_i$ given by the rule CREATE-MUSTSUMMARY) or a "no" (computation of a not-may summary for $\mathcal{P}_i$ given by the rule CREATE-MAYSUMMARY).

SMASH's intraprocedural analysis implements the DASH algorithm [5] which is an instance of the may-must analysis shown in Figure 7. We have implemented SMASH using the YOGI framework [28] which handles C programs with primitive data types, structs, pointers, function pointers and procedures. Pointer arithmetic is not handled— `*(p+i)` is treated as `*p`, similar to SLAM [4] and BLAST [23]. All predicates used to build may and must program abstractions are propositional logic formulas defined over linear arithmetic and uninterpreted functions stored in Z3's internal representation. With these assumptions, the logic is decidable and therefore every satisfiability/validity query to the theorem prover results in either a "yes" or "no" answer.

Program	Lines	Properties	DASH (Figure 7)			COMPOSITIONAL-MAY-DASH (Figure 8)			COMPOSITIONAL-MUST-DASH (Figure 9)			SMASH (Figure 10)			
	(LOC)		Summaries	Time-outs	Time	Summaries	Time-outs	Time	Summaries	Time-outs	Time	Summaries		Time-outs	Time
					(min)			(min)			(min)	Not-May	Must		(min)
parport	33987	8	0	0	**45**	0	0	**14**	6	0	**44**	0	6	0	**14**
serial1	32385	18	0	11	**420**	64	7	**310**	8	7	**278**	64	7	0	**59**
serial2	31861	17	0	11	**414**	83	4	**204**	11	7	**283**	21	11	0	**39**
sys1	12124	19	0	9	**340**	7	9	**340**	12	3	**144**	2	13	0	**19**
sys2	8593	13	0	5	**206**	0	5	**205**	10	0	**9**	0	9	0	**6**
flpydisk	6747	37	0	0	**59**	317	0	**47**	32	0	**43**	240	32	0	**43**
pscr2	5799	21	0	6	**264**	57	2	**124**	21	0	**27**	47	17	0	**24**
pscr1	5480	25	0	3	**154**	57	2	**133**	16	0	**57**	40	22	0	**26**
modem	3432	15	0	5	**208**	35	0	**55**	1	4	**178**	45	1	0	**15**
1394vdev	2746	15	0	2	**129**	140	2	**126**	8	1	**101**	10	7	0	**26**
1394diag	2745	19	0	3	**158**	23	3	**156**	7	2	**131**	10	9	1	**115**
featured2	2512	21	0	3	**202**	54	2	**158**	32	0	**52**	2	28	0	**20**
featured2a	2465	16	0	3	**167**	30	3	**156**	20	0	**49**	2	24	0	**14**
func_fail	2131	24	0	4	**184**	34	4	**186**	21	2	**120**	5	19	0	**19**
featured1	1880	16	0	3	**149**	36	3	**146**	17	1	**79**	3	19	0	**8**
featured1a	1838	19	0	5	**217**	30	2	**130**	25	3	**145**	2	25	0	**27**

Table 2. Empirical evaluation of SMASH on 16 device drivers.

5.2 Experiments

We evaluated SMASH on 69 Microsoft Windows Vista device drivers and 85 properties. We performed our experiments using a system with a 2.66 GHz Intel Xeon quad core processor with 4GB RAM running Microsoft Windows Server 2003.

Overall comparison. A comparison of SMASH with DASH [5] (this is the non-compositional may-must analysis described in Figure 7) on all 69 device drivers and 85 properties is shown in Table 1. The total number of lines of code analyzed is 342000 lines of code. The total number of checks (we will refer to every verification question involving a driver and a property[1] as a *check*) performed by both analyses is 2301 (the rest correspond to checks where the property is not applicable to the driver). The average number of not-may summaries per driver used by SMASH is 39 and the average number of must summaries per driver equals 12. With a time-out limit of 30 minutes, there are 61 checks where DASH times out while SMASH times out only on 9 checks. The total time taken for the analysis by DASH is 117 hours while SMASH takes only 44 hours, including time-outs. It is worth emphasizing that our non-compositional analysis baseline DASH is already an improvement over SLAM [4] (see [5]).

Detailed comparison on 16 drivers. To understand the effectiveness of SMASH, we drill down into empirical data for 16 drivers (arbitrarily picked from the 69 drivers) in Table 2. Every row of this table shows a driver along with its number of lines of code and the number of properties checked. We compare SMASH against DASH, DASH with compositional may analysis (denoted by COMPOSITIONAL-MAY-DASH), and DASH with compositional must analysis (denoted by COMPOSITIONAL-MUST-DASH). The total time taken by each analysis is reported in minutes. We also report the average number of not-may/must summaries used by each analysis for each driver. It is clear from Table 2 that SMASH outperforms COMPOSITIONAL-MAY-DASH, COMPOSITIONAL-MUST-DASH and DASH on all 16 drivers. This is further elucidated in Figure 11 where each point (x, y) in the graph denotes the fact that there are y checks that take an average time of x minutes. As indicated by the curve for SMASH, a large number of checks take a relatively short amount of time with SMASH. In contrast, with DASH, a large number of checks take a large amount of time. The curves for COMPOSITIONAL-MAY-DASH and COMPOSITIONAL-MUST-DASH are better than DASH, but are outperformed by

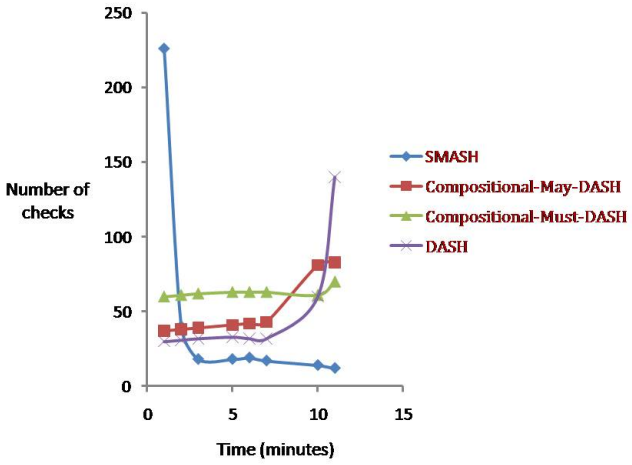

Figure 11. Comparison of SMASH with COMPOSITIONAL-MAY-DASH, COMPOSITIONAL-MUST-DASH and DASH on 303 checks.

SMASH. This data indicates that in addition to gains obtained by COMPOSITIONAL-MAY-DASH and COMPOSITIONAL-MUST-DASH analyses, extra gains are obtained in SMASH due to the interplay between may analysis and must analysis.

There are a number of checks where DASH, COMPOSITIONAL-MAY-DASH and COMPOSITIONAL-MUST-DASH time out while SMASH does not. These correspond to patterns like those outlined in Figures 3 and 4, which illustrate how the intricate interplay between not-may summaries and must summaries can prevent SMASH from getting "stuck". However, SMASH does timeout on 1 check for the driver `1394diag` and the reason for this (as well as all the 9 time-outs in Table 1) is that it is unable to discover the right invariant to prove the property.

Table 2 also shows that the total number of summaries used by SMASH is less than the total number of summaries used by COMPOSITIONAL-MAY-DASH and COMPOSITIONAL-MUST-DASH put together. Since the number of summaries is related to the number of queries (every distinct query that cannot be answered using an existing summary results in a new summary), this data points to reduced number of queries in SMASH, indicating that a combination of may and must analyses can drastically reduce the number of queries in the analysis.

[1] Properties are typestate properties for device drivers as described in `http://msdn.microsoft.com/en-us/library/aa469136.aspx`.

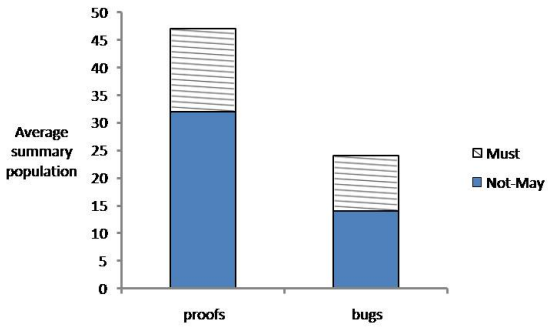

Figure 12. Average summary population for SMASH.

Interplay between not-may and must summaries. Finally, to quantify the interplay between not-may and must summaries in SMASH, we examine the average summary population for top-level queries/checks (those that result in proofs or bugs) for all the 16 drivers in Table 2. This data is presented in Figure 12— the first column in this figure represents the average number of not-may summaries and must summaries used to answer queries that resulted in a "no" answer (a proof/not-may answer), while the second column represents the average number of not-may summaries and must summaries used to answer queries that resulted in a "yes" (a bug/must answer). On an average, to generate a proof, 68% of summaries used were not-may summaries and 32% of the summaries used were must summaries. On an average, to identify a bug, 64% of summaries used were must summaries and 36% of the summaries used were not-may summaries. Indeed, it is this fine-grained coupling and alternation of not-may and must summary applications that allows SMASH to easily navigate through code fragments that are typically hard to analyze using COMPOSITIONAL-MAY-DASH, COMPOSITIONAL-MUST-DASH and DASH.

The experiments reported here were performed in an experimental setup similar to the one used in [5]: (1) environment models/code are used to simulate the effects of the operating system visible to the device drivers considered, and (2) "concrete" executions of the drivers (and of the environment code) are simulated by an interpreter. Note that assumption (1) is an inherent limitation to all may static program analysis: the impact of the external environment must be modeled abstractly somehow. In contrast, testing-based approaches [16] can simply run the actual environment code (assuming it is available) as a blackbox, although this typically makes the analysis incomplete, i.e., the analysis may miss bugs. Because of assumption (2), the level of precision when computing preconditions Pre and postconditions Post in the rules of Figure 7 is always the same, and one of the two rules MUST-POST or NOTMAY-PRE is always guaranteed to be applicable.

6. Related Work

As mentioned earlier, the SMASH algorithm generalizes and extends several existing algorithms.[2] SLAM [4] performs a compositional may analysis using predicate abstraction and partition refinement, but does not perform a must analysis. SMART [14] performs a compositional must analysis, extending the non-compositional must analysis of DART [16], but does not perform a may analysis. SYNERGY [19] combines SLAM and DART for intraprocedural analysis only. DASH [5] performs an interprocedural may-must analysis, extending the intraprocedural SYNERGY algorithm [19], but is non-compositional, i.e., it does not memoize (cache) intermediate results in the form of reusable summaries. Also, the formalization of the DASH algorithm in [5] does not account for imprecision in symbolic execution and constraint solving while computing preconditions Pre or postconditions Post.

To the best of our knowledge, SMASH is the first 3-valued *compositional may-must analysis algorithm*. Its key novel feature is its fined-grained coupling between may and must summaries, which allows using either type of summaries in a flexible and demand-driven manner. As shown in the experiments of the previous section, this alternation is the key feature that allows SMASH to outperform previous algorithms.

Several other algorithms and tools combine static and dynamic program analyses for property checking and test generation, e.g., [27, 31, 10, 6]. Most of these looser combinations perform a static analysis before a dynamic analysis, while some [6] allow for some feedback to flow between both. But none support fine-grained alternation between the may/static and must/dynamic parts, and most are not compositional.

Compositional may program analysis has been amply discussed in the literature [24], and has recently been extended to the must case [14, 1]. Our work combines both compositional may and compositional must analyses in a tight unified framework and shows how to leverage their complementarity.

Three-valued may-must program analysis using predicate abstraction has been proposed before [15, 21]. However, this earlier work used the same abstract states (sets of predicates) to define both may and must abstractions, and was not compositional. In contrast, SMASH is compositional and uses two different abstract domains for its may and must analyses (as in SYNERGY and DASH): may abstract states are defined using predicate abstraction and are iteratively refined by adding new predicates to split abstract regions; while must abstract states are defined using symbolic execution along program paths executed with concrete tests and are incrementally computed with more tests but without refining must abstract states.

Program analysis using three-valued shape graphs as abstract states has also been proposed for shape analysis [29]. There, abstract states are richer three-valued structures, while transitions between those states are traditional two-valued may transitions.

In the context of verification-condition-generation-style program verification, [13] discusses how to over-approximate and under-approximate recursive logic formulas representing whole programs and generated by a static program analysis for a fixed set of predicates. In contrast, our approach builds up a logical program representation *incrementally*, by refining simultaneously dual may over-approximate and must under-approximate compositional program abstractions defined by varying sets of predicates. Moreover, unlike [13], we do not suffer from false positives since our must analysis is grounded in concrete executions through program testing.

In this work, we focus on checking *safety* properties, which can be reduced to evaluating reachability queries, and arguably represent most properties one wants to check in practice. It is well known that may-must abstractions can also be used to check more expressive properties, such as liveness and termination properties, as well as properties represented by μ-calculus formulas with arbitrary alternation of universal and existential path quantifiers [15]. However, in order to check liveness properties with finite-state abstractions, more elaborate abstraction techniques, such as the generation of fairness constraints, are in general necessary [30, 25].

We assume in this paper that all concrete program executions terminate. In practice, this assumption can easily be checked for specific executions at run-time using timers. In other words, we assume and check for termination using the must part of our program analysis, but we do not attempt to prove that all program executions always terminate using the may part [9], nor do we try to find some

[2] The name SMASH is a combination of the names SMART and DASH.

non-terminating execution [20]. It would be interesting to combine techniques for proving termination [9] and non-termination [20] with compositional may-must program analysis.

SMASH fits in the category of *property-guided* algorithms and tools: each experiment reported in the previous section is aimed at either proving or disproving that a program (device driver) satisfies a specific property (assertion) of interest. In contrast, the approach taken in [16, 8, 18] is aimed at exercising as many program paths as possible while checking many properties simultaneously along each of those paths [17].

7. Conclusions

We have presented a unified framework for compositional may-must program analysis and a specific algorithm, SMASH, instantiating this framework. We have implemented SMASH using predicate abstraction for the may part and using dynamic test generation for the must part. Results of experiments with 69 Microsoft Windows Vista device drivers show that SMASH can significantly outperform may-only, must-only and non-compositional may-must algorithms.

The key technical novelty of SMASH is the tight integration of may and must analyses using interchangeable not-may/must summaries. Although the general idea of combining compositional may and must analyses is natural, the outcome of the experimental evaluation was surprising as SMASH performed much better than adding compositionality separately to may and must analyses. This led us to uncover the previously-unnoticed power of *alternation* of may and must summaries in the context of compositional program analysis. We have also been able to quantify the amount of interplay that happens between may and must summaries in our data set.

SMASH is implemented in YOGI [28] which is one of the tools in the Static Driver Verifier (SDV) toolkit for analyzing Windows device drivers and will eventually be shipped with Windows.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS '08: Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381, 2008.

[2] T. Ball, O. Kupferman, and G. Yorsh. Abstraction for falsification. In *CAV '05: Computer-Aided Verification*, 2005.

[3] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN '00: International SPIN Workshop*, pages 113–130, 2000.

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

[5] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA'08: International Symposium on Software Testing and Analysis*, pages 3–14, 2008.

[6] D. Beyer, T. A. Henzinger, and G. Theoduloz. Program analysis with dynamic precision adjustment. In *ASE '08: Automated Software Engineering*, 2008.

[7] 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, 2000.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS '06: Computer and Communications Security Conference*, 2006.

[9] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI '06: Programming Language Design and Implementation*, 2006.

[10] C. Csallner and Y. Smaragdakis. Check'n Crash: Combining static checking and testing. In *ICSE '05: International Conference on Software Engineering*, 2005.

[11] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI'02: Programming Language Design and Implementation*, pages 57–69, 2002.

[12] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *TACAS '08: Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[13] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI '08: Programming Language Design and Implementation*, pages 270–280, 2008.

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

[15] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR '01: International Conference on Concurrency Theory*, pages 426–440, 2001.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI '05: Programming Language Design and Implementation*, pages 213–223, 2005.

[17] P. Godefroid, M.Y. Levin, and D. Molnar. Active property checking. In *EMSOFT '08: Annual Conference on Embedded Software*, pages 207–216, 2008.

[18] P. Godefroid, M.Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS '08: Network and Distributed Systems Security*, pages 151–166, 2008.

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

[20] A. K. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R-G. Xu. Proving non-termination. In *POPL '08: Principles of Programming Languages*, pages 147–158, 2008.

[21] A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A software model checker for verification and refutation. In *CAV '06: Computer-Aided Verification*, pages 170–174, 2006.

[22] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific static analyses. In *PLDI'02: Programming Language Design and Implementation*, pages 69–82, 2002.

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

[24] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *FSE '95: Foundations of Software Engineering*, pages 104–115, 1995.

[25] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1), 2000.

[26] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV '03: Computer-Aid Verification*, pages 1–13, 2003.

[27] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL '02: Principles of Programming Languages*, pages 128–139, 2002.

[28] A. V. Nori, S. K Rajamani, S. Tetali, and A. V. Thakur. The Yogi Project: Software property checking via static analysis and testing. In *TACAS '09: Tools and Algorithms for the Construction and Analysis of Systems*, pages 178–181, 2009.

[29] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL '99: Principles of Programming Languages*, pages 105–118, 1999.

[30] T. Uribe. *Abstraction-based Deductive-Algorithmic Verification of Reactive Systems.* PhD thesis, Stanford University, 1999.

[31] W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA '04: International Symposium on Software Testing and Analysis*, 2004.