SLAM2: Static Driver Verification with Under 4% False Alarms

Thomas Ball
Microsoft Research
Redmond, USA

Ella Bounimova
Microsoft Research
Redmond, USA

Rahul Kumar
Microsoft
Redmond, USA

Vladimir Levin
Microsoft
Redmond, USA

***Abstract*—In theory, counterexample-guided abstraction refinement (**CEGAR**) uses spurious counterexamples to refine overapproximations so as to eliminate provably false alarms. In practice, CEGAR can report false alarms because: (1) the underlying problem CEGAR is trying to solve is undecidable; (2) approximations introduced for optimization purposes may cause CEGAR to be unable to eliminate a false alarm; (3) CEGAR has no termination guarantee - if it runs out of time or memory then the last counterexample generated is provably a false alarm.**

We report on advances in the SLAM analysis engine, which implements CEGAR for C programs using predicate abstraction, that greatly reduce the false alarm rate. SLAM is used by the Static Driver Verifier (SDV) tool. Compared to the first version of SLAM (SLAM1, shipped in SDV 1.6), the improved version (SLAM2, shipped in SDV 2.0) reduces the percentage of false alarms from 25.7% to under 4% for the WDM class of device drivers. For the KMDF class of device drivers, SLAM2 has under 0.05% false alarms. The variety and the volume of our experiments of SDV with SLAM2, significantly exceed those performed for other CEGAR-based model checkers.

These results made it possible for SDV 2.0 to be applied as an automatic and required quality gate for Windows 7 device drivers.

I. Introduction

A decade ago, the SLAM project [BR02b] introduced the concept of counterexample-guided abstraction refinement (CEGAR) for the analysis of temporal safety properties of C programs. This work resulted in the Static Driver Verifier (SDV) tool that Microsoft applies internally to its device drivers and ships with the Windows Driver Development Kit (WDK) for use by third-party device driver writers [BBC+06].

As shown in Figure 1, the essential points of the CEGAR process, as implemented by SLAM, are: (1) the automated creation of a Boolean program *abstraction* of an instrumented C program that contains information relevant to the property under consideration; (2) *model checking* of the Boolean program to determine the absence or presence of errors; (3) the *validation* of a counterexample *trace* to determine whether or not it is a feasible trace of the C program. The last step can either produce a validated counterexample trace or a proof that the trace is invalid (a provably false alarm), in which case information is added to the abstraction to rule out the false alarm.

The CEGAR process has three distinct attributes: first, it may terminate with either a proof of correctness ("verified") or a validated counterexample trace; second, if CEGAR proves a counterexample trace is invalid then, in theory, it can rule out at least this trace from the abstraction (the so-called *progress property*); third, even if CEGAR always makes progress it still has no guarantee of terminating [BPR02].

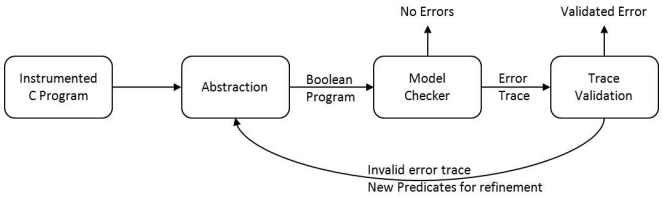

Fig. 1. The SLAM realization of the CEGAR loop.

Theoretically, the lack of a termination guarantee appears to be the death knell for CEGAR: most program analyses typically have termination guarantees despite having the problem of false alarms. However, we can set a time limit on a CEGAR run. If the run is aborted, we have the result that the last counterexample trace considered by CEGAR was invalid (provably a false alarm). So, CEGAR with a time limit has a three-valued outcome: (1) verified; (2) validated error trace; (3) not-useful result (NUR) due to lack of progress or timeout/spaceout. In the second case, the result still could be a false alarm due to bugs in the environment model, temporal safety property, or the SLAM engine itself. In the results reported in the abstract and here in the introduction, we count such cases as well as NURs as "false alarms".

In order to improve the chances for CEGAR to terminate with useful results and fewer false alarms, we explored four main ideas in SLAM2, which was derived from SLAM1.

First, we increase the precision of the predicate transformer over statement sequences. SLAM1 abstracts each C program statement (such as an assignment or **assume** statement representing a conditional branch) to a corresponding Boolean program statement. Thus, if the C program contains the statement sequence $(S_1; S_2)$ then the Boolean program abstraction computed by SLAM1 contains the statement sequence $({S_1}^{\#}; {S_2}^{\#})$, where $S^{\#}$ is the abstraction of statement S. We call this approach *fine-grained* abstraction. Our contribution here is to show how to construct the Cartesian/Boolean program abstraction [BPR01] for sequences of assignments and assume statements, so that the statement sequence $(S_1; S_2)$ abstracts to $(S_1; S_2)^{\#}$. We call this approach *coarse-grained* abstraction, which SLAM2 implements.

Second, we use diverse strategies for exploring counterexample traces. SLAM1 uses a "depth-first" strategy: it symbolically executes a counterexample trace in the C program forward from the initial state. As soon as it finds a trace prefix that is inconsistent, it generates a set of refinement predicates and a refined Boolean program abstraction. The SLAM1 symbolic execution step is complicated because of its use of symbolic (Skolem) constants, which must be tracked and eliminated in order to later generate properly scoped predicates [BR02a].

In contrast, SLAM2 uses both forward and backward symbolic execution. Forward symbolic execution is a simple interpreter that maintains a symbolic store. Backward symbolic execution is based on preconditions, decomposed and cached per program point in order to make predicate generation very simple. The combination of forward and backwards symbolic execution allows SLAM2 to detect inconsistencies near the beginning of a counterexample trace as well as near the end or in the middle, giving it more flexibility over SLAM1.

The third major difference is in how the two engines react to the lack of progress, which can occur because SLAM computes approximations to the best Boolean abstraction in order to speed the search for both proofs and counterexamples. Upon finding lack of progress (identified when none of the predicates generated in the current iteration of CEGAR is new), SLAM1 refines the Boolean program transition relation [BCDR04]. We call this the CONSTRAIN module of SLAM, which is common to both SLAM1 and SLAM2. In contrast, SLAM2 detects multiple inconsistencies in the same counterexample trace when a lack of progress stops it; it interleaves the discovery of new predicates with application of the CONSTRAIN module so that it is less likely to get stuck.

Fourth, SLAM2 uses information computed during forward symbolic execution to optimize backward symbolic execution in several ways. In particular, the value of pointers computed by the forward execution is critical to the optimization of the precondition calculation for assignment statements and procedure calls.

In addition to these four main ideas, SLAM2 has a completely re-implemented and more efficient pointer analysis. To optimize predicate evaluation, SLAM2 uses the Z3 state-of-the-art SMT solver [MB08] with two major improvements in the interface between SLAM and Z3: an efficient encoding of the predicates given to Z3 and a new set of axioms that express the SLAM memory model, in particular, relations between pointers and locations [BBdML10].

As the saying goes, "the proof is in the pudding": compared to SLAM1, SLAM2 reduces the percentage of false alarms from 25.7% to under 4% for the WDM class of device drivers. For the KMDF class of device drivers, SLAM2 has under 0.05% false alarms.[1] These figures come from 5727 unique checks using both SLAM1 and SLAM2 on 69 device drivers from the WDK against 83 temporal safety properties.

A common question about verification tools is "who verifies the verifier?". The typical answer is that one uses lots of benchmarks and testing, as well as cross comparison to other tools. In the development of SLAM2, we found numerous deficiencies in SLAM1, including its overconstraining of the abstract transition relation, which leads to "false verification", a real but little acknowledged problem with verification tools.

So, we also compared SLAM2 to the YOGI analysis engine [NRTT09] on the same benchmarks. For WDM, SLAM2 provides 7% fewer NURs, fewer false defects (2 versus 18), while finding 18 true defects that YOGI misses (YOGI finds 2 true defects that SLAM2 misses), and is two times faster than YOGI. For KMDF, SLAM2 produces 58 times fewer NURs (2 versus 117), and is 8 times faster than YOGI.

SLAM2 moves closer to the CEGAR promise to "abstract-and-refine" until it produces a proof of correctness or a validated trace. The false alarm rate of SLAM2 is so low that SLAM2 empowers a truly push-button software model checking experience for users of the SDV tool, which resulted in the technology being required as quality gate for shipping of Microsoft-produced Windows 7 device drivers.

The rest of this paper is organized as follows: Section II presents the coarse-grained abstraction; Section III describes the forward and backwards symbolic interpreters; Section IV describes how SLAM2 uses these interpreters to optimize the CEGAR loop; Section V presents the treatment of preconditions for assignments and procedure calls in the presence of pointers; Section VI presents experiments results; Section VII reviews related work, and Section VIII concludes the paper.

II. COARSE-GRAINED BOOLEAN ABSTRACTION

Given a C program P, a set of Boolean expressions E, SLAM's predicate abstraction step produces the Boolean program abstraction $BP(P, E)$ containing variables $V = \{b_1, b_2, \ldots, b_n\}$. Each variable b_i in V corresponds to the Boolean expression (predicate) ϕ_i in E. Boolean programs contain all the control-flow constructs of C, including procedures and procedure calls. We will focus here on the abstraction of a procedure with no procedure calls, as the handling of procedure calls and returns remain unchanged compared to SLAM1 [BMR05].

Each procedure of a C program is represented by a control-flow graph with basic blocks, where each basic block is a sequence of assignments, skips, and **assume** statements. The **assume** statements are used to model the semantics of **if-then-else** statements as well as assumptions about data (non-nullness of pointers).

SLAM2 generalizes the abstraction step compared to SLAM1 by abstracting sequences of statements as opposed to single statements:

$$S \rightarrow S_1; S_2 \mid \textbf{skip} \mid x := e \mid *x := e \mid \textbf{assume}(e)$$

The main advantage of coarse-grained abstraction compared to fine-grained is increased precision [CC77].

[1] The Windows Driver Model (WDM) is a widely-used kernel-level API that provides access to low-level kernel routines as well as routines specific to driver's operation and life-cycle. The Kernel-mode Driver Framework (KMDF) is a new kernel-level API which provides higher-level abstractions of common driver actions.

S	$pre(S, Q)$	$wp(S, Q)$
skip	Q	Q
$x := e$	$Q[e/x]$	$Q[e/x]$
$*x := e$	$(x = \&y_1 \wedge Q[e/y_1]) \vee \ldots \vee (x = \&y_k \wedge Q[e/y_k])$	same as $pre(S, Q)$
assume(e)	$e \wedge Q$	$e \implies Q$
$S_1; S_2$	$pre(S_1, pre(S_2, Q))$	$wp(S_1, wp(S_2, Q))$

Fig. 2. Predicate transformers pre and wp.

A. Transformation

We use the standard precondition (pre) and weakest precondition (wp) predicate transformers to assign meaning to C programs as well as to perform the abstraction to Boolean programs. Figure 2 shows the predicate transformers for the statements S under consideration. Recall that $wp(S, Q) = \neg pre(S, \neg Q)$.

We use a source-to-source transformation on the C program to simplify the abstraction process. Any statement sequence S is equivalent to **assume**$(pre(S, true)); sub(S)$, where the function $sub(S)$ is defined to be the maximal subsequence of S containing only assignment statements of S (and is defined to be the **skip** statement in the case that S contains no assignment statements).

Lemma 1 (*Correctness of transformation*). For all statement sequences S and predicates Q:

$$wp(S, Q) \iff wp(\textbf{assume}(pre(S, true)); sub(S), Q)$$

Proof. By induction on length of statement sequence S, show that

$$wp(S, Q) \iff (pre(S, true) \implies wp(sub(S), Q))$$

[The proof is straightforward but omitted due to lack of space]

B. Abstraction

A *cube* over V is a conjunction $c_{i_1} \wedge c_{i_2} \wedge \ldots \wedge c_{i_k}$, where each $c_{i_j} \in \{b_{i_j}, \neg b_{i_j}\}$ for some $b_{i_j} \in V$. For a variable $b_i \in V$, let $\mathcal{E}(b_i)$ denote the corresponding predicate φ_i, and let $\mathcal{E}(\neg b_i)$ denote the predicate $\neg\varphi_i$. Extend $\mathcal{E}$ to cubes and disjunctions of cubes in the natural way.

For any predicate φ and set of Boolean variables V, let $\mathcal{F}_V(\varphi)$ denote the largest disjunction of cubes c over V such that $\mathcal{E}(c)$ implies φ. The predicate $\mathcal{E}(\mathcal{F}_V(\varphi))$ represents the weakest predicate over $\mathcal{E}(V)$ that implies φ. The corresponding weakening of a predicate is also defined similarly. Let $\mathcal{G}_V(\varphi)$ be $\neg\mathcal{F}_V(\neg\varphi)$. The predicate $\mathcal{E}(\mathcal{G}_V(\varphi))$ represents the strongest predicate over $\mathcal{E}(V)$ that is implied by φ.

Following Lemma 1 and the definition of Cartesian/Boolean abstraction [BPR01], Figure 3 shows the translation of a statement S to a *guarded parallel assignment* in the Boolean program. Here the $*$ value represents a value non-deterministically chosen from $\{true, false\}$. The computation of the predicate abstraction of a formula ϕ, as represented by $\mathcal{F}_V(\phi)$, typically relies on an automated theorem prover [GS97]. SLAM1 and SLAM2 both rely on a specialized algorithms for predicate abstraction [LBC05].

assume$(\mathcal{G}_V(pre(S, true)))$;

$b_1 :=$ **if** $(\mathcal{F}_V(wp(sub(S), \varphi_1))$ **then** $true$ **else**
if $(\mathcal{F}_V(wp(sub(S), \neg\varphi_1)))$ **then** $false$ **else** $*$,

$\ldots$

$b_n :=$ **if** $(\mathcal{F}_V(wp(sub(S), \varphi_n))$ **then** $true$ **else**
if $(\mathcal{F}_V(wp(sub(S), \neg\varphi_n)))$ **then** $false$ **else** $*$;

Fig. 3. Cartesian/Boolean abstraction of statement sequence S.

III. Counterexample Trace Validation

In this section, we explain the two symbolic interpreters that SLAM2 uses to perform counterexample trace validation on C programs and predicate discovery. The first is a forward interpreter and the second a backwards interpreter (SLAM1 only performs forward symbolic execution). The next section will discuss more about how the two interpreters are used together.

The language of compound statements introduced in the previous section for the abstraction of basic blocks also serves as the basis for our discussion of symbolic execution of an *execution trace*. An execution trace is simply a sequence of basic blocks through the control-flow graph, whose code can be modeled by a sequence of assignment and **assume** statements (one very long basic block). For the rest of this section, let $S_1 \ldots S_n$ represent the sequence of statements in the execution trace under consideration.

A. Forward Symbolic Execution

Forward Symbolic Execution (FSE) processes the *entire* trace $S_1 \ldots S_n$ with two goals: (1) to find an invalid execution trace prefix of the form $S_1 \ldots S_i$; (2) to populate a "trace database" that maps each statement S_j to the store computed by FSE just before execution of S_j. The main use of the trace database is to resolve pointer-aliasing questions in a trace-sensitive manner, as detailed in Section V.

Operationally, forward symbolic execution is an interpreter that computes the strongest post-condition ($sp(P, S)$) of a statement sequence S with respect to the initial predicate $P = true$. Recall that

$$\begin{array}{lcl} sp(P, \textbf{skip}) & = & P \\ sp(P, \textbf{assume}(e)) & = & P \wedge e \\ sp(P, x := e) & = & \exists\theta_x.P[x/\theta_x] \wedge (x = e[x/\theta_x]) \\ sp(P, S_1; S_2) & = & sp(sp(P, S_1), S_2) \end{array}$$

C-like Program	Precondition Vectors

1: void main(){
2: int x, y, a;
3: x := y;
4: x := x+1;
5: if(a>0)
6: a := a+1;
7: if(x = y+2){
8: SLIC_ERROR:0;
9: }
10: }
	

(a)	(b)

	0	1	2
3-4	$true$	$y+1=y+2$	$\neg(a>0)$
5	$true$	$x=y+2$	$\neg(a>0)$
7	$true$	$x=y+2$	
8	$true$		

Fig. 4. Backwards Symbolic Execution

(for brevity, we omit the rule for $*x := e$).

FSE maintains a store mapping locations to values and processes the statements $S_1 \ldots S_n$ in order from S_1 to S_n. Symbolic evaluation of an assignment ($x := e$ or $*x := e$) involves: (1) evaluation of the RHS expression e in the context of the current store to get a value v; (2) evaluation of the LHS expression in the context of the current store to get a location l; (3) mapping location l to value v in the store (possibly overwriting the previous mapping for location l). During symbolic execution, if a location l (such as the address of variable x) doesn't have a mapping in the store then a fresh symbolic value θ_l for the value of l is created and l is mapped to θ_l in the store.

Execution of a statement $S_i =$ **assume**(e_i) first evaluates the Boolean expression e_i in the current store, which results in an expression ϕ_i solely over constants of the programming language (such as 1, 42, ...) and symbolic constants (such as θ_l). FSE maintains a trace condition ϕ (initially $true$), which is the conjunction of the ϕ_i. A call to the theorem prover Z3 [MB08] determinines the satisfiability of the formula $\exists\theta.\phi \wedge e_i$. If the formula is satisfiable, then there is an assignment of values to the symbolic constants θ (the primary inputs to the execution trace) that witness the validity of the execution trace. If it is unsatisfiable then the trace prefix $S_1 \ldots S_i$ is inconsistent/invalid.

B. Backwards Symbolic Execution

Operationally, backwards symbolic execution (BSE) computes $pre(S_1 \ldots S_k, true)$, $k \leq n$, but decomposes and caches the representation of each application of pre in order to enable predicate generation if the counterexample is determined to be invalid. The benefits of symbolic execution with pre are: (1) there is no need to introduce symbolic constants; (2) assignments to variables that don't appear in the postcondition Q have no effect. An issue with the use of pre is a blow-up in the size of the precondition formula due to pointer aliasing (see the rule for $*x := e$ in Figure 2), which we will return to later.

The decomposition of pre is based on the simple observation that $pre($**assume**$(e), Q) = (e \wedge Q)$. If Q is a conjunction $(q_0 \wedge \ \ldots \ \wedge q_r)$, represented implicitly by the vector $< q_0, \ \ldots, \ q_r >$, then we represent $(e \ \wedge \ Q)$ by $< q_0, \ \ldots, \ q_r, \ e >$, which preserves the positions of the q_i in the vector.

BSE starts with the one element vector $Q = \ < true >$. Processing of an **assume** statement lengthens the vector by one element, as described above. For an assignment statement, the pre computation for the assignment is applied point-wise to the input vector, resulting in a new vector of the same length.

We can visualize the computation of pre as creating an upper-left-triangular matrix of row vectors, where the first column contains $true$ everywhere and each subsequent column represents the history of a subformula introduced by an **assume** statement. The last row ($k+1$) of the matrix represents the starting point where $Q_k =< true >$. The i^{th} row of the matrix ($1 \leq i < k$) represents $Q_i = pre(S_i \ldots S_k, true)$.

For each new precondition vector Q_i computed, Z3 is called to query if the conjunction of formulas in the vector is satisfiable. If it is unsatisfiable then the trace $S_i \ldots S_k$ is invalid and the predicate discovery algorithm starts, as described in the next subsection. Otherwise, BSE proceeds to consider statement S_{i-1} in the trace. If BSE determines that Q_1 is satisfiable then the execution trace is valid.

Figure 4 illustrates BSE on a simple C program (a). Consider the false counterexample trace `2-3-4-5-7-8`. Figure 4(b) shows the vector-based computation of pre on this trace, with the corresponding trace step numbers in the left-most column (only the steps where the preconditions change are shown).

Columns 0-2 in the table show the precondition computation for each step of the trace, going backwards from the error step 7. For example, at step 6 a new vector element $x = y + 2$ is added, which corresponds to the **then** branch of the conditional. At steps 3 and 4, which correspond to the sequence of assignments $y := x; x := x + 1$, the precondition in column 1 is computed as $pre(y := x; x := x+1, x = y+2) = (y+1 = y+2)$, whereas the precondition in column 2 is not affected.[2]

C. Predicate Discovery

Given an invalid execution trace $S_i \ldots S_k$, the goal of predicate discovery is very simple: find a set of predicates

[2]Note that the two assignment statements occupy the same basic block, so are treated together, just as they are during the abstraction step. This reduces the number of predicates generated.

E such that the abstract version of pre induced by E (pre_E) can prove $S_i \ldots S_k$ is an invalid execution trace.

More formally, let $\alpha_E(\phi)$ be the weakest formula ϕ' (in the implication ordering) such that ϕ' is a Boolean combination of the predicates in E and ϕ' implies ϕ. Then for a basic block S, $pre_E(S, Q) = \alpha_E(pre(S, Q))$ and for a sequence of two basic blocks S_1 and S_2, $pre_E(S_1; S_2, Q) = pre_E(S_1, pre_E(S_2, Q))$. Suppose that $pre(S_i \ldots S_k, true) = false$, where the S_x are basic blocks, then we wish to find a *sufficient* set of predicates E such that $pre_E(S_i \ldots S_k, true) = false$.

Once BSE has discovered that a precondition vector Q_i is unsatisfiable, it is clear that the set of predicates in the precondition matrix $M_{i+1} =< Q_{i+1} \ \ldots \ Q_k >$ are sufficient. Of course, we can do much better: the underlying theorem prover can provide us an unsatisfiable core of Q_i, a small subset of the elements of Q_i whose conjunction is unsatisfiable. This subset identifies a set of "inconsistent" columns in M_{i+1}. Again, it is clear that the set of predicates from this set of columns are sufficient. In our example at line 3, the formula

$$\exists y. \exists a. true \wedge (y + 1 = y + 2) \wedge \neg(a > 0)$$

is unsatisfiable. An unsatisfiable core is $\{(y+1 = y+2)\}$. So, a sufficient set E includes predicates from the second column: $\{x = y + 2\}$.

IV. Optimizing the CEGAR Loop: Multiple Inconsistencies

Optimizations of the CEGAR loop are based on analysis of the cases when SDV fails on Windows device drivers with "not-useful results" (NURs, in SDV terminology). In theory, for a CEGAR run, the set of predicates strictly increases as the iterations of CEGAR increase. Let E_i be the set of predicates discovered by iteration i of CEGAR. In practice, both SLAM1 and SLAM2 may discover predicates E_j such that $E_j \subseteq \bigcup_{0 \leq i < j} E_i$. This lack of progress condition can arise due to approximations introduced in the abstraction step, which can result in the same counterexample trace being produced in consecutive iterations.

Upon finding lack of progress, SLAM1 employs a tool called CONSTRAIN to refine the Boolean program abstraction computed for the current set of predicates [BCDR04]. Our experiments indicated that CONSTRAIN was a bottleneck in SLAM1, so we experimented with techniques in SLAM2 to reduce the need to use CONSTRAIN.

The optimized CEGAR loop makes use of both FSE and BSE, as well as the CONSTRAIN module. Given a counterexample trace $S_1 \ldots S_n$, SLAM2 first applies FSE. If FSE finds an invalid trace prefix $S_1 \ldots S_i$ then BSE is applied to the trace $S_1 \ldots S_i$ to discover new predicates.

The approach outlined above is similar to SLAM1: predicates are discovered based on invalid trace *prefixes*. However, an invalid trace can have several invalid subtraces. So, SLAM2 also uses BSE in two new ways to discover more invalid subtraces. First, if there is lack of progress on invalid trace prefix $S_1 \ldots S_i$, SLAM2 will apply BSE to the entire trace $S_1 \ldots S_n$ to try to find an invalid trace suffix $S_k \ldots S_n$. Second, if there is lack of progress on invalid trace suffix $S_k \ldots S_n$, SLAM2 will perform a *partial reset* of the pre computation and continue BSE, as follows. Suppose that the set of inconsistent columns of the precondition matrix after processing $S_k \ldots S_n$ are $k_1, k_2, \ldots, k_m$. The partial reset removes these columns from the precondition matrix and resumes BSE at statement S_{k-1}. The partial reset can be done multiple times to find multiple invalid traces.[3]

The above approach is interleaved with the application of the CONSTRAIN module, which is applied just once when a lack of progress is first identified. SLAM1 does not attempt to find multiple invalid subtraces. Upon lack of progress, it attempts to resolve the issue using CONSTRAIN. If lack of progress continues, SLAM1 terminates with a "GiveUp" result, whereas SLAM2 will continue to analyze the trace to find new predicates. If SLAM2 finishes exploring $S_1 \ldots S_n$ with no new predicates, it too will terminate with a "GiveUp" result.

V. Procedure Calls and Pointers

A key aspect of the SLAM approach to CEGAR is that the Boolean program abstraction contains procedures and procedure calls. Thus, Boolean variables introduced by predicate discovery can be locally scoped to a procedure, which reduces the cost of model checking.

SLAM2 remains unchanged with respect to SLAM1 regarding Boolean program abstractions with procedures. BSE performs precondition evaluation at procedure return and procedure call steps by converting the precondition from the scope of the caller into the scope of the callee (for returns) and back (for calls). This is done by using relations between actual and formal parameters of the call/return, and between the return value of the procedure call (if any) and the return variable of the callee.

As discussed before, the precondition computation applied during BSE has the potential to blow up in size because of pointers. But, in fact, SLAM2 eliminates this possibility by making the pre computation trace-sensitive for BSE, using the pointer aliasing information computed by FSE. Consider a statement $S_i : *x := e$ in the trace. Recall that $pre(*x := e, Q)$ is

$$(x = \&y_1 \wedge Q[e/y_1]) \vee \ldots \vee (x = \&y_k \wedge Q[e/y_k])$$

To reduce the size of this formula, BSE looks up the location pointed to by x in the store computed by FSE on entry to statement S_i. Suppose that in this store x maps to $\&y_j$. Then the above equation reduces to $Q[e/y_j]$.

VI. Experimental Results

We now present a comparison of SLAM2, SLAM1 and YOGI by running SDV on two large test suites developed and maintained by Microsoft quality assurance teams for testing SDV. We first describe our evaluation platform and

[3]One could also perform a full reset of the precondition matrix to the initial vector $< true >$ - we did not experiment with this approach.

Metric	SDV 1.6 (SLAM1)	SDV 2.0 (SLAM1)	SDV 2.0 (SLAM2)
Drivers	69	69	69
Rules	68	83	83
Total checks	4692	5727	5727
LightweightPass results	-	2477	2477
Pass results	-	2563	2551
NUR results	6% (285/4692)	2.1% (123/5727)	3.3% (187/5727)
Defects reported	157	564	512
GiveUp results only	-	0.52% (30/5727)	0.3% (16/5727)
False defects	19.7% (31/157)	9.04% (51/564)	0.4% (2/512)
Time for identical pass	-	39922	65800
Time for identical defect	-	4440	2669

TABLE I
COMPARISON OF SLAM1 AND SLAM2 FOR WDM DRIVER CHECKS.

criteria. At Microsoft, SDV is used for verification of device drivers built in multiple driver development models. For our analysis, we have chosen test suites developed for WDM and KMDF drivers. These comprehensive test suites include drivers of different sizes (1-30K LOC), with a mix of test drivers written to test SDV rules (with injected defects), sample drivers that are shipped in WDK to provide guidance to driver developers, and drivers that are shipped as part of the Windows operating system. Note that all the data presented in this section has been extracted from test runs performed by the test team.

Most of the metrics used in this section were explained in previous sections. New to this section are the following metrics. A "check" is a run on one driver for one rule. A "LIGHTWEIGHTPASS" result refers to the fact that before starting the CEGAR loop, SDV first applies property instrumentation, pointer analysis, and function pointer resolution to show that the error state of a rule is not reachable in the call-graph of the C program. An "out of resource" (OOR) result refers to checks that exceeded the allocated time or memory resources. The NUR results include both the OOR and GiveUp results.

SDV can report a false defect for a number of reasons: a bug in the verification engine, a bug in the rule, or a bug in the environment model (the C code that calls into a driver and provides stubs of kernel routines used by drivers). Hence, improvements to any of those components can result in the reduction in the number of false defects.

SDV can report a Pass result which is actually a "false verification", due to overconstraining of the abstract transition relation. This problem can be revealed by comparing SDV runs with different engines, for example, SLAM1 versus SLAM2. In particular, we observed that some Pass results with SLAM1 turn into Defect or OOR results with SLAM2. The OOR result would mostly occur on the runs for large drivers and/or hard rules. Specific data for such cases are presented in Tables I and II.

For the purposes of profiling SDV and comparing the analysis engines, we use the two official releases of SDV, SDV 1.6 and 2.0, and also runs of SDV 2.0 with SLAM1, for a more accurate comparison.

Table I compares the data for the WDM drivers for SLAM1 as part of both SDV 1.6 and SDV 2.0, and for SLAM2 as part of SDV 2.0. Dashes in the table indicate that the data is not available for that particular metric.

SDV 2.0 (SLAM1)	SDV 2.0 (SLAM2)	COUNT	CHANGE
OOR	Pass	31	$\surd$
Defect (false)	Pass	5	$\surd$
Defect (true)	Pass	2	$\times$
GiveUp	Pass	15	$\surd$
OOR	Defect (true)	2	$\surd$
Defect (false)	OOR	36	$\surd$
GiveUp	OOR	13	$\surd$
Pass	OOR	64	$\sim$
OOR	GiveUp	2	$\sim$
Defect (false)	GiveUp	11	$\surd$
Defect (true)	GiveUp	1	$\times$

TABLE II
BREAKDOWN OF CHANGES OBSERVED BETWEEN SLAM1 AND SLAM2 USING SDV 2.0 FOR WDM DRIVERS.

Table I shows significant reduction in the number of false defects and GiveUp results for SLAM2. This is due to the better precision of coarse-grained abstraction, as well as to the improved trace validation and predicate discovery. All three factors play a role in these improvements. In particular, better predicate discovery helps make progress (discover new predicates) in the cases where SLAM1 couldn't; more precise abstraction reduces the need for additional predicates in the first place. The number of NURs significantly decreased between SDV 1.6 and SDV 2.0 for both engines. This is mostly due to the improvements in SDV environment and rules, in particular, NULL pointer dereference bugs. Those bugs have been found by running SDV with SLAM2 (but not with SLAM1). Finally, SLAM2 is faster in finding defects, but takes more time to prove Pass results. The time difference for the Pass results is due to the problem of overconstraining of the abstract transition relation in SLAM1, i.e., "false verification".

According to Table I, for WDM drivers, SLAM2 provides a useful result 96.7% of the time, and upon discovery of a defect, provides a 99.6% guarantee that this is a true defect.

Table II shows the breakdown of the individual results and changes observed between SDV 2.0 with SLAM1 and with SLAM2 for WDM drivers. The leftmost column is the result reported by SLAM1, followed by the result reported by SLAM2 and the count for such changes. The rightmost column indicates whether the changes are in favor ($\surd$), against

Metric	SDV 2.1 (SLAM2)	SDV 2.1 (YOGI)
LightweightPass results	2457	2457
Pass results	2556	2538
NUR results	3.3% (194/5727)	3.65% (209/5727)
Defects reported	520	523
False/reported defects	0.4% (2/520)	3.4% (18/523)
Missed defects	2	18
Time for identical pass	76922s	147189s (∼2x)
Time for identical defect	1795s	9984s (∼6x)

TABLE III
COMPARISON OF SLAM2 WITH YOGI USING SDV 2.1 FOR WDM DRIVERS.

SDV 2.1 (YOGI)	SDV 2.1 (SLAM2)	COUNT	CHANGE
NUR	Pass	114	√
Defect (false)	Pass	8	√
NUR	Defect (true)	10	√
Pass	Defect (true)	8	√
Defect (false)	OOR	1	√
Pass	OOR	94	×
NUR	GiveUp	4	∼
Defect (false)	GiveUp	10	√
Defect (true)	GiveUp	2	×
Pass	GiveUp	2	×

TABLE IV
BREAKDOWN OF CHANGES OBSERVED BETWEEN SDV 2.1 WITH SLAM2 AND SDV 2.1 WITH YOGI FOR WDM DRIVERS.

(×), or neutral (∼), for SLAM2 with respect to SLAM1.

There are 28 cases where GiveUp results by SLAM1 changed into Pass (15 cases) or OOR (13 cases) for SLAM2. The change from GiveUp to OOR indicates that progress has been made beyond the GiveUp point (but not until a definite result, due to insufficient resources). Out of 14 cases where SLAM2 produces a GiveUp, there are 11 cases for which SLAM1 produces a (false) defect. There are 36 cases where false defects reported by SLAM1 changed into OOR for SLAM2, which is clearly favorable for SLAM2. Finally, we mark the changes from the Pass result for SLAM1 into the OOR result for SLAM2 (64 cases) as neutral, because we have a strong evidence that SLAM1 was able to prove the Pass result by overconstraining, but it is unrealistic to investigate each case to validate this claim. Note that the two defects found by SLAM1 but not by SLAM2 are being investigated.

Table III presents a comparison of SLAM2 with YOGI [NRTT09] for WDM drivers. SLAM2 provides 7% fewer NURs, fewer false defects (2 versus 18), while finding 18 true defects that YOGI misses (the respective number for YOGI is 2), and is two times faster than YOGI. Note that YOGI does not report GiveUp results in the same way as SLAM does, so this analysis is not performed - instead, the GiveUp cases are included into the NUR cases. Notably, YOGI takes 6 times longer for finding the same defects as SLAM2, but only 2 times longer for finding the same proofs as SLAM2.

According to Table III, for WDM drivers, YOGI provides a useful result 96.3% of the time, and upon discovery of a defect, provides a 96.6% guarantee that this is a true defect. SLAM2 provides a useful result 96.6% of the time and a true defect guarantee of 99.8%.

Table IV provides a breakdown of the changes observed between SLAM2 and YOGI using SDV 2.1 on WDM drivers. The format is the same as in Table II. The table shows that in general, SLAM2 provides a higher rate of useful results: 114 Pass results and 10 defect reports for which YOGI reports NUR. There are 8 Pass results for SLAM2 for which YOGI reports false defects. There are 11 cases where SLAM2 finishes with an NUR result, and YOGI reports a false defect.

On the other hand, there are two cases where YOGI finds a defect which SLAM2 is unable to find (GiveUp) - those proved to be useful in identifying limitations of SLAM2.

Table V compares SLAM1, SLAM2, and YOGI using SDV on KMDF drivers. Note that KMDF drivers are significantly smaller than WDM drivers, due to the higher level of the APIs provided by the KMDF model. This explains the comparable results for both SLAM1 and SLAM2. There is a significant improvement in the number of NURs (1% to 0.04%) and false defects (25% to 0%) between SDV 1.6 and SDV 2.0, regardless of the SLAM version. This improvement is primarily due to the improvements in the KMDF environment model and rules between the two releases. Comparing SLAM2 to YOGI, we observe significantly larger number of NURs for YOGI: 117 vesus 2 for SLAM2. Additionally, YOGI takes 8 times longer than SLAM2 for checks with the identical results. Note that the defect analysis (true versus false defects) for comparing YOGI to SLAM2 has not been performed for KMDF drivers.

Table V shows the comparison of SLAM1, SLAM2, and YOGI for KMDF drivers. SLAM2 provides a useful result 99.8% of the time, and upon discovery of a defect, provides a 100% guarantee that this is a true defect. Comparatively, YOGI provides a useful result 97.8% of the time.

In summary, our comprehensive analysis of the realistic empirical data confirms that SLAM2 provides highly reliable results by reporting defects with a high degree of confidence that those are true defects, or finding proofs when there's no defect. Our comparison involves two driver models and three verification engines and is based on the data obtained in an industrial setting by independent testers.

VII. RELATED WORK

Coarse-grained Abstraction. After the development of SLAM1, it became clear that we were underutilizing the power of automated theorem provers such as Z3 to cope with complex Boolean formulae, relying instead on the Boolean program model checker to deal with arbitrary Boolean combinations of predicates. With coarse-grain abstraction, we give Z3 a little bit more work to do and increase the precision of the abstraction. However, one can do much more, as explored by Beyer and colleagues in their work on "software model checking via large-block encoding" [BCG+09]. They show that one can abstract over loop-free fragments of code such as sequences of **if-then-else** statements. They compared their large-block approach to the approach where each single

Metric	SDV 1.6 (SLAM1)	SDV 2.0 (SLAM1)	SDV 2.0 (SLAM2)	SDV 2.1 (SLAM2)	SDV 2.1 (YOGI)
Driver	51	51	51	51	51
Rules	61	102	102	103	103
Total checks	3111	5202	5202	5253	5253
NUR results	1% (31/3111)	0.04% (2/5202)	0.04% (2/5202)	0.04% (2/5253)	2.2% (117/5253)
Defects reported	300	271	271	271	-
False defects	25% (75/300)	0% (0/271)	0% (0/271)	0% (0/271)	-
Total time for identical checks	-	-	-	8414s	63645s (∼8x)

TABLE V
COMPARISON OF SLAM1, SLAM2 AND YOGI USING SDV FOR KMDF DRIVERS.

statement is abstracted in isolation. It would be interesting to compare their approach to the presented approach.

Multiple Inconsistencies Per Trace. We are not aware of other work that explores the idea of finding multiple invalid subtraces of a single counterexample trace. We found this technique to be very valuable for making more progress, but it does come at an increased cost in model checking, as more predicates are introduced. The ability to recover from "irrelevant refinements" (retracting predicats that are not useful) would be valuable in order to explore multiple inconsistencies during CEGAR. McMillan explores how to give CEGAR such a flexibility, which would be very helpful for the case of detecting multiple inconsistencies. [McM10]

Path/Trace-Sensitive Pointer Aliasing. SLAM2's use of pointer aliasing information, computed by forward symbolic execution, to refine the precondition computation is very similar to that used by the DASH algorithm [BNRS08], that forms the basis of the the YOGI tool we compare against. However, SLAM2 only uses this technique during symbolic execution and not the abstraction process, as YOGI does.

VIII. CONCLUSION

We have described major improvements in the SLAM verification engine, shipped with SDV 2.0 in September, 2009 as a part of the Windows 7 WDK. SLAM2 significantly improved the reliability, robustness and precision of SDV. SDV adoption inside Microsoft proved to be very successful, with "SDV clean" being a requirement for Microsoft drivers to be shipped with Windows 7.

Our results show that SDV 2.0 with SLAM2 is an industrial quality static analysis tool, compared to previous versions of SDV based on SLAM1, which was in many respects a research prototype. The SDV tool has benefited greatly from a multi-engine approach, allowing us to easily compare SLAM2 to YOGI.

REFERENCES

[BBC+06] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys 06*, pages 73–85, 2006.

[BBdML10] T. Ball, E. Bounimova, L. de Moura, and V. Levin. Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2. Technical Report MSR-TR-2010-24, Microsoft Research, 2010.

[BCDR04] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS 04: Tools and Algorithms for the Construction and Analysis of Systems*, pages 388–403, 2004.

[BCG+09] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD 09: Formal Methods in Computer Aided Design*, pages 25–32, 2009.

[BMR05] T. Ball, T. D. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst.*, 27(2):314–343, 2005.

[BNRS08] N. 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.

[BPR01] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, pages 268–283, 2001.

[BPR02] T. Ball, A. Podelski, and S. K. Rajamani. On the relative completeness of abstraction refinement. In *TACAS 02: Tools and Algorithms for Construction and Analysis of Systems*, pages 158–172, April 2002.

[BR02a] T. Ball and S. K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, January 2002.

[BR02b] 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, January 2002.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252, 1977.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, pages 72–83. 1997.

[LBC05] S. K. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. In *CAV 05: Computer-Aided Verification*, pages 24–38, 2005.

[MB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 08: Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[McM10] K. L. McMillan. Lazy annotation for program testing and verification. In *CAV 10: Computer-Aided Verification*, 2010.

[NRTT09] 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.