SLAM is a program-analysis engine used to check if clients of an API follow the API's stateful usage rules.

BY THOMAS BALL, VLADIMIR LEVIN, AND SRIRAM K. RAJAMANI

A Decade of Software Model Checking with SLAM

LARGE-SCALE SOFTWARE DEVELOPMENT is a notoriously difficult problem. Software is built in layers, and APIs are exposed by each layer to its clients. APIs come with usage rules, and clients must satisfy them while using the APIs. Violations of API rules can cause runtime errors. Thus, it is useful to consider whether API rules can be formally documented so programs using the APIs can be checked at compile time for compliance against the rules.

Some API rules (such as agreement on the number of parameters and data types of each parameter) can be checked by compilers. However, certain rules involve hidden state; for example, consider the rule that the acquire method and release method of a

spinlock must be done in strict alternation and the rule that a file can be read only after it is opened. We built the SLAM engine (SLAM from now on) to allow programmers to specify stateful usage rules and statically check if clients follow such rules. We wanted SLAM to be scalable and at the same time have a very low false-error rate. To scale the SLAM engine, we constructed abstractions that retain only information about certain predicates related to the property being checked. To reduce false errors, we refined abstractions automatically using counterexamples from the model checker. Constructing and refining abstractions for scaling model checking has been known for more than 15 years; Kurshan[35] is the earliest reference we know.

SLAM automated the process of abstraction and refinement with counterexamples for programs written in common programming languages (such as C) by introducing new techniques to handle programming-language constructs (such as pointers, procedure calls, and scoping constructs for variables).[2,4–8] Independently and simultaneously with our work, Clarke et al.[17] automated abstraction and refinement with counterexamples in the context of hardware, coining the term "counterexample-driven abstraction refinement," or CEGAR, which we use to refer to this technique throughout

» **key insights**

■ **Even though programs have many states, it is possible to construct an abstraction of a program fine enough to represent parts of a program relevant to an API usage rule and coarse enough for a model checker to explore all the states.**

■ **SLAM synthesizes and extends diverse ideas from model checking, theorem proving, and data-flow analysis to automate construction, checking, and refinement of abstractions.**

■ **SLAM showed that such abstractions can be constructed automatically for real-world programs, becoming the basis of Microsoft's Static Driver Verifier tool.**
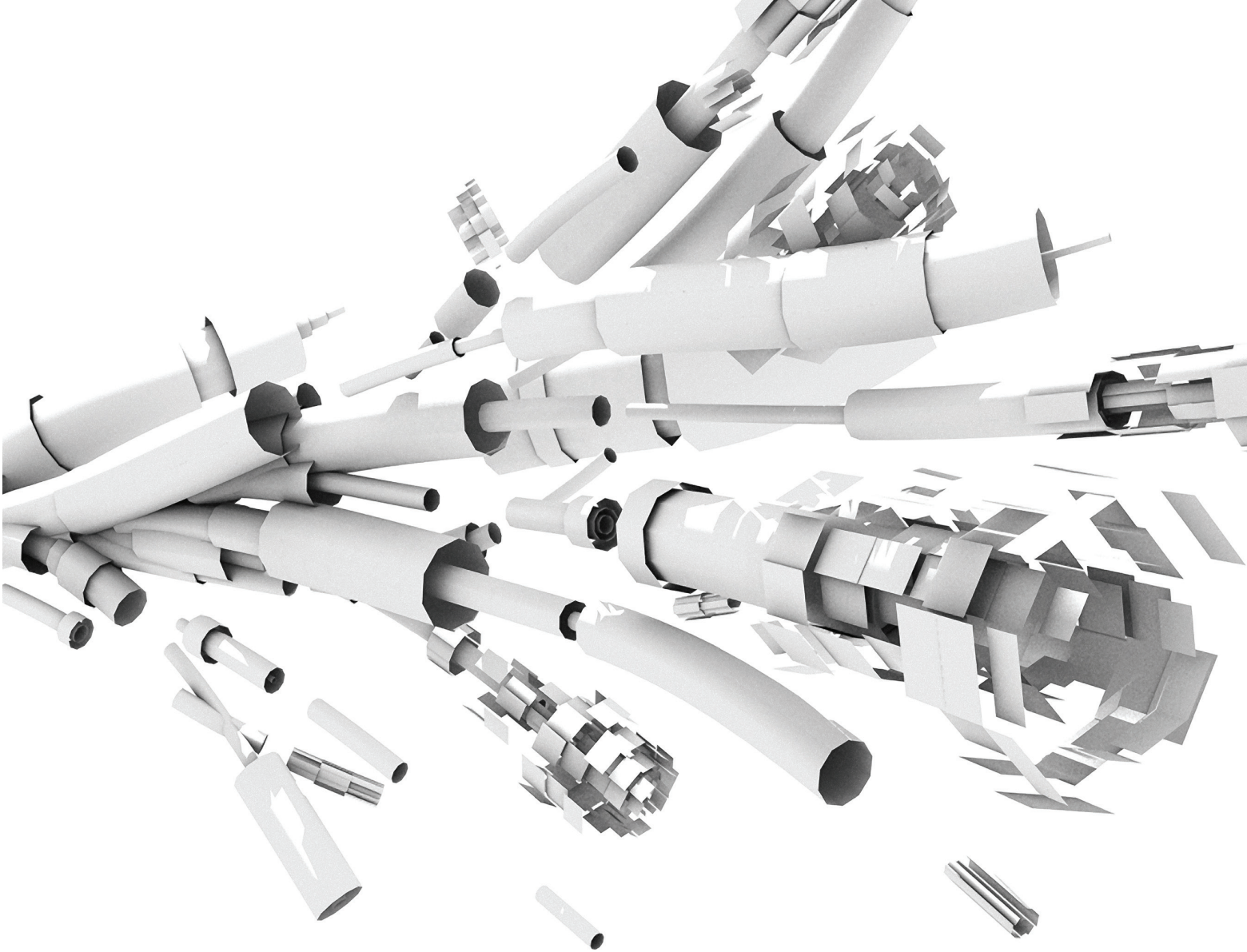
this article. The automation of CEGAR for software is technically more intricate, since software, unlike hardware, is infinite state, and programming languages have more expressive and complex features compared to hardware-description languages. Programming languages allow procedures with unbounded call stacks (handled by SLAM using pushdown model-checking techniques), scoping of variables (exploited by SLAM for efficiency), and pointers allowing the same memory to be aliased by different variables (handled by SLAM using pointer-alias-analysis techniques).

We also identified a "killer-app" for SLAM—checking if Windows device drivers satisfy driver API usage rules. We wrapped SLAM with a set of rules specific to the Windows driver API and a tool chain to enable pushbutton validation of Windows drivers, resulting in a tool called "static driver verifier," or SDV. Such tools are stra-

tegically important for the Windows device ecosystem, which encourages and relies on hardware vendors making devices and writing Windows device drivers while requiring vendors to provide evidence that the devices and drivers perform acceptably. Because many drivers use the same Windows-driver API, the cost of manually specifying the API rules and writing them down is amortized over the value obtained by checking the same rules over many device drivers.

Here, we offer a 10-year retrospective of SLAM and SDV, including a self-contained overview of SLAM, our experience taking SLAM to a full-fledged SDV product, a description of how we built and deployed SDV, and results obtained from the use of SDV.

SLAM
Initially, we coined the label SLAM as an acronym for "software (specifications), programming languages,

abstraction, and model checking." Over time, we used SLAM more as a forceful verb; to "SLAM" a program is to exhaustively explore its paths and eliminate its errors. We also designed the "Specification Language for Interface Checking," or SLIC,[9] to specify stateful API rules and created the SLAM tool as a flexible verifier to check if code that uses the API follows the SLIC rules. We wanted to build a verifier covering all possible behaviors of the program while checking the rule, as opposed to a testing tool that checks the rule on a subset of behaviors covered by the test.

In order for the solution to scale while covering all possible behaviors, we introduced Boolean programs. Boolean programs are like C programs in the sense that they have all the control constructs of C programs—sequencing, conditionals, loops, and procedure calls—but allow only Boolean variables (with local, as well as global,

scope). Boolean programs made sense as an abstraction for device drivers because we found that most of the API rules drivers must follow tend to be control-dominated, and so can be checked by modeling control flow in the program accurately and modeling only a few predicates about data relevant to each rule being checked.

The predicates that need to be "pulled into" the model are dependent on how the client code manages state relevant to the rule. CEGAR is used to discover the relevant state automatically so as to balance the dual objectives of scaling to large programs and reducing false errors.

SLIC specification language. We designed SLAM to check temporal safety properties of programs using a well-defined interface or API. Safety properties are properties whose violation is witnessed by a finite execution path. A simple example of a safety property is that a lock should be alternatively acquired and released. SLIC allows us to encode temporal safety properties in a C-like language that defines a safety automaton[44] that monitors a program's execution behavior at the level of function calls and returns. The automaton can read (but not modify) the state of the C program that is visible at the function call/return interface, maintain a history, and signal the occurrence of a bad state.

A SLIC rule includes three components: a static set of state variables, described as a C structure; a set of events and event handlers that specify state transitions on the events; and a set of annotations that bind the rule to various object instances in the program (not shown in this example). As an example of a rule, consider the locking rule in Figure 1a. Line 1 declares a C structure containing one field `state`, an enumeration that can be either `Unlocked` or `Locked`, to capture the state of the lock. Lines 3–5 describe an event handler for calls to `KeInitializeSpinLock`. Lines 7–13 describe an event handler for calls to the function `KeAcquireSpinLock`. The code for the handler expects the `state` to be in `Unlocked` and moves it to `Locked` (specified in line 9). If the `state` is already `Locked`, then the program has called `KeAcquireSpinLock` twice without an intervening call to `KeReleaseSpinLock` and is an error (line 9). Lines 15–21 similarly describe an event handler for calls to the function `KeReleaseSpinLock`[a]. Figure 1b is a piece of code that uses the functions `KeAcquireSpinLock` and `KeReleaseSpinLock`. Figure 1c

---

a A more detailed example of this rule would handle different instances of locks, but we cover the simple version here for ease of exposition.

is the same code after it has been instrumented with calls to the appropriate event handlers. We return to this example later.

CEGAR via predicate abstraction. Figure 2 presents ML-style pseudocode of the CEGAR process. The goal of SLAM is to check if all executions of the given C program *P* (type *cprog*) satisfy a SLIC rule *S* (type *spec*).

The instrument function takes the program *P* and SLIC rule *S* as inputs and produces an instrumented program *P′* as output, based on the product-construction technique for safety properties described in Vardi and Wolper.[44] It hooks up relevant events via calls to event handlers specified in the rule *S*, maps the error statements in the SLIC rule to a unique error state in *P′*, and guarantees that *P* satisfies *S* if and only if the instrumented program *P′* never reaches the error state. Thus, this function reduces the problem of checking if *P* satisfies *S* to checking if *P′* can reach the error state.

The function *slam* takes a C program *P* and SLIC rule specification *S* as input and passes the instrumented C program to the tail-recursive function *cegar*, along with the predicates extracted from the specification *S* (specifically, the guards that appear in *S* as predicates).

The first step of the *cegar* function is to abstract program *P′* with respect to

---

Figure 1. (a) Simplified SLIC locking rule; (b) code fragment using spinlocks; (c) fragment after instrumentation.

```
1    state { enum {Unlocked, Locked} state; }
2
3    KeInitializeSpinLock.call {
4        state = Unlocked;
5    }
6
7    KeAcquireSpinLock.call {
8        if ( state == Locked ) {
9            error;
10       } else {
11           state = Locked;
12       }
13   }
14
15   KeReleaseSpinLock.call {
16       if ( !(state == Locked) ) {
17           error;
18       } else {
19           state = Unlocked;
20       }
21   }
22
                    (a)
```

```
1    ..
2    KeInitializeSpinLock();
3    ..
4    ..
5    if(x > 0)
6        KeAcquireSpinlock();
7    count = count+1;
8    devicebuffer[count] = localbuffer[count];
9    if(x > 0)
10       KeReleaseSpinLock();
11   ...
12   ...

                    (b)
```

```
1    ..
2    { state = Unlocked;
3      KeInitializeSpinLock();}
4    ..
5    ..
6    if(x > 0)
7        { SLIC_KeAcquireSpinLock_call();
8          KeAcquireSpinlock(); }
9    count = count+1;
10   devicebuffer[count] = localbuffer[count];
11   if(x > 0)
12       { SLIC_KeReleaseSpinLock_call();
13         KeReleaseSpinLock(); }
14   ...
15   ...

                    (c)
```

the predicate set preds to create a Boolean program abstraction B. The automated transformation of a C program into a Boolean program uses a technique called predicate abstraction, first introduced in Graf and Saïdi[29] and later extended to work with programming-language features in Ball et al.[2] and Ball et al.[3]

The program B has exactly the same control-flow skeleton as program P'. By construction, for any set of predicates preds, every execution trace of the C program P' also is an execution trace of $B = abstract(P', preds)$; that is, the execution traces of P' are a subset of those of B. The Boolean program B models only the portions of the state of P' relevant to the current SLIC rule, using nondeterminism to abstract away irrelevant state in P'.

Once the Boolean program B is constructed, the *check* function exhaustively explores the state space of B to determine if the (unique) error state is reachable. Even though all variables in B are Boolean, it can have procedure calls and a potentially unbounded call stack. Our model checker performs symbolic reachability analysis of the Boolean program (a pushdown system) using binary decision diagrams.[11] It

uses ideas from interprocedural data flow analysis[42,43] and builds summaries for each procedure to handle recursion and variable scoping.

If the *check* function returns **AbstractPass**, then the error state is not reachable in B and therefore is also not reachable in P'. In this case, SLAM has proved that the C program P satisfies the specification S. However, if the *check* function returns **AbstractFail** with witness trace trc, the error state is reachable in the Boolean program B but not necessarily in the C program P'. Therefore, the trace trc must be validated in the context of P' to prove it really is an execution trace of P'.

The function *symexec* symbolically executes the trace trc in the context of the C program P'. Specifically, it constructs a formula $\phi(P', trc)$ that is satisfiable if and only if there exists an input that would cause program P' to execute trace trc. If *symexec* returns **Satisfiable**, then SLAM has proved program P does not satisfy specification S and returns the counterexample trace trc.

If the function *symexec* returns **Unsatisfiable**(prf), then it has found a proof prf that there is no input that would cause P' to execute trace trc. The function *refine* takes this proof of

unsatisfiability, reduces it to a smaller proof of unsatisfiability, and returns the set of constituent predicates from this smaller proof. The function *refine* guarantees that the trace trc is not an execution trace of the Boolean program

$$abstract(P', preds \cup refine(pr\,f))$$

The ability to refine the (Boolean program) abstraction to rule out a spurious counterexample is known as the progress property of the CEGAR process.

Despite the progress property, the CEGAR process offers no guarantee of terminating since the program P' may have an intractably large or infinite number of states; it can refine the Boolean program forever without discovering a proof of correctness or proof of error.

However, as each Boolean program is guaranteed to overapproximate the behavior of the C program, stopping the CEGAR process before it terminates with a definitive result is no different from any terminating program analysis that produces false alarms. In practice, SLAM terminates with a definite result over 96% of the time on large classes of device drivers: for Windows Driver Framework (WDF) drivers, the figure is

Figure 2. Graphical illustration and ML-style pseudocode of CEGAR loop.
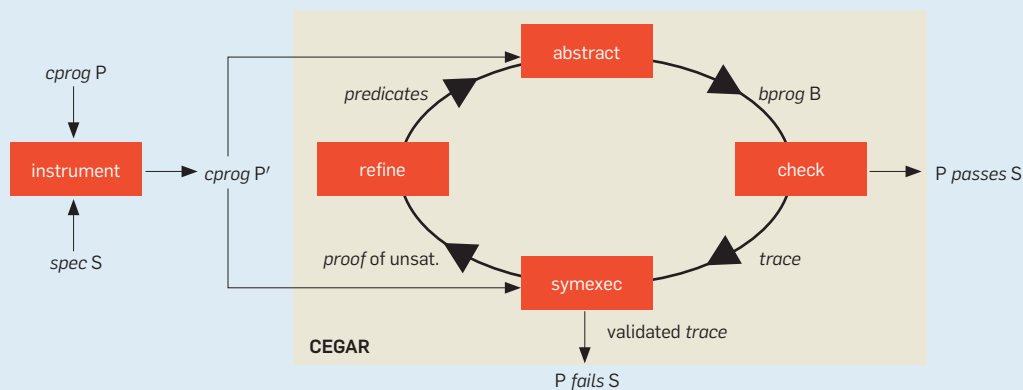
```
type cprog, spec, predicates, bprog, trace, proof

type result =
    Pass | Fail of trace

type chkresult =
    AbstractPass | AbstractFail of trace

type excresult =
    Satisable | Unsatisable of proof
```

```
let rec cegar (P':cprog) (preds :predicates) : result =
    let B: bprog = abstract (P',preds) in
    match check(B) with
    | AbstractPass -> Pass
    | AbstractFail(trc) ->
        match symexec(P', trc) with
        | Satisable -> Fail(trc)
        | Unsatisable(prf) -> cegar P' ( preds ∪ (refine prf))

let slam ( P:cprog) (S:spec) : result =
    cegar (instrument (P,S)) (preds S)
```

100%, and for Windows Driver Model (WDM) drivers, the figure is 97%.

Example. We illustrate the CEGAR process using the SLIC rule from Figure 1a and the example code fragment in Figure 1b. In the program, we have a single spinlock being initialized at line 4. The spinlock is acquired at line 8 and released at line 12. However, both calls `KeAcquireSpinLock` and `KeReleaseSpinLock` are guarded by the conditional ($x > 0$). Thus, tracking correlations between such conditionals is important for proving this property. Figures 3a and 3b show the Boolean program obtained by the first application of the *abstract* function to the code from Figures 1a and 1c, respectively.

Figure 3a is the Boolean program abstraction of the SLIC event handler code. Recall that the instrumentation step guarantees there is a unique error state. The function `slic _ error` at line 1 represents that state; that is, the function `slic _ error` is unreachable if and only if the program satisfies the SLIC rule. There is one Boolean variable named {state==Locked}; by convention, we name each Boolean variable with the predicate it stands for, enclosed in curly braces. In this case, the predicate comes from the guard in the SLIC rule (Figure 1a, line 8). Lines 5–8 and lines 10–13 of Figure 3a show the Boolean procedures corresponding to the SLIC event handlers `SLIC _ KeAcquireSpinLock _ call` and `SLIC_KeReleaseSpinLock_ call` from Figure 1a.

Figure 3b is the Boolean program abstraction of the SLIC-instrumented C program from Figure 1c. Note the Boolean program has the same control flow as the C program, including procedure calls. However, the conditionals at lines 7 and 12 of the Boolean program are nondeterministic since the Boolean program does not have a predicate that refers to the value of variable x. Also note that the references to variables `count`, `devicebuffer`, and `localbuffer` are elided in lines 10 and 11 (replaced by `skip` statements in the Boolean program) since the Boolean program does not have predicates that refer to these variables.

The abstraction in Figure 3b, though a valid abstraction of the instrumented C, is not strong enough to prove the program conforms to the SLIC rule. In particular, the reachability analysis of the Boolean program performed by the check function will find that `slic _ error` is reachable via the trace 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, which skips the call to `SLIC _ KeAcquireSpinLock _ call` at line 8 and performs the call to `SLIC _ KeReleaseSpinLock _ call` at line 13. Since the Boolean variable state==Lock is false, `slic _ error` will be called in line 11 of Figure 3a.

SLAM feeds this error trace to the symexec function that executes it symbolically over the instrumented C program in Figure 1c and determines the trace is not executable since the branches in "if" conditions are correlated. In particular, the trace is not executable because there does not exist a value for variable x such that ($x > 0$) is false (skipping the body of the first conditional) and such that ($x > 0$) is true (entering the body of the second conditional). That is, the formula $\exists x.(x \le 0) \wedge (x > 0)$ is unsatisfiable. The result of the refine function is to add the predicate {$x>0$} to the Boolean program to refine it. This addition results in the Boolean program abstraction in Figure 3c, including the Boolean variable {$x>0$}, in addition to {state==Locked}.

Using these two Boolean variables, the abstraction in Figure 3c is strong enough to prove `slic _ error` is unreachable for all possible executions of the Boolean program, and hence SLAM proves this Boolean program satisfies the SLIC rule. Since the Boolean program is constructed to be an overapproximation of the C program in Figure 1c, the C program indeed satisfies the SLIC rule.

From SLAM to SDV
SDV is a completely automatic tool (based on SLAM) device-driver developers can use at compile time. Requiring nothing more than the build script of the driver, the SDV tool runs fully automatically and checks a set of pre-packaged API usage rules on the device driver. For every usage rule violated by the driver, SDV presents a possible execution trace through the driver that shows how the rule can be violated.

Figure 3. (a) Boolean program abstraction for locking and unlocking routines; (b) Boolean program: CEGAR iteration 1; (c) Boolean program: CEGAR iteration 2.

```
1  slic_error() { assert(false); }
2
3  bool {state==Locked};
4
5  SLIC_KeAcquireSpinLock_call() {
6     if( {state==Locked}) slic_error();
7     else {state==Locked} := true;
8  }
9
10 SLIC_KeReleaseSpinLock_call() {
11    if( !{state==Locked}) slic_error();
12    else {state==Locked} := false;
13 }
14
```
(a)

```
1  ...
2  ...
3  {state==Locked} := false;
4  KeInitializeSpinLock();
5  ...
6  ...
7  if(*)
8     { SLIC_KeAcquireSpinLock_call();
9        KeAcquireSpinLock(); }
10 skip;
11 skip;
12 if(*)
13    { SLIC_KeReleaseSpinLock_Call();
14       KeReleaseSpinLock(); }
15 ...
16 ...
```
(b)

```
1  bool {x > 0};
2  ...
3  {state==Locked} := false;
4  KeInitializeSpinLock();
5  ...
6  ...
7  if({x>0})
8     { SLIC_KeAcquireSpinLock_call();
9        KeAcquireSpinLock(); }
10 skip;
11 skip;
12 if({x>0})
13    { SLIC_KeReleaseSpinLock_Call();
14       KeReleaseSpinLock(); }
15 ..
16 ...
```
(c)

Model checking is often called "push-button" technology,[16] giving the impression that the user simply gives the system to the model checker and receives useful output about errors in the system, with state-space explosion being the only obstacle. In practice, in addition to state-space explosion, several other obstacles can inhibit model checking being a "push-button" technology: First, users must specify the properties they want to check, without which there is nothing for a model checker to do. In complex systems (such as the Windows driver interface), specifying such properties is difficult, and these properties must be debugged. Second, due to the state-explosion problem, the code analyzed by the model checker is not the full system in all its gory complexity but rather the composition of some detailed component (like a device driver) with a so-called "environment model" that is a highly abstract, human-written description of the other components of the system—in our case, kernel procedures of the Windows operating system. Third, to be a practical tool in the toolbox of a driver developer, the model checker must be encapsulated in a script incorporating it in the driver development environment, then feed it with the driver's source code and report results to the user. Thus, creating a push-button experience for users requires much more than just building a good model-checking engine.

Here, we explore the various components of the SDV tool besides SLAM: driver API rules, environment models, scripts, and user interface, describing how they've evolved over the years, starting with the formation of the SDV team in Windows in 2002 and several internal and external releases of SDV.

API rules. Different classes of devices have different requirements, leading to class-specific driver APIs. Thus, networking drivers use the NDIS API, storage drivers use the StorPort and MPIO APIs, and display drivers the WDDM API. A new API called WDF was designed to provide higher-level abstractions for common device drivers. As described earlier, SLIC rules capture API-level interactions, though they are not specific to a particular device driver but to a whole class of drivers that use a common API. Such a specification

We wanted to build a verifier covering all possible behaviors of the program while checking the rule, as opposed to a testing tool that checks the rule on a subset of behaviors covered by the test.

means the manual effort of writing rules can be amortized by checking the rules on thousands of device drivers using the API. The SDV team has made significant investment in writing API rules and teaching others in Microsoft's Windows organization to write API rules.

Environment models. SLAM is designed as a generic engine for checking properties of a closed C program. However, a device driver is not a closed program with a main procedure but rather a library with many entry points (registered with and called by the operating system). This problem is standard to both program analysis and model checking.

Before applying SLAM to a driver's code, we first "close" the driver program with a suitable environment consisting of a top layer called the *harness*, a `main` procedure that calls the driver's entry points, and a bottom layer of *stubs* for the Windows API functions that can be called by the device driver. Thus, the harness calls into the driver, and the driver calls the stubs.

Most API rules are local to a driver's entry points, meaning a rule can be checked independently on each entry point. However, some complex rules deal with sequences of entry points. For the rules of the first type, the body of the harness is a nondeterministic switch in which each branch calls a single and different entry point of the driver. For more complex rules, the harness contains a sequence of such nondeterministic switches.

A stub is a simplified implementation of an API function intended to approximate the input-output relation of the API function. Ideally, this relation should be an overapproximation of the API function. In many cases, a driver API function returns a scalar indicating success or failure. In these cases, the API stub usually ends with a nondeterministic switch over possible return values. In many cases, a driver API function allocates a memory object and returns its address, sometimes through an output pointer parameter. In these cases, the harness allocates a small set of such memory objects, and the stub picks up one of them and returns its address.

Scaling rules and models. Initially, we (the SDV team) wrote the API rules in SLIC based on input from driver API

experts. We tested them on drivers with injected bugs, then ran SDV with the rules on real Windows drivers. We discussed the bugs found by the rules with driver owners and API experts to refine the rules. At that time, a senior manager said, "It takes a Ph.D. to develop API rules." Since then, we've invested significant effort in creating a discipline for writing SLIC rules and spreading it among device-driver API developers and testers.

In 2007, the SDV team refined the API rules and formulated a set of guidelines for rule development and driver environment model construction. This helped us transfer rule development to two software engineers with backgrounds far removed from formal verification, enabling them to succeed and later spread this form of rule development to others. Since 2007, driver API teams have been using summer interns to develop new API rules for WDF, NDIS, StorPort, and MPIO APIs and for an API used to write file system mini-filters (such as antiviruses) and Windows services. Remarkably, all interns have written API rules that found true bugs in real drivers.

SDV today includes more than 470 API rules. The latest version SDV 2.0 (released with Windows 7 in 2009) includes more than 210 API rules for the WDM, WDF, and NDIS APIs, of which only 60 were written by formal verification experts. The remaining 150 were written or modified from earlier drafts by software engineers or interns with no experience in formal verification.

Worth noting is that the SLIC rules for WDF were developed during the design phase of WDF, whereas the WDM rules were developed long after WDM came into existence. The formalization of the WDF rules influenced WDF design; if a rule could not be expressed naturally in SLIC, the WDF designers tried to refactor the API to make it easier to verify. This experience showed that verification tools (such as SLAM) can be forward-looking design aids, in addition to being checkers for legacy APIs (such as WDM).

Scripts. SDV includes a set of scripts that perform various functions: combining rules and environment models; detecting source files of a driver and its build parameters; running the SLIC compiler on rules and the C compiler

A unique SLAM contribution is the complete automation of CEGAR for software written in expressive programming languages (such as C).

on a driver's and environment model's source code to generate an intermediate representation (IR); invoking SLAM on the generated IR; and reporting the summary of the results and error traces for bugs found by SLAM in a GUI.

The SDV team worked hard to ensure these scripts would provide a very high degree of automation for the user. The user need not specify anything other than the build scripts used to build the driver.

SDV Experience
The first version of SDV (1.3, not released externally outside Microsoft) found, on average, one real bug per driver in 30 sample drivers shipped with the Driver Development Kit (DDK) for Windows Server 2003. These sample drivers were already well tested. Eliminating defects in the WDK samples is important since code from sample drivers is often copied by third-party driver developers.

Versions 1.4 and 1.5 of SDV were applied to Windows Vista drivers. In the sample WDM drivers shipped with the Vista WDK (WDK, the renamed DDK), SDV found, on average, approximately one real bug per two drivers. These samples were mostly modifications of sample drivers from the Windows Server 2003 DDK, with fixes applied for the defects found by SDV 1.3. The newly found defects were due to improvements in the set of SDV rules and to defects introduced due to modifications in the drivers.

For Windows Server 2008, SDV version 1.6 contained new rules for WDF drivers, with which SDV found one real bug per three WDF sample drivers. The low bug count is explained by simplicity of the WDF driver model described earlier and co-development of sample drivers, together with the WDF rules.

For the Windows 7 WDK, SDV 2.0 found, on average, one new real bug per WDF sample driver and few bugs on all the WDM sample drivers. This data is explained by more focused efforts to refine WDF rules and few modifications in the WDM sample drivers. SDV 2.0 shipped with 74 WDM rules, 94 WDF rules, and 36 NDIS rules. On WDM drivers, 90% of the defects reported by SDV are true bugs, and the rest are false errors. Further, SDV reports nonresults (such as timeouts

and spaceouts) on only 3.5% of all checks. On WDF drivers, 98% of defects reported by SDV are true bugs, and non-results are reported on only 0.04% of all checks. During the development cycle of Windows 7, SDV 2.0 was applied as a quality gate to drivers written by Microsoft and sample drivers shipped with the WDK. SDV was applied later in the cycle after all other tools, yet found 270 real bugs in 140 WDM and WDF drivers. All bugs found by SDV in Microsoft drivers were fixed by Microsoft. We do not have reliable data on bugs found by SDV in third-party device drivers.

Here, we give performance statistics from a recent run of SDV on 100 drivers and 80 SLIC rules. The largest driver in the set is about 30,000 lines of code, and the total size of all drivers is 450,000 lines of code. The total runtime for the 8,000 runs (each driver-rule combination is a run) is about 30 hours on an eight-core machine. We kill a run if it exceeds 20 minutes, and SDV yields useful results (either a bug or a pass) on over 97% of the runs. We thus find SDV checks drivers with acceptable performance, yielding useful results on a large fraction of the runs.

Limitations. SLAM and SDV also involve several notable limitations. Even with CEGAR, SLAM is unable to handle very large programs (with hundreds of thousands of lines of code). However, we also found SDV is able to give useful results for control-dominated properties and programs with tens of thousands of lines of code. Though SLAM handles pointers in a sound manner, in practice, it is unable to prove properties that depend on establishing invariants of heap data structures. SLAM handles only sequential programs, though others have extended SLAM to deal with bounded context switches in concurrent programs.[40] Our experience with SDV shows that in spite of these limitations, SLAM is very successful in the domain of device-driver verification.

Related Work
SLAM builds on decades of research in formal methods. Model checking[15,16,41] has been used extensively to algorithmically check temporal logic properties of models. Early applications of model checking were in hardware[38]

and protocol design.[32] In compiler and programming languages, abstract interpretation[21] provides a broad and generic framework to compute fixpoints using abstract lattices. The particular abstraction used by SLAM was called "predicate abstraction" by Graf and Saïdi.[29] Our contribution was to show how to perform predicate abstraction on C programs with such language features as pointers and procedure calls in a modular manner.[2,3] The predicate-abstraction algorithm uses an automated theorem prover. Our initial implementation of SLAM used the Simplify theorem prover.[23] Our current implementation uses the Z3 theorem prover.[22]

The Bandera project explored the idea of user-guided finite-state abstractions for Java programs[20] based on predicate abstraction and manual abstraction but without automatic refinement of abstractions. It also explored the use of program slicing for reducing the state space of models. SLAM was influenced by techniques used in Bandera to check typestate properties on all objects of a given type.

SLAM's Boolean program model checker (Bebop) computes fixpoints on the state space of the generated Boolean program that can include recursive procedures. Bebop uses the Context Free Language Reachability algorithm,[42,43] implementing it symbolically using Binary Decision Diagrams.[11] Bebop was the first symbolic model checker for pushdown systems. Since then, other symbolic checkers have been built for similar purposes,[25,36] and Boolean programs generated by SLAM have been used to study and improve their performance.

SLAM and its practical application to checking device drivers has been enthusiastically received by the research community, and several related projects have been started by research groups in universities and industry. At Microsoft, the ESP and Vault projects were started in the same group as SLAM, exploring different ways of checking API usage rules.[37] The Blast project[31] at the University of California, Berkeley, proposed a technique called "lazy abstraction" to optimize constructing and maintaining the abstractions across the iterations in the CEGAR loop. McMillan[39] proposed "in-

terpolants" as a more systematic and general way to perform refinement; Henzinger et al.[30] found predicates generated from interpolants have nice local properties that were then used to implement local abstractions in Blast.

Other contemporary techniques for analyzing C code against temporal rules include the meta-level compilation approach of Engler et al.[24] and an extension of SPIN developed by Holzmann[33] to handle ANSI C.[33] The Cqual project uses "type qualifiers" to specify API usage rules, using type inference to check C code against the type-qualifier annotations.[26]

SLAM works by computing an overapproximation of the C program, or a "may analysis," as described by Godefroid et al.[28] The may analysis is refined using symbolic execution on traces, as inspired by the PREfix tool,[12] or a "must analysis." In the past few years, must analysis using efficient symbolic execution on a subset of paths in the program has been shown to be very effective in finding bugs.[27] The Yogi project has explored ways to combine may and must analysis in more general ways.[28] Another way to perform underapproximation or must analysis is to unroll loops a fixed number of times and perform "bounded model checking"[14] using satisfiability solvers, an idea pursued by several projects, including CBMC,[18] F-Soft,[34] and Saturn.[1]

CEGAR has been generalized to check properties of heap-manipulating programs,[10] as well as the problem of program termination.[19] The Magic model checker checks properties of concurrent programs where threads interact through message passing.[13] And Qadeer and Wu[40] used SLAM to analyze concurrent programs through an encoding that models all interleavings with two context switches as a sequential program.

Conclusion
The past decade has seen a resurgence of interest in the automated analysis of software for the dual purpose of defect detection and program verification, as well as advances in program analysis, model checking, and automated theorem proving. A unique SLAM contribution is the complete automation of CEGAR for software written in expres-

sive programming languages (such as C). We achieved this automation by combining and extending such diverse ideas as predicate abstraction, interprocedural data-flow analysis, symbolic model checking, and alias analysis. Windows device drivers provided the crucible in which SLAM was tested and refined, resulting in the SDV tool, which ships as part of the Windows Driver Kit.

Acknowledgments

References

1. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., and Hawkins, P. An overview of the Saturn project. In *Proceedings of the 2007 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (San Diego, June 13–14). ACM Press, New York, 2007, 43–48.
2. Ball, T., Majumdar, R., Millstein, T., and Rajamani, S.K. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, UT, June 20–22). ACM Press, New York, 2001, 203–213.
3. Ball, T., Millstein, T.D., and Rajamani, S.K. Polymorphic predicate abstraction. *ACM Transactions on Programming Languages and Systems 27*, 2 (Mar. 2005), 314–343.
4. Ball, T., Podelski, A., and Rajamani, S.K. Boolean and Cartesian abstractions for model checking C programs. In *Proceedings of the Seventh International Conference on Tools and Algorithms for Construction and Analysis of Systems* (Genova, Italy, Apr. 2–6). Springer, 2001, 268–283.
5. Ball, T. and Rajamani, S.K. Bebop: A symbolic model checker for Boolean programs. In *Proceedings of the Seventh International SPIN Workshop on Model Checking and Software Verification* (Stanford, CA, Aug. 30–Sept. 1). Springer, 2000, 113–130.
6. Ball, T. and Rajamani, S.K. *Boolean Programs: A Model and Process for Software Analysis. Technical Report MSR-TR-2000-14.* Microsoft Research, Redmond, WA, Feb. 2000.
7. Ball, T. and Rajamani, S.K. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth International SPIN Workshop on Model Checking of Software Verification* (Toronto, May 19–20). Springer, 2001, 103–122.
8. Ball, T. and Rajamani, S.K. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, OR, Jan. 16–18). ACM Press, New York, Jan. 2002, 1–3.
9. Ball, T. and Rajamani, S.K. *SLIC: A Specification Language for Interface Checking. Technical Report MSR-TR-2001-21.* Microsoft Research, Redmond, WA, 2001.
10. Beyer, D., Henzinger, T.A., Théoduloz, G., and Zufferey, D. Shape refinement through explicit heap analysis. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering* (Paphos, Cyprus, Mar. 20–28). Springer, 2010, 263–277.
11. Bryant, R. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (Aug. 1986), 677–691.
12. Bush, W.R., Pincus, J.D., and Siela, D.J. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience 30*, 7 (June 2000), 775–802.
13. Chaki, S., Clarke, E., Groce, A., Jha, S., and Veith, H. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, OR, May 3–10). IEEE Computer Society, 2003, 385–395.
14. Clarke, E., Grumberg, O., and Peled, D. *Model Checking.* MIT Press, Cambridge, MA, 1999.
15. Clarke, E.M. and Emerson, E.A. Synthesis of synchronization skeletons for branching time temporal logic. In *Proceedings of the Workshop on Logic of Programs* (Yorktown Heights, NY, May 1981). Springer, 1982, 52–71.
16. Clarke, E.M., Emerson, E.A., and Sifakis, J. Model checking: Algorithmic verification and debugging. *Commun. ACM 52*, 11 (Nov. 2009), 74–84.
17. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement. In *Proceedings of the 12 International Conference on Computer-Aided Verification* (Chicago, July 15–19). Springer, 2000, 154–169.
18. Clarke, E.M., Kroening, D., and Lerda, F. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Barcelona, Mar. 29–Apr. 2). Springer, 2004, 168–176.
19. Cook, B., Podelski, A., and Rybalchenko, A. Abstraction refinement for termination. In *Proceedings of the 12th International Static Analysis Symposium* (London, Sept. 7–9). Springer, 2005, 87–101.
20. Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., and Zheng, H. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering* (Limerick, Ireland, June 4–11). ACM Press, New York, 2000, 439–448.
21. Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages* (Los Angeles, Jan.). ACM Press, New York, 1977, 238–252.
22. de Moura, L. and Bjørner, N. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Mar. 29–Apr. 6). Springer, 2008, 337–340.
23. Detlefs, D., Nelson, G., and Saxe, J.B. Simplify: A theorem prover for program checking. *Journal of the ACM 52*, 3 (May 2005), 365–473.
24. Engler, D., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation* (San Diego, Oct. 23–25). Usenix Association, 2000, 1–16.
25. Esparza, J. and Schwoon, S. A BDD-based model checker for recursive programs. In *Proceedings of the 13th International Conference on Computer Aided Verification* (Paris, July 18–22). Springer, 2001, 324–336.
26. Foster, J.S., Terauchi, T., and Aiken, A. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, June 17–19). ACM Press, New York, 2002, 1–12.
27. Godefroid, P., Levin, M.Y., and Molnar, D.A. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium* (San Diego, CA, Feb. 10–13). The Internet Society, 2008.
28. Godefroid, P., Nori, A.V., Rajamani, S.K., and Tetali, S.D. Compositional may-must program analysis: Unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Jan. 17–23). ACM Press, New York, 2010, 43–56.
29. Graf, S. and Saïdi, H. Construction of abstract state graphs with PVS. In *Proceedings of the Ninth International Conference on Computer-Aided Verification* (Haifa, June 22–25). Springer, 72–83.
30. Henzinger, T.A., Jhala, R., Majumdar, R., and McMillan, K.L. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Jan. 14–16). ACM Press, New York, 2004, 232–244.
31. Henzinger, T.A., Jhala, R., Majumdar, R., and Sutre, G. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium Principles of Programming Languages* (Portland, OR, Jan. 16–18). ACM Press, New York, 2002, 58–70.
32. Holzmann, G. The SPIN model checker. *IEEE Transactions on Software Engineering 23*, 5 (May 1997), 279–295.
33. Holzmann, G. Logic verification of ANSI-C code with SPIN. In *Proceedings of the Seventh International SPIN Workshop on Model Checking and Software Verification* (Stanford, CA, Aug. 30–Sept. 1). Springer, 2000, 131–147.
34. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., and Ashar, P. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science 404*, 3 (Sept. 2008), 256–274.
35. Kurshan, R. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ, 1994.
36. La Torre, S., Parthasarathy, M., and Parlato, G. Analyzing recursive programs using a fixed-point calculus. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, June 15–21). ACM Press, New York, 2009, 211–222.
37. Larus, J.R., Ball, T., Das, M., DeLine, R., Fähndrich, M., Pincus, J., Rajamani, S.K., and Venkatapathy, R. Righting software. *IEEE Software 21*, 3 (May/June 2004), 92–100.
38. McMillan, K. *Symbolic Model Checking: An Approach to the State-Explosion Problem.* Kluwer Academic Publishers, 1993.
39. McMillan, K.L. Interpolation and SAT-based model checking. In *Proceedings of the 15th International Conference on Computer-Aided Verification* (Boulder, CO, July 8–12). Springer, 2003, 1–13.
40. Qadeer, S. and Wu, D. KISS: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington, D.C., June 9–12). ACM Press, New York, 2004, 14–24.
41. Queille, J. and Sifakis, J. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium on Programming* (Torino, Italy, Apr. 6–8). Springer, 1982, 337–350.
42. Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural data flow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, Jan. 23–25). ACM Press, New York, 1995, 49–61.
43. Sharir, M. and Pnueli, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, N.D. Jones and S.S. Muchnick, Eds. Prentice-Hall, 1981, 189–233.
44. Vardi, M.Y. and Wolper, P. An automata theoretic approach to automatic program verification. In *Proceedings of the Symposium Logic in Computer Science* (Cambridge, MA, June 16–18). IEEE Computer Society Press, 1986, 332–344.

Thomas Ball (tball@microsoft.com) is a principal researcher, managing the Software Reliability Research group in Microsoft Research, Redmond, WA.

Vladimir Levin (vladlev@microsoft.com) is a principal software design engineer and the technical lead of the Static Driver Verification project in Windows in Microsoft, Redmond, WA.

Sriram Rajamani (sriram@microsoft.com) is assistant managing director of Microsoft Research India, Bangalore.