# SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft

Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani

Microsoft Corporation

**Abstract.** The SLAM project originated in Microsoft Research in early 2000. Its goal was to automatically check that a C program correctly uses the interface to an external library. The project used and extended ideas from symbolic model checking, program analysis and theorem proving in novel ways to address this problem. The SLAM analysis engine forms the core of a new tool called Static Driver Verifier (SDV) that systematically analyzes the source code of Windows device drivers against a set of rules that define what it means for a device driver to properly interact with the Windows operating system kernel.

We believe that the history of the SLAM project and SDV is an informative tale of the technology transfer of formal methods and software tools. We discuss the context in which the SLAM project took place, the first two years of research on the SLAM project, the creation of the SDV tool and its transfer to the Windows development organization. In doing so, we call out many of the basic ingredients we believe to be essential to technology transfer: the choice of a critical problem domain; standing on the shoulders of those who have come before; the establishment of relationships with "champions" in product groups; leveraging diversity in research and development experience and careful planning and honest assessment of progress towards goals.

## 1 Introduction

In the early days of computer science, the ultimate goal of formal methods and program verification was to provide technology that could rigorously prove programs fully correct. While this goal remains largely unrealized, many researchers now focus on the less ambitious but still important goal of stating *partial* specifications of program behavior and providing methodologies and tools to check their correctness. The growing interest in this topic is due to the technological successes and convergence of four distinct research areas–type checking, model checking, program analysis, and automated deduction–on the problems of software quality. Ideas about specification of properties, abstraction of programs, and algorithmic analyses from these four areas are coming together in new ways to address the common problem of software quality.

E. Boiten, J. Derrick, G. Smith (Eds.): IFM 2004, LNCS 2999, pp. 1–20, 2004.

The SLAM[1] project is just one of many exploring this idea. In early 2000 we set out to build a software tool that could automatically check that a C program correctly uses the interface to an external library. The outcome of this project is the SLAM analysis engine, which forms the core of a soon-to-be-released tool called Static Driver Verifier (SDV). SDV systematically analyzes the source code of Windows device drivers against a set of rules that define what it means for a device driver to properly interact with the Windows kernel, the heart of the Windows operating system (referred to as "Windows" from now on). In effect, SDV tests all possible execution paths through the C code.

To date, we have used SDV internally to find defects in Microsoft-developed device drivers, as well as in the sample device drivers that Microsoft provides in the Windows Driver Development Kit (DDK). However, the most important aspect of Window's stability is the quality of the device drivers written outside of Microsoft, called *third-party drivers*. For this reason we are now preparing SDV for release as part of the DDK.

We have written many technical research papers about SLAM but we have never before written a history of the non-technical aspects of the project. Our goal is to discuss the process of technology transfer from research to development groups and to highlight the reasons we believe that we have been successful to date, some of which are:

- *Choice of Problem*: We chose a critical, but not insurmountable, problem domain to work on (device drivers). We had access to the Windows source code and the source code of the device drivers. We also had extensive access to the foremost experts on device drivers and Windows.
- *Standing on Shoulders*: SLAM builds on decades of research in formal methods and programming languages. We are fortunate to have had many people contribute to SLAM and SDV, both in Microsoft Research, the Windows division, as well as from outside Microsoft.
- *Research Environment*: Microsoft's industrial research environment and general "hands-on/can-do" culture allowed us great freedom in which to attempt a risky solution to a big problem, and provided support when we needed it the most.
- *Software Engineering*: We developed SLAM in an "open" architectural style using very simple conceptual interfaces for each of its core components. This allowed us to experiment quickly with various tools and settle on a set of algorithms that we felt best solved the problem. This architecture also allows us to reconfigure the various components easily in response to new problems.
- *The Right Tools for the Job*: We developed SLAM using INRIA's O'Caml functional programming language. The expressiveness of this language and robustness of its implementation provided a great productivity boost.
- *Good Luck*: We experienced good luck at many points over the past four years and fortunately were able to take advantage of it.

[1] SLAM originally was an acronym but we found it too cumbersome to explain. We now prefer to think of "slamming" the bugs in a program.

While some of these factors may be unique to our situation, many are the basic ingredients of successful research, development, and technology transfer. We believe that the history of our project makes an interesting case study in the technology transfer of formal methods and software tools in industry.

We tell the story in four parts. Section 2 discusses the context in which the SLAM and SDV projects took place. In particular, this section provides background on Windows device drivers and Microsoft Research. Section 3 discusses the first two years of the SLAM project, when the bulk of the research took place. Section 4 discusses the development of the Static Driver Verifier tool and its transfer to the Windows development organization. Section 5 concludes with an analysis of the lessons we learned from our four year experience and a look at the future.

## 2 Prologue

We will now provide some pre-SLAM history so that the reader will better understand the context in which our project originated.

### 2.1 Windows Device Drivers

Windows hides from its users many details of the myriad hardware components that make up a personal computer (PC). PCs are assembled by companies who have purchased many of the PC's basic components from other companies. The power of Windows is that application programmers are still able to write programs that work using the interface provided by Windows with little to no concern for the underlying hardware that their software eventually will execute on.

Examples of devices include keyboards, mice, printers, graphics and audio cards, network interface cards, cameras, and a number of storage devices, such as CD and DVD drives. Device drivers are the software that link the component devices that constitute a PC, as well as its peripheral devices, to Windows. The number of devices and device drivers for Windows is enormous, and grows every day. While only about 500 device drivers ship on a Windows CD, data collected through Microsoft's Online Crash Analysis (OCA) tool shows orders of magnitude more device drivers deployed in the field.

Most device drivers run within the Windows kernel, where they can run most efficiently. Because they execute in the kernel, poorly written device drivers can cause the Windows kernel (and thus the entire operating system) to crash or hang. Of course, such device driver failures are perceived by the end-user as a failure of Windows, not the device driver. Driver quality is a key factor in the Windows user experience and has been a major source of concern within the company for many years.

The most fundamental interface that device drivers use to communicate with the Windows kernel is called the Windows Driver Model (WDM). As of today,

this interface includes over 800 functions providing access to various kernel facilities: memory allocation, asynchronous I/O, threads, events, locking and synchronization primitives, queues, deferred procedure calls, interrupt service routines, etc. Various classes of drivers (network drivers, for example) have their own driver models, which provide device-specific interfaces on top of the WDM to hide its complexity.

Microsoft provides the Driver Development Kit (DDK) to aid third-parties in writing device drivers. The DDK contains the Microsoft compiler for the C and C++ languages, supporting tools, documentation of the WDM and other driver models, and the full source code of many drivers that ship on the Windows CD. The DDK also contains a number of software tools specifically oriented towards testing and analyzing device drivers. One is a tool called Driver Verifier, which finds driver bugs while the drivers execute in real-time in Windows. In addition to the DDK, Microsoft has a driver certification program whose goal is to ensure that drivers digitally signed by Microsoft meet a certain quality bar. Finally, Microsoft uses the OCA feature of Windows to determine which device drivers are responsible for crashes in the field. This data is made available to Microsoft's partners to ensure that error-prone drivers are fixed as quickly as possible. Despite all these measures, drivers are a continuing source of errors. Developing drivers using a complex legacy interface such as WDM is just plain hard. (This is not just true of Windows–Engler found the error rate in Linux device drivers was much higher than for the rest of the Linux kernel [CYC+01]).

Device drivers are a great problem domain for automated analysis because they are relatively small in size (usually less that 100,000 lines of C code), and because most of the WDM usage rules are control-dominated and have little dependence on data. On the other hand, drivers use all the features of the C language and run in a very complex environment (the Windows kernel), which makes for a challenging analysis problem.

One of the most difficult aspects of doing work in formal methods is the issue of where specifications come from, and the cost of writing and maintaining them. A welcome aspect of the WDM interface, from this perspective, is that the cost of writing the specifications can be amortized by checking the same specifications over many WDM drivers. Interfaces that are widely used (such as the WDM) provide good candidates for applying formal methods, since specifications can be done at the level of the interface and all clients that use the interface can be analyzed automatically for consistent usage of the interface with respect to the specifications.

### 2.2 Microsoft Research

Over the past decade, Microsoft Research (MSR) has grown to become one of the major industrial research organizations in basic computer science, with over 600 researchers in five labs worldwide.

It is worthwhile to note the major differences between industrial research, as found in Microsoft, and research at academic institutions. First, there is no tenure in MSR, as in academia. Performance reviews take place every year, as

done in corporate America. Second, performance is measured not only by contributions to basic science (one measure of which is peer-reviewed publications) but also by contributions to Microsoft. Balancing long-term basic research with more directed work for the company is one of the most challenging but also the most rewarding aspects of industrial research. Third, working with other researchers within MSR (as well as outside) is encouraged and rewarded. Fourth, there are no graduate students. Instead, during three brief summer months each year, we are fortunate to attract high quality graduate students for internships. One final thing is worth noting: MSR generally puts high value on seeing ideas take form in software, as this is the major mechanism for demonstrating value and enabling technology transfer within Microsoft. To say this in a different way: developers are not the only Microsoft employees who program computers; researchers also spend a good deal of time creating software to test their ideas. As we discovered in SLAM, new research insights often come from trying to take an idea from theory to practice through programming.

The Programmer Productivity Research Center (PPRC) is a research and development center in MSR whose charter is "to radically improve the effectiveness of software development and the quality of Microsoft software". Founded in March of 1999, PPRC's initial focus was on performance tools but quickly grew to encompass reliability tools with the acquisition of Intrinsa and its PREfix defect detection tool [BPS00]. The PREfix technology has been deployed in many of Microsoft's product groups. More than twelve percent of the bugs fixed before Windows 2003 server shipped were found with the PREfix and PREfast tools, which are run regularly over the entire Windows source base. PPRC has developed an effective infrastructure and pipeline for developing new software tools and deploying them throughout the company.

## 3 SLAM (2000–2001)

So, the stage is set to tell the story of SLAM. Device drivers were (and still are) a key problem of concern to the company. PPRC, which supports basic research in programming languages, formal methods and software engineering, was seeking to improve development practices in Microsoft through software tools. In this section, we describe the first two years of the SLAM project.

### 3.1 Software Productivity Tools

SLAM was one of the initial projects of the Software Productivity Tools (SPT) group within PPRC, founded by Jim Larus. The members of this group were Tom Ball, Manuvir Das, Rob DeLine, Manuel Fähndrich, Jim Larus, Jakob Rehof and Sriram Rajamani. The SPT group spent its first months brainstorming new project ideas and discussing software engineering problems. The problem of device drivers was one of the topics that we often discussed.

Three projects came out of these discussions: SLAM, Vault [DF01], and ESP [DLS02]. Each of these projects had a similar goal: to rigorously check that

a program obeys "interface usage rules". The basic differences in the projects were in the way the rules were specified and in the analysis technology used. Vault was a new programming language with an extended type system in which the rules were specified using pre-/post-conditions attached to types. ESP and SLAM shared a similar specification language but took different approaches to addressing the efficiency/precision tradeoffs inherent in program analysis. (For a more detailed comparison of these three projects, see [LBD+04].)

Having several projects working in friendly competition on a common problem made each project stronger. We benefited greatly from many technical discussions with SPT members. All three projects are still active today: Manuvir now leads a group based on the ESP project to extend the scope and scale of static analysis tools; Rob and Manuel retargeted the Vault technology to MSIL (Microsoft's Intermediate Language, a byte-code like language for Microsoft's new virtual machine, the Common Language Runtime) and extended its capabilities. This analyzer is called Fugue [DF04] and is a plug-in to the Visual Studio programming environment and will be available soon as part of the freely-available FxCop tool.

### 3.2 A Productive Peer Partnership

SLAM was conceived as the result of conversations between Tom and Sriram on how symbolic execution, model checking and program analysis could be combined to solve the interface usage problem for C programs (and drivers in particular). Tom's background was in programming languages and program analysis, while Sriram's background was in hardware verification and model checking. Both had previous experience in industry. Tom worked six years as a researcher in Bell Labs (at AT&T and then Lucent Technologies) after his Ph.D. and Sriram worked over five years at Syntek and Xilinx before his Ph.D. Two months of initial discussions and brainstorming at the end of 1999 led to a technical report published in January of 2000 [BR00b] that contained the basic ideas, theory and algorithms that provided the initial foundation for the SLAM project.

Our basic idea was that checking a simple rule against a complex C program (such as a device driver) should be possible by simplifying the program to make analysis tractable. That is, we should be able to find an *abstraction* of the original C program that would have all of the behaviors of the original program (plus additional ones that did not matter when checking the rule of interest).

The basic question we then had to answer was "What form should an abstraction of a C program take?". We proposed the idea of a Boolean program, which would have the same control flow structure as the original C program but only permit the declaration of Boolean variables. These Boolean variables would track important predicates over the original program's state (such as $x < 5$).

We found Boolean programs interesting for a number of reasons. First, because the amount of storage a Boolean program can access at any point is finite, questions of reachability and termination (which are undecidable in general) are decidable for Boolean programs. Second, as Boolean programs contain the control-flow constructs of C, they form a natural target for investigating model

checking of software. Boolean programs can be thought of as an abstract representation of C programs in which the original variables are replaced by Boolean variables that represent relational observations (predicates) between the original variables. As a result, Boolean programs are useful for reasoning about properties of the original program that are expressible through such observations.

Once we fixed Boolean programs as our form of abstraction, this led us naturally to an automated process for abstraction, checking and refinement of Boolean programs in the spirit of Kurshan [Kur94]:

- **Abstract**. Given a C program $P$ and set of predicates $E$, the goal of this step is to efficiently construct a precise Boolean program abstraction of $P$ with respect to $E$. Our contribution was to extend the predicate abstraction algorithm of Graf and Saïdi [GS97] to work for programs written in common programming languages (such as C).
- **Check**. Given a Boolean program with an error state, the goal of this step is to check whether or not the error state is reachable. Our contribution was to solve this problem by using a data structure called Binary Decision Diagrams [Bry86,BCM+92] from the model checking community in the context of traditional interprocedural dataflow analysis [SP81,RHS95].
- **Refine**. If the Boolean program contains an error path and this path is a feasible execution path in the original C, then the process has found a potential error. If this path is not feasible in the C program then we wish to refine the Boolean program so as to eliminate this false error path. Our contribution was to show how to use symbolic execution and a theorem prover [DNS03] to find a set of predicates that, when injected into the Boolean program on the next iteration of the SLAM process, would eliminate the false error path.

In the initial technical report, we formalized the SLAM process and proved its soundness for a language with integer variables, procedures and procedure calls but without pointers. Through this report we had laid out a plan and a basic architecture that was to remain stable and provide a reference point as we progressed. Additionally, having this report early in the life of the project helped us greatly in recruiting interns. The three interns who started on the SLAM project in the summer of 2000 had already digested and picked apart the technical report before they arrived.

After we had written the technical report we started implementing the **Check** step in the Bebop model checker [BR00a,BR01a]. Although only one of the three steps in SLAM was implemented, it greatly helped us to explore the SLAM process as we could simulate the other two steps by hand (for small examples). Furthermore, without the **Check** step, we could not test the **Abstract** step, which we planned to implement in the summer.

During the implementation of Bebop, we often worked side-by-side as we developed code. We worked to share our knowledge about our respective fields, program languages/analysis (Tom) and model checking (Sriram). Working in this fashion, we had an initial implementation of Bebop working in about two months.

With only Bebop working, we manually extracted Boolean program models from several drivers and experimented with the entire approach. Then, over the summer of 2000, we built the first version of the **Abstract** step with the help of our interns Rupak Majumdar and Todd Millstein. After this was done, we experimented with more examples where we manually supplied predicates, but automatically ran the **Abstract** and **Check** steps. Finally, in the fall of 2000, we built the first version of the **Refine** step. Since this tool discovers predicates we named it Newton [BR02a]. We also developed a language called Slic to express interface usage rules in a C-like syntax, and integrated it with the rest of the tools [BR01b].

### 3.3 Standing on Shoulders

As we have mentioned before, the ideas that came out of the SLAM project built on and/or extended previous results in the areas of program analysis, model checking and theorem proving. A critical part to SLAM's success was not only to build on a solid research foundation but also to build on existing technology and tools, and to enlist other people to help us build and refine SLAM.

The parts of SLAM that analyze C code were built on top of existing infrastructure developed in MSR that exports an abstract syntax tree interface from the Microsoft C/C++ compiler and that performs alias analysis of C code [Das00]. The Bebop model checker uses a BDD library called CUDD developed at The University of Colorado [Som98]. (This library also has been incorporated in various checking tools used within Intel and other companies that develop and apply verification technology.) We also relied heavily on the Simplify theorem prover from the DEC/Compaq/HP Systems Research Center [DNS03]. Finally, the SLAM code base (except for the Bebop model checker) was written in the functional programming language Objective Caml (O'Caml) from INRIA [CMP]. Bebop was written in C++.

In our first summer we were fortunate to have three interns work with us on the SLAM project: Sagar Chaki from Carnegie Mellon University (CMU), Rupak Majumdar from the University of California (UC) at Berkeley and Todd Millstein from the University of Washington. Rupak and Todd worked on the first version of the predicate abstraction tool for C programs [BMMR01], while Sagar worked with us on how to reason about concurrent systems [BCR01]. After returning to Berkeley, Rupak and colleagues there started the BLAST project, which took a "lazy" approach to implementing the process we had defined in SLAM [HJMS02]. Todd continued to work with us after the summer to finish the details of performing predicate abstraction in the presence of procedures and pointers [BMR01]. Back at CMU, Sagar started the MAGIC project [CCG+03], which extended the ideas in SLAM to the domain of concurrent systems.

During these first two years, we also had the pleasure of hosting other visitors from academia. Andreas Podelski, from the Max Plank Institute, spent his sabbatical at MSR and helped us understand the SLAM process in terms of abstract interpretation [CC77]. Andreas' work greatly aided us in understanding the theoretical capabilities and limitations of the SLAM process [BPR01,

BPR02]. Stefan Schwoon, a Ph.D. candidate from the Technical University of München, visited us in the fall of 2001. Stefan had been working on a model checking tool [ES01]—called MOPED—that was similar to BEBOP. We had sent him information about Boolean programs, which allowed him to target MOPED to our format. In a few weeks of work with us, he had a version of SLAM that worked with MOPED instead of BEBOP. As a result, we could directly compare the performance of the two model checkers. This led to a fruitful exchange of ideas about how to improve both tools.

Later on, Rustan Leino joined the SPT group and wrote a new Boolean program checker (called "Dizzy") that was based on translating Boolean programs to SAT [Lei03]. This gave us two independent ways to analyze Boolean programs and uncovered even more bugs in BEBOP.

Finally, as we mentioned before, the PREfix and PREfast tools blazed the trail for static analysis at Microsoft. These two tools have substantially increased the awareness within the company of the benefits and limitations of program analysis. The success of these tools has made it much easier for us to make a case for the next generation of software tools, such as SDV.

### 3.4 Champions

A key part of technology transfer between research and development organizations is to have "champions" on each side of the fence. Our initial champions in the Windows organization were Adrian Oney, Peter Wieland and Bob Rinne.

Adrian is the developer of the Driver Verifier testing tool built into the Windows operating system (Windows 2000 and on). Adrian spent many hours with us explaining the intricacies of device drivers. He also saw the potential for Static Driver Verifier to complement the abilities of Driver Verifier, rather than viewing it as a competing tool, and communicated this potential to his colleagues and management. Peter Wieland is an expert in storage drivers and also advised us on the complexities of the driver model. If we found what we thought might be a bug using SLAM, we would send email to Adrian and Peter. They would either confirm the bug or explain why this was a false error. The latter cases helped us to refine the accuracy of our rules. Additionally, Neill Clift from the Windows Kernel team had written a document called "Common Driver Reliability Problems" from which we got many ideas for rules to check.

Having champions like these at the technical level is necessary but not sufficient. One also needs champions at the management level with budgetary power (that is, the ability to hire people) and the "big picture" view. Bob Rinne was our champion at the management level. Bob is a manager of the teams responsible for developing many of device drivers and driver tools that Microsoft ships. As we will see later, Bob's support was especially important for SLAM and SDV to be transferred to Windows.

### 3.5 The First Bug ... and Counting

In initial conversations, we asked Bob Rinne to provide us with a real bug in a real driver that we could try to discover with the SLAM engine. This would be the first test of our ideas and technology. He presented us with a bug in the floppy disk driver from the DDK that dealt with the processing of IRPs (I/O Request Packets). In Windows, requests to drivers are sent via IRPs. There are several rules that a driver must follow with regards to the management of IRPs. For instance, a driver must mark an IRP as pending (by calling `IoMarkIrpPending`) if it returns `STATUS_PENDING` as the result of calling the driver with that IRP. The floppy disk driver had one path through the code where the correlation between returning `STATUS_PENDING` and calling `IoMarkIrpPending` was missed. On March 9, 2001, just one year after we started implementing SLAM, the tool found this bug.

In the summer of 2001, we were again fortunate to have excellent interns working on the SLAM project: Satyaki Das from Stanford, Sagar Chaki (again), Robby from Kansas State University and Westley Weimer from UC Berkeley. Satyaki and Westley worked on increasing the performance of the SLAM process [ABD+02,BCDR04] and the number of device drivers to which we could successfully apply SLAM. Robby worked with Sagar on extending SLAM to reason more accurately about programs which manipulate heap data structures. Towards the end of the summer Westley and Satyaki found two previously unknown bugs in DDK sample drivers using SLAM.

Manuel Fähndrich developed a diagram of the various legal states and transitions an IRP can go through by piecing together various bits of documentation, and by reading parts of the kernel source code. Using this state diagram, we encoded a set of rules for checking IRP state management. With these rules we found five more previously unknown bugs in IRP management in various drivers.

### 3.6 Summary

In the first two years of the SLAM project we had defined a new direction for software analysis based on combining and extending results from the fields of model checking, program analysis and theorem proving, published a good number of papers (see references for a full list), created a prototype tool that found some non-trivial bugs in device drivers, and had attracted attention from the academic research community. The first two years culminated in an invited talk which we were asked to present at the Symposium on the Principles of Programming Languages in January of 2002 [BR02b].

However, as we will see, the hardest part of our job was still ahead of us. As Thomas Alva Edison noted, success is due in small part to "inspiration" and in large part to "perspiration". We had not yet begun to sweat.

## 4 Static Driver Verifier (2002-2003)

From an academic research perspective, SLAM was a successful project. But, in practice, SLAM could only be applied productively by a few experts. There

was a tremendous amount of work left to do so that SLAM could be applied automatically to large numbers of drivers. In addition to improving the basic SLAM engine, we needed to surround this engine with the framework that would make it easy to run on device drivers. The product that solved all of these problems was to be called "Static Driver Verifier" (SDV). Our vision was to make SDV a fully automatic tool. It had to contain, in addition to the SLAM engine, the following components:

- A large number of rules for the Windows Driver Model (and in future releases, other driver models as well)–we had written only a handful of rules;
- A model of the Windows kernel and other drivers, called the environment model–we had written a rough model of the environment model in C, but it needed to be refined;
- Scripts to build a driver and configure SDV with driver specific information;
- A graphical user interface (GUI) to summarize the results of running SDV and to show error traces in the source code of the driver.

SDV was not going to happen without some additional help.

Having produced promising initial results, we went to Amitabh Srivastava, director of the PPRC, and asked for his assistance. He committed to hiring a person for the short term to help us take SLAM to the next stage of life. Fortunately, we had already met just the right person for the task: Jakob Lichtenberg from the IT University of Copenhagen. We met Lichtenberg in Italy at the TACAS conference in 2001 where we presented work with our summer interns from 2000. After attending our talk, Jakob had spent the entire night re-coding one of our algorithms in a model checking framework he had developed. We were impressed. Lichtenberg joined the SLAM team in early February of 2002 and the next stage of the roller-coaster ride began. Jakob was originally hired for six months. In the end, he stayed 18 months.

### 4.1 TechFest and Bill Gates Review

The first task Lichtenberg helped us with was preparing a demonstration for an internal Microsoft event in late February of 2002 called TechFest. TechFest is an annual event put on by MSR to show what it has accomplished in the past year and to find new opportunities for technology transfer. TechFest has been an incredibly popular event. In 2001, when TechFest started, it had 3,700 attendees. In its second year, attendance jumped to 5,200. In 2003, MSR's TechFest was attended by over 7,000 Microsoft employees.

The centerpiece of TechFest is a demo floor consisting of well over 100 booths. In our booth, we showed off the results of running SLAM on drivers from the Driver Development Kit of Windows XP. Many driver developers dropped by for a demo. In some cases, the author of a driver we had found a bug in was present to confirm that we had found a real bug. Additionally, two other important people attended the demo: Jim Allchin (head of the Windows platform division) and Bill Gates.

Two weeks after TechFest (in early March 2002), we made a presentation on SLAM as part of a regular review of research by Bill Gates. At this point, managers all the way up the management chain in both MSR and Windows (with the least-common ancestor being Gates) were aware of SLAM. The rapidity with which key people in the company became aware of SLAM and started referring to it was quite overwhelming.

### 4.2 The Driver Quality Team

Around this time, a new team in Bob Rinne's organization formed to focus on issues of driver quality. Bob told us that he might be able to hire some people into this group, called the *Driver Quality Team* (DQT), to help make a product out of SDV. In the first four months of 2002, we had received a number of resumes targeted at the SLAM project. We told Bob of two promising applicants: Byron Cook, from the Oregon Graduate Institute (OGI) and Prover Technology, and Vladimir Levin, from Bell Labs. Byron was in the process of finishing his Ph.D. in Computer Science and had been working on tools for the formal verification of hardware and aircraft systems at Prover for several years. Vladimir had a Ph.D. in Computer Science and had been working on a formal verification tool at Bell Labs for six years.

By the beginning of July, both Byron and Vladimir were interviewed and hired. They would join Microsoft in August and September of 2002, respectively, as members of DQT. The importance of the Windows kernel development organization hiring two Ph.D.s with formal verification backgrounds and experience cannot be overstated. It was another major milestone in the technology transfer of SLAM. Technology transfer often requires transfer of expertise in addition to technology. Byron and Vladimir were to form the bridge between research and development that would enable SLAM to be more successful.

Nar Ganapathy was appointed as the manager of DQT. Nar is the developer and maintainer of the I/O subsystem of the Windows kernel — the piece of the kernel that drivers interact with most. This meant that half of the SDV team would now be reporting directly to the absolute expert on the behavior of the I/O subsystem.

### 4.3 SDV 1.0

Our first internal release of SDV (1.0) was slated for the end of the summer. This became the major focus of our efforts during the late spring and summer of 2002. While in previous years, summer interns had worked on parts of the SLAM engine, we felt that the analysis engine was stable enough that we should invest energy in problems of usability. Mayur Naik from Purdue University joined as a summer intern and worked on how to localize the cause of an error in an error trace produced by SLAM [BNR03].

On September 03, 2002, we made the release of SDV 1.00 on an internal website. It had the following components: the SLAM engine, a number of interface

usage rules, a model of the kernel used during analysis, a GUI and scripts to build the drivers.

### 4.4 Fall 2002: Descent into Chaos (SDV 1.1)

In the autumn of 2002, the SDV project became a joint project between MSR and Windows with the arrival of Byron and Vladimir, who had been given offices in both MSR and Windows. While we had already published many papers about SLAM, there was a large gap between the theory we published and the implementation we built. The implementation was still a prototype and was fragile. It only had been run on about twenty drivers. We had a small set of rules. Dependence on a old version of the Microsoft compiler and fundamental performance issues prevented us from running on more drivers.

When Byron and Vladimir began working with the system they quickly exposed a number of significant problems that required more research effort to solve. Byron found that certain kinds of rules made SLAM choke. Byron and Vladimir also found several of SLAM's modules to be incomplete. At the same time, a program manager named Johan Marien from Windows was assigned to our project part-time. His expectation was that we were done with the research phase of the project and ready to be subjected to the standard Windows development process. We were not ready. Additionally, we were far too optimistic about the timeframe in which we could address the various research and engineering issues needed to make the SLAM engine reliable. We were depending on a number of external components: O'Caml, the CUDD BDD package, the automatic theorem prover Simplify. Legal and administrative teams from the Windows organization struggled to figure out the implications of these external dependencies.

We learned several lessons in this transitional period. First, code reviews, code refactoring and cleanup activities provide a good way to educate others about a new code base while improving its readability and maintainability. We undertook an intensive series of meetings over a month and a half to review the SLAM code, identify problems and perform cleanup and refactoring to make the code easier to understand and modify. Both Byron and Vladimir rewrote several modules that were not well understood or buggy. Eventually, ownership of large sections of code was transferred from Tom and Sriram to Byron and Vladimir. Second, weekly group status meetings were essential to keeping us on track and aware of pressing issues. Third, it is important to correctly identify a point in a project where enough research has been done to take the prototype to product. We had not yet reached that point.

### 4.5 Winter 2002/Spring 2003: SDV Reborn (SDV 1.2)

The biggest problem in the autumn of 2002 was that a most basic element was missing from our project, as brought to our attention by Nar Ganapathy: we were lacking a clear statement of how progress and success on the SDV project would be measured. Nar helped us form a "criteria document" that we could use

to decide if SDV was ready for widespread use. The document listed the type of drivers that SDV needed to run on, specific drivers on which SDV needed to run successfully, some restrictions on driver code (initial releases of SDV were not expected to support C++), performance expectations from SDV (how much memory it should take, how much time it should take per driver and per rule), and the allowable ratio of false errors the tool could produce (one false error per four error reports).

Another problem was that we now had a project with four developers and no testers. We had a set of over 200 small regression tests for the SLAM engine itself, but we needed more tests, particularly with complete device drivers. We desperately needed better regression testing. Tom and Vladimir devoted several weeks to develop regression test scripts to address this issue. Meanwhile Byron spent several weeks convincing the Windows division to devote some testing resources to SDV. As a result of his pressure, Abdullah Ustuner joined the SDV team as a tester in February 2003.

One of the technical problems that we encountered is called NDF, an internal error message given by SLAM that stands for "no difference found". This happens when SLAM tries to eliminate a false error path but fails to do so. In this case, SLAM halts without having found a true error or a proof of correctness. A root cause of many of these NDFs was SLAM's lack of precision in handling pointer aliasing. This led us to invent novel ways to handle pointer aliasing during counter-example-driven refinement, which we implemented. SLAM also needed to be started with a more precise model of the kernel and possible aliases inside kernel data structures, so we rewrote the kernel models and harnesses to initialize key data structures. As a result of these solutions, the number of NDFs when we shipped SDV 1.2 went down dramatically. Some still remained, but the above solutions converted the NDF problem from a show-stopper to a minor inconvenience.

With regression testing in place, a clear criterion from Nar's document on what we need to do to ship SDV 1.2, and reduction of the NDF problem, we slowly recovered from the chaos that we experienced in the winter months. SDV 1.2 was released on March 31st, 2003, and it was the toughest release we all endured. It involved two organizations, two different cultures, lots of people, and very hard technical problems. We worked days, nights and weekends to make this release happen.

### 4.6 Taking Stock in the Project: Spring 2003

Our group had been hearing conflicting messages about what our strategy should be. For example, should we make SDV work well on third party drivers and release SDV as soon as possible, or should we first apply it widely on our own internally developed drivers and find the most bugs possible? Some said we should take the first option; others said the latter option was more critical. Our group also needed more resources. For example, we needed a full-time program manager who could manage the legal process and the many administrative complications involved in transferring technology between organizations. We desper-

ately needed another tester. Additionally, we needed to get a position added in the Windows division to take over from Jakob, whose stay at Microsoft was to end soon.

Worst of all, there was a question as to whether SDV had been successful or not. From our perspective, the project had been a success based on its reception by the formal verification research community and MSR management. Some people within the Windows division agreed. Other members of the Windows division did not. The vast majority of people in the Windows division were not sure and wanted someone else to tell them how they should feel.

Byron decided that it was time to present our case to the upper management of the Windows division and worked with Nar to schedule a project review with Windows vice-president Rob Short. We would show our hand and simply ask the Windows division for the go-ahead to turn SDV into a product. More importantly, a positive review from Rob would help address any lingering doubts about SDV's value within his organization.

We presented our case to Rob, Bob Rinne and about ten other invited guests on April 28th 2003. We presented statistics on the number of bugs found with SDV and the group's goals for the next release: we planned on making the next release available at the upcoming Windows Driver Development Conference (DDC), where third-party driver writers would apply SDV to their own drivers. We made the case for hiring three more people, (a program manager, another tester and developer to take over from Jakob) and buying more machines to parallelize runs of SDV. In short order, Rob gave the "thumbs-up" to all our plans. It was time to start shopping for people and machines.

### 4.7 Summer/Fall 2003: The Driver Developer Conference (SDV 1.3)

Ideally we would have quickly hired our new team-members, bought our machines and then began working on the next release. However, it takes time to find the right people, as we found out. At the end of May, John Henry joined the SDV group as our second tester. Bohus Ondrusek would eventually join the SDV team as our program manager in September. Con McGarvey later joined as a developer in late September. Jakob Lichtenberg left to return to Denmark at about the same time. By the time we had our SDV 1.3 development team put together, the Driver Developer Conference was only a month away.

Meanwhile, we had been busy working on SLAM. When it became clear that we would not know if and when our new team-members would join, we decided to address the following critical issues for the DDC event:

- More expressiveness in the SLIC rule language.
- More rules. We added more than 60 new rules that were included in the DDC distribution of SDV.
- Better modeling of the Windows kernel. While not hoping to complete our model of the kernel by the DDC, we needed to experiment with new ways to generate models. A summer intern from the University of Texas at Austin

named Fei Xie spent the summer trying a new approach in which SLAM's analysis could be used to *train* with the real Windows code and find a model that could be saved and then reused [BLX04]. Abdullah wrote a tool that converted models created by PREfix for use by SLAM.
- Better integration with the "driver build" environment used by driver writers. This included supporting libraries and the new C compiler features used by many drivers.
- Removal of our dependency on the Simplify theorem prover. SLAM uses a first-order logic theorem prover during the **Abstract** and **Refine** steps described in Section 3.2. Up until this time we had used Simplify. But the license did not allow us to release SLAM based on this prover. Again, we relied on the help of others. Shuvendu Lahiri, a graduate student from CMU with a strong background in theorem proving, joined us for the summer to help create a new theorem prover called "Zapato". We also used a SAT solver created by Lintao Zhang of MSR Silicon Valley. By the fall of 2003, we had replaced Simplify with Zapato in the SLAM engine, with identical performance and regression results. [BCLZ04]

In the end, the release of SDV 1.3 went smoothly. We released SDV 1.3 on November 5th, a week before the DDC. The DDC event was a great success. Byron gave two presentations on SDV to packed rooms. John ran two labs in which attendees could use SDV on their own drivers using powerful AMD64-based machines. Almost every attendee found at least one bug in their code. The feedback from attendees was overwhelmingly positive. In their surveys, the users pleaded with us to make a public release of SDV as soon as possible.

The interest in SDV from third-party developers caused even more excitement about SDV within Microsoft. Some of the attendees of the DDC were Microsoft employees who had never heard of SDV. After the DDC we spent several weeks working with new users within Microsoft. The feedback from the DDC attendees also helped us renew our focus on releasing SDV. Many nice features have not yet been implemented. On some drivers the performance could be made much better. But, generally speaking, the attendees convinced us (while the research in this class of tools is not yet done) that we have done enough research in order to make our first public release.

### 4.8 Summary

As of the beginning of 2004, the SDV project has fully transferred from Microsoft Research to Windows. There are now six people working full-time on SDV in Windows: Abdullah, Bohus, Byron, Con, John and Vladimir. Sriram and Tom's involvement in the project has been reduced to "consultancy"; they are no longer heavily involved in the planning or development of the SLAM/SDV technology but are continuing research that may eventually further impact SDV .

## 5 Epilogue: Lessons Learned and the Future

We have learned a number of lessons from the SLAM/SDV experience:

- *Focus on Problems not Technology.* It is easier to convince a product group to adopt a new solution to a pressing problem that they already have. It is very hard to convince a product group to adopt new technology if the link to the problem that it solves is unclear. Concretely, we do not believe that trying to transfer the SLAM engine as an analysis vehicle could ever work. However, SDV as a solution to the driver reliability problem is an easier concept to sell to a product group (We thank Jim Larus for repeatedly emphasizing the important difference between problem and solution spaces).
- *Exploit Synergies.* It was the initial conversations between Tom and Sriram that created the spark that became the SLAM project. We think it is a great idea for people to cross the boundaries of their traditional research communities to collaborate with people from other communities and to seek diversity in people and technologies when trying to solve a problem. We believe that progress in research can be accelerated by following this recipe.
- *Plan Carefully.* As mentioned before, research is a mix of a small amount inspiration and a large amount of perspiration. To get maximum leverage in any research project, one has to plan in order to be successful. In the SLAM project, we have spent long hours planning intern projects and communicating with interns long before they even showed up at MSR. We think that it is crucial not to underestimate the value of such ground work. Usually, we have had clarity on what problems interns and visitors would address even before they visit. However, our colleagues had substantial room for creativity in the approaches used to solve these problems. We think that such a balance is crucial. Most of our work with interns and visitors turned into conference papers in premier conferences.
- *Maintain Continuity and Ownership.* Interns and visitors can write code but then they leave! Someone has to maintain continuity of the research project going. We had to spend several months consolidating code written by interns after every summer, taking ownership of it, and providing continuity for the project.
- *Reflect and Assess.* In a research project that spans several years, it is important to regularly reassess the progress you are making towards your main goal. In the SLAM project we did several things that were interesting technically (for example, checking concurrency properties with counting abstractions, heap-logics, etc.) but in the end did not contribute substantially to our main goal of checking device driver rules. We reassessed and abandoned further work on such sub-projects. Deciding what to drop is very important, otherwise one would have too many things to do, and it would be hard to achieve anything.
- *Avoid the Root of All Evil.* It is important not to optimize prematurely. We believe it is best to let the problem space dictate what you will optimize. For example, we used a simple greedy heuristic in Newton to pick relevant predicates and we have not needed to change it to date! We also had the experience of implementing complicated optimizations that we thought would be beneficial but were hard to implement and were eventually abandoned because they did not produce substantial improvements.

- *Balance Theory and Practice.* In hindsight, we should have more carefully considered the interactions of pointers and procedures in the SLAM process, as this became a major source of difficulty for us later on (see Section 4.5). Our initial technical report helped us get started and get our interns going, but many difficult problems were left unsolved and unimagined because we did not think carefully about pointers and procedures.
- *Ask for Help.* One should never hesitate to ask for help, particularly if it is possible to get help. With SLAM/SDV, in retrospect, we wish we had asked for help on testing resources sooner.
- *Put Yourself in Another's Shoes.* Nothing really helped us to prepare for how the product teams operate, how they allocate resources, and how they make decisions. One person's bureaucracy is another's structure. Companies with research labs need to help researchers understand how to make use of that structure. On the other hand, researchers have to make a good faith effort to understand how product teams operate and learn about what it takes to turn a prototype into a product.

At this point, SLAM has a future as an analysis engine for SDV. Current research that we are doing addresses limitations of SLAM, such as dealing with concurrency, more accurately reasoning about data structures, and scaling the analysis via compositional techniques. We also want to question the key assumptions we made in SLAM, such as the choice of the Boolean program model. We also hope that the SLAM infrastructure will be used to solve other problems. For example, Shaz Qadeer is using SLAM to find races in multi-threaded programs.

Beyond SLAM and SDV, we predict that in the next five years we will see partial specifications and associated checking tools widely used within the software industry. These tools and methodologies eventually will be integrated with widely used programming languages and environments. Additionally, for critical software domains, companies will invest in software modeling and verification teams to ensure that software meets a high reliability bar.

**Acknowledgements.** We wish to thank everyone mentioned in this paper for their efforts on the SLAM and SDV projects, and to the many unnamed researchers and developers whose work we built on.

## References

[ABD+02] S. Adams, T. Ball, M. Das, S. Lerner, S. K. Rajamani, M. Seigle, and W. Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *SAS 02: Static Analysis Symposium*, LNCS 2477, pages 230–246. Springer-Verlag, 2002.

[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*, To appear in LNCS. Springer-Verlag, 2004.

[BCLZ04] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. *Under review*, 2004.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BCR01] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031. Springer-Verlag, 2001.

[BLX04] T. Ball, V. Levin, and F. Xei. Automatic creation of environment models via training. In *TACAS 04: Tools and Algorithms for the Construction and Analysis of Systems*, To appear in LNCS. Springer-Verlag, 2004.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.

[BMR01] T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, 2001.

[BNR03] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL 03: Principles of programming languages*, pages 97–105. ACM, 2003.

[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*, LNCS 2031, pages 268–283. Springer-Verlag, 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*, LNCS 2280, pages 158–172. Springer-Verlag, April 2002.

[BPS00] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.

[BR00a] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

[BR00b] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, January 2000.

[BR01a] T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE 01: Workshop on Program Analysis for Software Tools and Engineering*, pages 97–103. ACM, 2001.

[BR01b] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[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. ACM, January 2002.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[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. ACM, 1977.

[CCG+03] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *ICSE 03: International Conference on Software Engineering*, pages 385–395. ACM, 2003.

[CMP] E. Chailloux, P. Manoury, and B. Pagano. *Dévelopment d'Applications Avec Objective CAML*. O'Reilly (Paris).

[CYC+01] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP 01: Symposium on Operating System Principles*, pages 73–88. ACM, 2001.

[Das00] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.

[DF01] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation*, pages 59–69. ACM, 2001.

[DF04] R. DeLine and M. Fähndrich. The Fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2004.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, June 2002.

[DNS03] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[ES01] J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *CAV 01: Computer Aided Verification*, LNCS 2102, pages 324–336. Springer-Verlag, 2001.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM, January 2002.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[LBD+04] J. R. Larus, T. Ball, M. Das, Rob DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software (to appear)*, 2004.

[Lei03] K. R. M. Leino. A sat characterization of boolean-program correctness. In *SPIN 03: SPIN Workshop*, LNCS 2648, pages 104–120. Springer-Verlag, 2003.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

[Som98] F. Somenzi. Colorado university decision diagram package. Technical Report available from `ftp://vlsi.colorado.edu/pub`, University of Colorado, Boulder, 1998.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.