An SDVRP Platform Verification Method for Microprocessor-Based Systems Software

Sergey Shershakov
Software Engineering School
National Research University Higher School of Economics
Moscow, Russia
Email: sashershakov@edu.hse.ru

Scientific Advisor: Prof. Irina Lomazova
Software Engineering School
National Research University Higher School of Economics
Moscow, Russia
Email: ilomazova@hse.ru

***Abstract*—The correctness of embedded systems software is of critical importance as invalid states can cause a physical damage to hardware. One of approaches to verification of such systems is using source code analyzers. The Static Driver Verifier Research Platform (SDVRP), which is based on Simultaneous Localization and Mapping (SLAM) and represents a tool that systematically analyzes source code and allows writing custom Specification Language for Interface Checking (SLIC) rules for various platforms, provided a potent verification mechanism for a thermal printer software system based on ARM Cortex-M0 microprocessor. An example of creating a custom platform plugin and rule verification is provided for the given embedded system.**

***Keywords:* Static Verification, SLAM, SDVRP, SLIC, Embedded Systems, ARM Microprocessor, Thermal Printer**

I. Introduction

In the early 2000s Microsoft seriously attended to issues of Windows drivers reliability. Thomas Ball and Sriram K. Rajamani, members of the Software Productivity Tools (SPT) research group of the Programmer Productivity Research Center (PPRC) at Microsoft Research, initiated a project called Simultaneous Localization and Mapping (SLAM) [1], the goal of which was to implement a rigorous performance test that some program is subject to the use rules of interface with which it interacts.

SLAM was based on the idea of using symbolic execution, model checking, and program analysis for proper interface use of programs in C language, which were represented by driver modules for Windows.

SLAM was the basis for the Static Driver Verifier (SDV), which is a compile-time static verification tool included in the Windows Driver Kit (WDK). The main purpose of the SDV is automated static testing of Windows drivers developed with the assistance of WDK tools. The SDV uses the "platform" ideology to determine how a driver must interact with a system (platform), which input-output interfaces to use and which methods of interaction with it to provide. To verify the interaction between a driver and a specific platform, a set of rules in the specialized SLIC language is developed.

The SDVRP (SDV Research Platform) is an extension of the SDV, the main difference with which is the ability to develop custom platforms and test drivers for them. The basis of the idea presented in this paper is speculation about the possibility of using SDVRP tools in an unusual way: for static verification of C programs for ARM microcontrollers (MCUs), wherein the MCU environment library is proposed to be used as a custom platform and a software module that implements a required functionality of MCU firmware is used as a test "driver".

The rest of this paper is organized as follows. Section II discusses the context in which the SLAM project took place. Section III discusses the Static Driver Verifier (SDV) tool and the SDV Research Platform (SDVRP) which is an extension to the SDV that allows adapting the SDV to support additional platforms for verification. Section IV is devoted to the research on application of the SDVRP for verifying MCU software (firmware) by the example of an embedded thermal printing system based on the ARM Cortex-M0 thermal microprocessor architecture. Practical application of the SDVRP for testing a thermal printing complex is discussed in Section V. Finally, Section VI concludes with an analysis of the work done and a look at the future.

II. SLAM

The main idea of SLAM, that is checking a simple rule for a complex program written in C (e.g., a driver), has to be realized by streamlining the program in order to make possible a comprehensive analysis. In other words, one needs a mechanism to obtain an *abstraction model* of the program which preserves the behavior of the original program in C.

The main question that arises at this point: „What form should take an abstraction model of a program in C?" An approach based on *Boolean programs* was proposed.

A. Boolean Programs and "Bebop" checker

While shaping the idea of *Boolean programs* the developers were guided by the following characteristics, which a model checking tool should have [2]:

- for the modeled program there must be a representation of R analogous to a finite state machine (FSM)[1];
- a model checking algorithm in R should represent the shortest path to the model error, if any is found;
- there should be developed a mechanism of translating programs in high-level programming languages such as C, C++, Java to a model in R (abstracting);

[1]At that time the theory of finite state models verification was well-developed

- a specific model r in R can be refined to a model $r\prime$ in R and proved correct;
- the algorithms for model validation in R should support modularity and abstract constraints existing in the original program.

As a result, there has been developed an ideology of Boolean programs that have control structures common in imperative languages (such as C) and in which the only data type of all variables is Boolean. The Boolean programs contain procedures with parameters called by value, recursion, and control nondeterminism in a restricted form [3].

The Boolean programs are of interest for the following reasons.

1) The amount of memory (number of data cells), which the Boolean program operates, is finite (and even more so due to the limitation of the actual computing power), so the problem of accessibility and termination, that does not have a general solution, is soluble for Boolean programs (it is noted that the Boolean programs are equivalent in power to push-down memory automata generating context-free grammar).
2) Once the control structures of a Boolean program are similar to those of programs written in C, the Boolean programs are obvious candidates for testing programs with similar structures.
3) A Boolean program can be regarded as an abstract representation of a C program with available explicit correspondence between the data and control, and Boolean variables may represent some predicates on a generally unrestricted state of the C program.

To test Boolean program models a special model checker *Bebop* was developed. Formally, given a Boolean program B and an expression s in B, Bebop determines whether s is accessible in B. In other words, s is accessible in B if there exists an initial state of the program (as defined by its Boolean variables) such that starting execution of the program from this state, s will eventually be fulfilled. If the expression s is accessible, the shortest route to s is built.

B. Automatic Predicate Abstraction of C Programs

As noted earlier, the Boolean programs in their control structures are similar to programs in C. One of the main criteria considered in the development of Boolean program principles was the possibility of getting a model of a real C program in the form of its Boolean equivalent.

One of the most promising approaches which allows getting a Boolean model out of an original C program automatically was the principle of *predicate abstraction*, which consists in the following: the specific states of the system (source program) are mapped onto an abstract state in accordance with their assessment on a finite set of predicates.

In the continuation of the SLAM project there was developed a tool called *C2BP*, which performs such an operation [4]. Given a C program P, a set of predicates E, which are usual expressions in C without calling functions, *C2BP* automatically creates a Boolean program $BP(P, E)$, which is an abstract model of the program P. The Boolean program has the same control structure as the original program P but contains only $|E|$-Boolean variables, each of which has its own representation in the set of predicates E.

For example, suppose there is a predicate $(x < y)$ in a set E, where x and y are integer variables of a program P. Then there is a Boolean variable in a corresponding program $BP(P, E)$ such that if it is true at some point p of the Boolean program, it means that at this very point p of the original program P the predicate $(x < y)$ is estimated the true way.

For each expression s of the program P *C2BP* automatically creates a corresponding Boolean transfer function, which displays the effect of expression s on the predicates of E. The resulting Boolean program is to be evaluated by Bebop utility considered earlier.

III. Static Driver Verifier (SDV) and Static Driver Verifier Research Platform (SDVRP)

The SLAM analyzing mechanism became a core part of a new tool *Static Driver Verifier* (SDV), the main purpose of which is a systematic analysis of the source code for Windows drivers with the view of compliance with the set of rules defining how the drivers have to interact with the OS kernel [1].

A. Interface Temporal Safety Properties and Their Validation

An interface has a number of temporal safety properties which have to be respected by a program that uses the former; checking this is among the goals set by SDVRP developers. Safety means here that nothing abnormal occurs to the program. For instance, safety for a lock may signify that it has to be released only when it has been acquired [5]. Once a certain program has been attributed a safety property to comply with, it makes sense to verify the code does respect it, and if it does not, then to locate an execution path demonstrating how the program would break it.

The user does not have to supply abstractions or annotations (invariants) in order to validate or invalidate system software safety properties with the help of model checking. This implies that model checking is used for automatic computing of fixpoints over a C code abstraction expressed by a Boolean program. One constructs a proper abstraction in two steps: first, one gets an initial abstraction from the property that has to be validated, and then an automatic refinement algorithm is applied to refine the abstraction.

B. Specification Language for Interface Checking (SLIC)

There is a number of ways of how reliability of an APIs-based software system can be hindered: an API may not be correctly executed by an implementation L or the API may be treated wrongly by a client P [6]. Usually, only the API's documentation contains a set of requirements imposed on the client as well as on the implementer.

For specifying temporal safety properties of C language APIs a special low-level specification language *SLIC* was introduced to the SLAM project.

Suppose a state machine defined by a SLIC specification S monitors the behavior of a sequential composition $P||L$ of two programs P and L which happens at the API's procedural interface. This finite state automaton discards definite sequences of interface states of $P||L$ if the API is not properly implemented by L or P uses incorrectly the L-implemented API.

A SLIC *product construction* $Q\prime$ has an important property: it is such a product of a program $Q = P||L$ combined with the specification S that if and only if Q satisfies the specification S, a unique label (`SLIC_ERROR`) is not reachable in $Q\prime$.

C. Static Driver Verifier (SDV)

The SLAM tools enable verifying system software temporal safety properties completely automatically. Breakdowns are listed by the SLAM tools as paths over the program P. It automatically refines the abstraction by using the spurious error paths found.

The Static Driver Verifier (SDV), a tool that systematically analyzes the source code of Windows device drivers for compliance with the rules defining what it is to properly interact with the Windows operating system kernel, is essentially based on the SLAM analysis engine [1].

It contains, besides the SLAM engine, other constituents:

- a model of the Windows kernel and other drivers, called the *Environment Model* (Fig. 1);
- a large number of rules for the Windows Driver Model;
- scripts to build a driver and configure the SDV with driver specific information;
- a graphical user interface (GUI) to summarize the results of running the SDV and to show error traces in the source code of the driver.

The most important component of the SDV is the *Platform Model*, which represents an abstraction of the Platform.

The Platform Model includes three components (Fig. 2):

- a Platform Manager Model responsible for exercising the module by calling the module's entry points. It is in a way the entry point (main routine) of the system. For WDM drivers this is the Windows IO Manager;
- a set of Platform API Models that the module can use for completing requests from the Platform Manager. The APIs normally contain functionality to access the underlying hardware. In Windows this is the Windows Device Driver Interfaces (DDIs), consisting of hardware abstraction layer, APIs for controlling execution (locks, queues, etc.) and APIs for accessing resources such as memory;
- a Platform Model Infrastructure, which contains shared header files, common functions and states for the Platform Manager Model and the Platform API Models.

D. Static Driver Verifier Research Platform (SDVRP)

The SDV Research Platform (SDVRP) is an extension to the SDV that allows adapting the SDV to support additional platforms and writing custom SLIC rules for this platform [7]. Typically, driver platforms are the platforms one would adapt the SDV to verify, but it can also be any other module embedded to an OS or an application. If additional settings are made, one can use the SDV to check that API or protocol clients comply with the protocol/API specification.

IV. Background of using the SDVRP for verification of MCU software

A methodology for applying the SDVRP to verify embedded software systems based on an ARM Cortex-M0 microprocessor is hereinafter under consideration. This system is part of a hardware-software complex which performs the function of printing on heat-sensitive tape (thermal printing).

The correctness of this software is of particular importance in the sense that there are states of the program which can cause physical damage to hardware; exposing the program to such conditions is therefore unacceptable. For example, an incorrect sequence of pulses in the windings of the stepping motor can lead to mechanical jamming of the shaft, and an incorrect circuitry of the thermocouples can make them burn out and even make the print substrate go up in flames, which can lead to material damage and personal injury.

In light of these problems, the microprocessor firmware verification is important. This paper reflects the results of research on the use of SDVRP tools (positioned by the developers primarily as a tool for Windows drivers troubleshooting) for static verification of microprocessor firmware.

The key features allowing the use of the SDVRP for these purposes are considered as follows.

A. Source code language

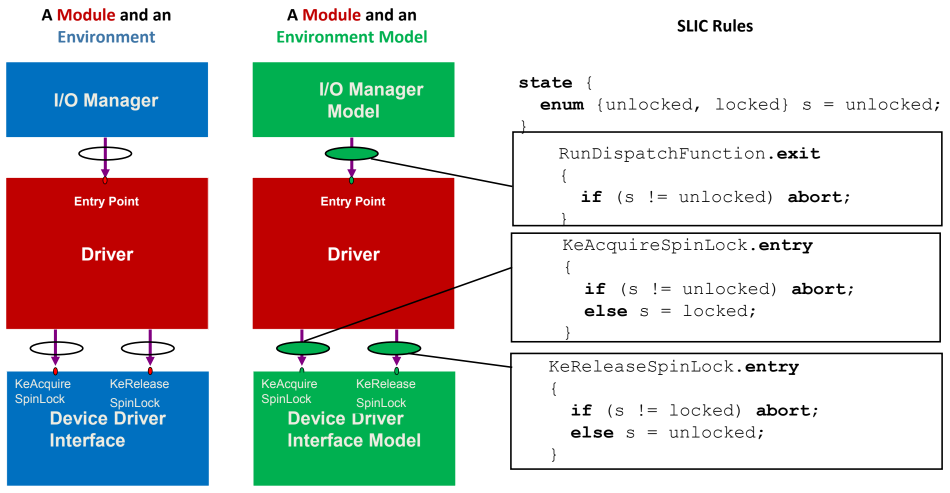
The ARMs are RISC-microprocessors[2]. The MCU's instruction set is implemented in ARM assembler, which is the main programming language. At the same time, there are high-performance optimizing compilers for high-level languages C and C++, and the share of MCU programs developed in these languages increases every day, which is primarily due to an increase in MCU hardware resources.

The best is to use a C language compiler, which allows obtaining an equally efficient code, as if it were written in assembler. This allows, on the one hand, to use it in time-critical applications (which include the majority of embedded applications, including those for ARM MCUs), on the other hand, to use the entire power and convenience of a high-level programming language.

As it was shown in section II, SLAM, which underlies the SDVRP, allows automatically receiving predicate abstractions exactly for programs written in C.

It should be noted that the SDV imposes some restrictions on the code of the programs written in C which do not affect the expressive properties of the language but eliminate various tricky ambiguities. In C programs for microprocessor-based systems (in particular, those for C-to-ARM assembler compiler) an ANSI C superset with some additional constructs is used. Basically these are additional type identifiers and

[2]Restricted (Reduced) Instruction Set Computer



```
state {
  enum {unlocked, locked} s = unlocked;
}

RunDispatchFunction.exit
{
  if (s != unlocked) abort;
}

KeAcquireSpinLock.entry
{
  if (s != unlocked) abort;
  else s = locked;
}

KeReleaseSpinLock.entry
{
  if (s != locked) abort;
  else s = unlocked;
}
```

Figure 1: A Module and an Environment Which Are Substituted by an Environment Model and SLIC Rules

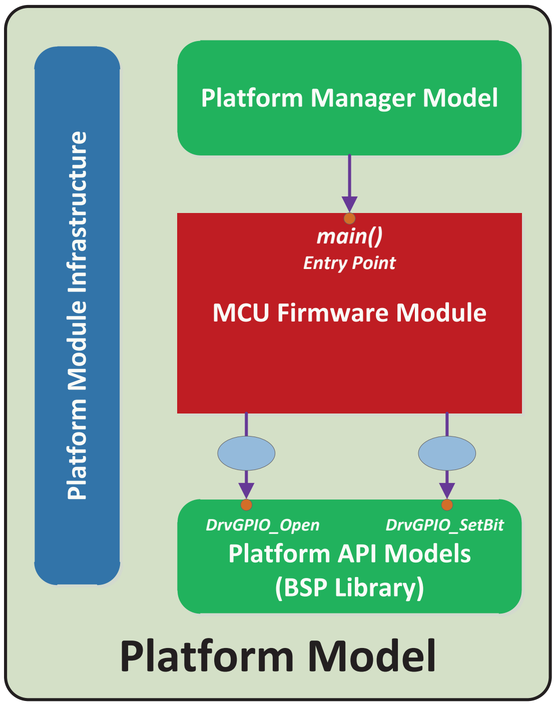


Figure 2: A Module and the Components of a Platform Model

keywords that govern the choice of memory to store the data. In order for the SDV to treat such input programs, the following workaround is proposed: to express such constructions in lexically similar directives of the C preprocessor, which are processed in due course while being compiled but ignored as irrelevant during verification.

The undoubted advantage is also the fact that the software library (the BSP library) for support of MP NUC140 (the ARM CMSIS for the kernel and Nuvoton Platform for the peripherals blocks) comes with the source code in two versions: in assembler and in C. The BSP library (refer to

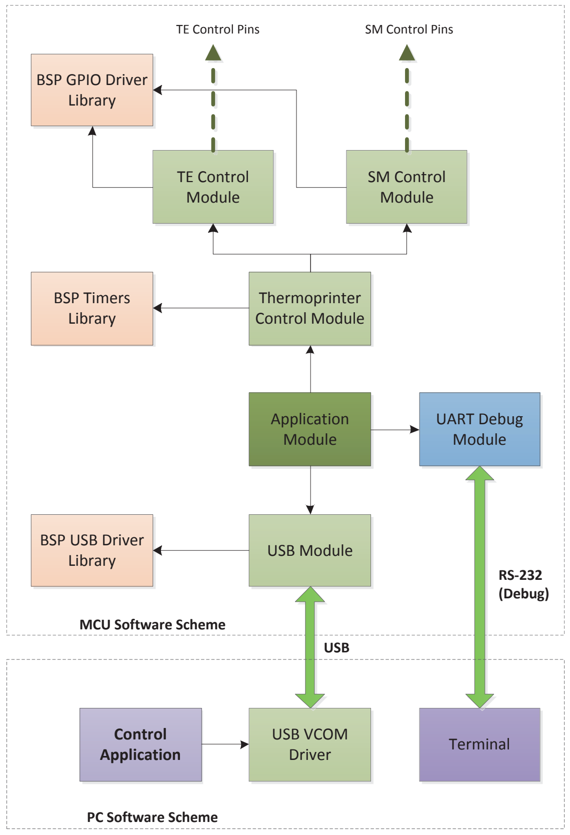


Figure 3: Thermal Printer Software Chart

Fig. 2) corresponds to the Device Driver Interface Model in the diagram Fig. 1.

B. Modular structure of the program

Each of the units of the thermal head constituting the thermal printer has its own specific control which defines the rules for its programming: structures used, initialization algorithms, etc (refer to SectionV). Each such unit has its own programming model expressed as a *function module*, which can be represented for clarity's sake as a separate translation unit — source file .c (and a corresponding header file .h).

The SDV verifies that a *module* interacts correctly with a *platform*. The platform is essentially a set of APIs, or a library. In this case the BSP library serves as a platform architecture and an ARM microprocessor.

The functional purpose and specificity of control of each unit is also a good source for developing SLIC rules which describe its specification. Formulation of SLIC rules that are specific for each module (and corresponding units) belongs is one of the main current tasks.

C. A look at the system as a whole

Another source for SLIC rules are BSP library program modules used jointly by several units/modules of programs to verify. An example of such a module is General Purpose Input-Output (GPIO) — routines to control the status of input-output pins for general purpose.

In general, access to shared resources is quite common while developing microprocessor-based software, which provides an opportunity for research on formulation of rules that control the correctness of such operations. The sequence of test calls is injected into the module under verification by an element called *harness*.

Positioned as an important property of the SDV is the fact that verification (while executing preliminary procedures such as writing SLIC rules, designing harness, etc.) is carried out automatically when building a driver project. Similarly, it is possible to use the SDVRP as an external batch process that is run by an IDE (e.g., Keil or IAR) while building the project. In addition, for translation of programs one can use compiler tools of these IDEs in conjunction with Eclipse, known for its scalable modular structure. This allows developing a special plug-in that performs verification on the basis of SDVRP modules controlled from Eclipse (this is a subject of future work).

Futher, a practical application of the SDVRP for testing a thermal complex is considered.

V. Device Software Verification: Study Case

The hardware-software complex considered previously (Fig. 3) includes:

- a Fujitsu-Siemens FTP-628MCL054 *thermal head*, which is a complete plug-in thermal printer module with physical layer interfaces to control of a stepping motor (SM), which feeds print substrate, and of low-inertia heating elements (TE) engaged in short-cycle heating-cooling in the printing zone (refer to the specification [8]);
- an *interface card* — a printed circuit board which was specially designed for this project and carries out conversion of control signal interfaces to physical interfaces of SM and TE control; the card also contains the necessary circuitry to support the functions of a comparator for paper detector and an ADC for the thermal sensor;
- a NU-LB_002 *debug board* — an evaluation board for developers of systems based on Nuvoton NUC140 microcontrollers, which provides physical access to the microcontroller's GPIO, CMP, ADC interfaces used to control the thermal head through the interface card;
- a *personal computer* (PC) which connects to the debug board via USB interface; another (special) interface is used for programming the microcontroller's USB ICE; debugging information from the MCU connected to the debug board is obtained via RS-232 interface.

The thermal head consists of the following functional units (refer to [8]):

- stepping motor;
- thermal head;
- thermal head temperature detector (thermistor);
- paper detector – mark detector (photo interrupter);
- platen release (platen open switch).

These units are controlled independently from each other through the formation of a sequence of analog-digital signals and analysis of responses received. In most cases, to perform functionally complete operations of the head (such as printing, feeding tape, etc.) coordinated participation of several units (e.g., the thermocouples and the stepping motor) is required. Control, coordination and analysis of the current state of the thermal head and the supporting elements (parts of the interface card) is carried out using the software (firmware) recorded in volatile memory of the MCU (flash memory).

A following problem of MC software correctness is considered below. Nu140-family MCU has a large number of pins, which can be used by the developer to control a device based on this MCU. These include the so-called GPIO pins (grouped in a few ports), which have various types of circuit implementation, depending on the task. Modern circuitry allows selecting an operation mode of each pin at the software level, by setting a value of the corresponding configuration register. In some tasks it is critical to correctly set an operation mode of a pin *before* it is first used, as otherwise the pin port and the net attached to it may be damaged. This way the *correctness of MCU software* implies a code that is guaranteed to correctly initialize a pin before its first use.

Below is shown how one can validate MCU software with the help of the SDVRP.

A. Custom Platform Plugin

Files that belong to a certain platform can be organized into a "plugin" if the SDV and SDVRP are implemented. A plugin for a custom program should be created in order to apply the SDVRP to that platform.

Let the test plugin be called *XiPlatform1*. Physically, three subdirectories in the SDVRP directory correspond to it; the path to the SDVRP directory is given by the environment variable `%SDV%`. The directory `%SDV%\data\XiPlatform1` is for configuration files, `%SDV%\rules\XiPlatform1` — for rules files and `%SDV%\osmodel\XiPlatform1` — for the harness and other auxilary files.

Platform API Models on the diagram in Fig. 2 is a software environment which is called by the verified module. With respect to the system under consideration such a program environment is the BSP Library supplied by the MCU manufacturer and ARM core licensee. Finally, the verified MCU software itself corresponds to the MCU Firmware Module element.

B. Rule verification

Pin initialization is done by calling a BSP Library function `DrvGPIO_Open` with corresponding parameters. In this example, `ENA` pin enabling the SM driver microchip, the state of which is subsequently changed in a certain sequence by calling the functions `DrvGPIO_SetBit` and `DrvGPIO_ClrBit`, is initialized.

A rule that checks that `ENA` pin is initialized before its first use looks as follows:

```
#include <XiGPIOUse_slic.h>
state{
    enum {closed, opened} enpinout = closed;
}
DrvGPIO_Open.entry
{  if($1 == TSM_PORT_EN && $2 == TSM_PIN_EN)
      enpinout = opened;
}
DrvGPIO_ClrBit.entry
{  if(enpinout == closed)
      abort "The driver is calling $fname
               before the pin is opened.";
}
DrvGPIO_SetBit.entry
{  if(enpinout == closed)
      abort "The driver is calling $fname
               before the pin is opened.";
}
```

The algorithm that calls pin initialization and control methods is either a separate algorithm or a part of another algorithm. The call of this algorithm is carried out in a routine which is the entry point of the module and corresponds to the function `main` of the C program. This routine is part of the Platform Manager Model harness and described in a special way.

VI. Conclusion

In this paper there was given a review of SLAM tools and the potential of SDV tools based on the former, as well as that of SDVRP version ones. There was revealed a possibility of using SDVRP tools for static verification of embedded microcontroller software system that use source codes in C language by the example of a control system of thermal printing based on a NU140 microprocessor with the ARM Cortex-M0 core. Among the tasks of current importance there are the completion of a SDVRP plug-in which implements the Environment Model for NU140 and the BSP library, as well as the development of the necessary SLIC rules for modules of the embedded system and their verification on the SDVRP platform.

The considered study case (creating a custom platform plugin and rule verification) is part of a larger verification system and presented in a nutshell, due to limitations on paper length which do not allow more scrutiny.

Acknowledgment

The author would like to thank Prof. Irina A. Lomazova for her vital encouragement and support.

References

[1] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, “SLAM and static driver verifier: Technology transfer of formal methods inside microsoft,” *IFM*, pp. 1–20, 2004.

[2] T. Ball and S. K. Rajamani, “Boolean programs: A model and process for software analysis,” Microsoft Research, Tech. Rep., 2000.

[3] ——, “Bebop: A symbolic model checker for boolean programs,” *SPIN 00: SPIN Workshop*, pp. 113–130, 2000.

[4] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, “Automatic predicate abstraction of C programs,” *SIGPLAN Not.*, vol. 36, no. 5, pp. 203–213, may 2001.

[5] T. Ball and S. K. Rajamani, “Automatically validating temporal safety properties of interfaces,” pp. 103–122, 2001.

[6] ——, “SLIC: a specification language for interface checking (of C),” Software Productivity Tools, Microsoft Research, Tech. Rep., 2002.

[7] *Static Driver Verifier Research Platform. Introduction (sdvrp.docx)*.

[8] *Fujitsu Takamisawa Component Ltd. Thermal Printer FTP-628MCL054. Product Specification*, 2000.