A System for Synthesizing Abstraction-Enabled Simulators for Binary Code Verification

Dominique Gückel, Jörg Brauer, and Stefan Kowalewski
Embedded Software Laboratory, RWTH Aachen University
Ahornstraße 55, 52074 Aachen, Germany
Email: {lastname}@embedded.rwth-aachen.de

Abstract—**Formal verification of embedded software is crucial in safety-critical applications, ideally requiring as little human intervention as possible. Binary code model checking based on hardware simulators already comes close to this goal, although with high initial effort for developing a simulator of the respective target platform. In the embedded systems domain with its varieties of different architectures in use, this can severely restrict the applicability of this approach. To remedy this drawback, we describe a system for automatically synthesizing simulators, which are suited for model checking in that they support automatic abstraction. We evaluate the practicality of this approach by synthesizing simulators for the Atmel ATmega16 and Intel MCS-51 microcontrollers.**

I. Introduction

The application of formal methods to verify – or at least increase confidence in – the correctness of safety-critical embedded systems is becoming increasingly important. A wide variety of formal methods, including but not limited to abstract interpretation [1], concolic testing [2], model checking [3], [4], and theorem proving [5], have successfully been applied to embedded systems on different levels of abstraction, from the verification of conceptual models down to the level of binary code.

A. Focus

The focus of our work is model checking of binary code, which has several distinct advantages (and some disadvantages) compared to the verification of high-level models. Firstly, the approach is tied to a specific hardware model, which is particularly acute when working with microcontrollers, where errors are often introduced through misuse of certain hardware features, for instance, messing with interrupt-priority levels. The effects of such misuse cannot be evaluated properly without a detailed model of the underlying hardware platform. Secondly, it is capable of finding bugs that are not visible in source code due to a lack of knowledge how some high-level language constructs are actually compiled. Further, in binary code, each instruction has a well-defined semantics, which contrasts with the many ambiguities present in the C programming language, to name only one example. Thirdly, and maybe most importantly, formal methods based on binary code do not have to rely on correctness of compilers and linkers, a problem that is particularly acute in embedded systems software [6].

On the other hand, however, there are two (correlated) drawbacks. Since the verification process is hardware-specific, and the hardware configurations have a strong influence on the behavior of the microcontroller, the state-explosion problem tends to be worse compared to model checking at higher degrees of abstraction. Further, hardware simulators are required for building the state spaces of the programs, and these have to be adapted for every target platform. Moreover, due to state explosion, existing simulators as delivered by hardware vendors cannot be reused, since they fail for verifying even the smallest programs. Instead, applicable simulators in the context of model checking need to automatically apply abstraction, ideally requiring as few human intervention as possible. On the other hand, abstraction techniques need to employ knowledge about the hardware platform in order to be effective. Hence, there is usually interplay with the degree of automation and effectiveness.

In the past, such simulators have been designed and written by hand, requiring at least 6 man-months from our experiences. The effectively required time frame, however, strongly depends on the complexity of the target microcontroller, and the intricacy of the integrated abstraction techniques.

B. Approach

To remedy this significant workload required for supporting a new microcontroller, we propose to automatically synthesize simulators for model checking from hardware descriptions. The target platform – including memory layout, instruction set, and peripherals – is specified in an architecture description language (ADL), which serves as the input for a synthesis system that automatically generates a compilable simulator. Such specifications of the target platform can easily be derived from the documentation delivered by hardware vendors, for instance, from the instruction-set specification. A synthesized simulator is then automatically integrated into [MC]SQUARE[1], a binary code verification platform that serves as the basis for our efforts. Further, additional tools such as disassemblers are automatically derived. Since the state-explosion problem is particularly severe for binary code model checking, the synthesis system (semi-) automatically integrates automatic abstractions, and no further human intervention is needed.

[1]http://mcsquare.embedded.rwth-aachen.de

C. Contributions

In this paper, we make the following contributions:

- We describe an ADL called *State-space Generator Description Language* (SGDL), which is an extension of the language ISILDUR used in the AVRORA project[2].
- We detail the synthesis system that is needed to translate specifications in SGDL into executable simulators.
- To tackle the state-explosion problem, we detail the integration of automatic abstraction techniques at the example of the so-called *lazy stack evaluation*, which exploits the way the stack is used in binary programs. We also present the integration of another abstraction, called *path reduction*, which reduces paths in the state space to single transitions in case they satisfy certain conditions.
- We evaluate the effectiveness of the approach by comparing the effort required for hand-written and synthesized simulators as well as the sizes of state spaces generated by both types of simulators.
- We show the flexibility of the system by focussing on the synthesis of simulators for programs targeting the Atmel ATmega16 and the Intel MCS-51 microcontrollers.

II. Related work

The description of hardware, especially of processor-based architectures, is important in a wide variety of applications. VHDL and Verilog are the most prominent general-purpose hardware description languages, in that they are extremely flexible and can be synthesized to actual hardware. However, in providing great flexibility they also provide little restriction on the design, and thus, little support for solving specific tasks such as describing processors and microcontrollers. For this purpose, more specialized *architecture description languages (ADLs)* have been developed. These languages and associated tool chains are especially used in design space exploration, which aims at creating application-specific controllers and digital signal processors. Examples are EXPRESSION [7], MADL [8], and LISA / LISAtek [9]. Developers can create descriptions of their hardware in these languages, and the tool chain automatically generates software tools such as assemblers, C compilers, cycle- or phase-accurate simulators, profilers, etc., for devices that are still under construction.

However, none of these languages is suitable for retargeting a software model checker such as [MC]SQUARE, which does not aim at verifying any hardware, but software intended to run *on* the hardware. The problem with VHDL is that it covers far too many details about the hardware, which have to be stored in each state in the state space. Considering that a state space for a typical program consists of millions of states, each state must not exceed more than a few kB of size. This is the case for instruction-accurate simulation, but not for cycle-accurate or even phase-accurate simulation. Hence, ADLs developed so far are also infeasible for this purpose, as these kinds of hardware simulation are exactly what they were designed for.

[2]http://compilers.cs.ucla.edu/avrora/

Apart from general-purpose model checkers such as SMV / NuSMV [10], which require developers to create their own model, there are several model checkers which are suitable for verifying software. These tools interpret instructions in order to create the system model. However, most of these aim at high-level languages (HLL) such as C. When compared to assembly level model checkers such as [MC]SQUARE, this certainly has the advantage that state spaces tend to be smaller. There are disadvantages, though, that severely limit the use of HLL approaches, especially when verifying embedded software. A detailed discussion of this issue is given in [11]. An example of a HLL model checker is STEAM [12], which aims at C programs.

III. [MC]SQUARE

[MC]SQUARE is an explicit-state model checker for microcontroller binary code, which operates on disassembled binary programs. It currently supports the Atmel ATmega16 and ATmega128, Infineon XC167, Intel MCS-51, and Renesas R8C\23 microcontrollers. Additionally, it supports programs for Programmable Logic Controllers (PLCs) written in Instruction List (IL) and abstract state machines. Formulas are given in Computation Tree Logic (CTL) and can contain propositions about general-purpose registers, I/O registers, and the main memory.

Figure 1 shows the model checking process applied in [MC]SQUARE, which is the same for all supported hardware platforms. The core component is a simulator, which is responsible for building successor states by means of interpretation. A simulator is needed for each supported hardware platform. However, these are not regular simulators as typically provided by hardware vendors, but special ones that can handle nondeterminism and support abstraction. Nondeterminism is introduced into the system using peripherals such as timers, since [MC]SQUARE abstracts from time and resorts to nondeterminism whenever time is involved, as well as interrupt handlers. Further, several abstraction techniques are implemented in these simulators to reduce state spaces, whilst at the same time generating over-approximations of the behavior shown by the real hardware.

In the model checking process, the program file and optional files such as C and debug files are loaded by the program parser first, and then, the CTL formula is processed. The optional files are used to map information between the assembly and the C code when model checking binary code. Thus, one can use variables from the C code in the specification, which are then automatically mapped to the corresponding memory locations. After loading the files and the formula, the static analyzer conducts its analyses and annotates the program. These annotations serve as the basis for several abstraction techniques used in the simulator to reduce state spaces.

The actual model checking works as follows. First, the model checker obtains the initial state from the state space and checks certain subformulas (because it uses on-the-fly model checking). Then, depending on the formula and the result, it requests successors of the current state from the state space.

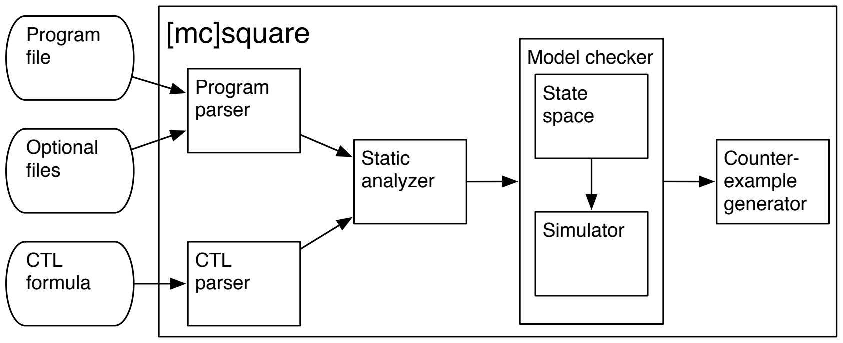

Figure 1. Model checking process in [MC]SQUARE

If the successors of the current state are not created yet, the state space uses the simulator to create them on-the-fly. The generation of successor states by the simulator is performed using the following steps:

1) Load state into model of hardware
2) Determine assignments needed for resolving nondeterminism
3) For each assignment
    a) If assignment indicates occurrence of an active interrupt, simulate effect of interrupt. Else simulate effect of next instruction.
    b) Evaluate truth values of atomic propositions
4) Return resulting states

Depending on the underlying hardware platform, different parts of a state of the hardware can be nondeterministic. In models of microcontrollers, certain I/O registers and external devices such as timers and AD-converters are modeled as nondeterministic. For simulating the effect of instructions, the nondeterminism has to be resolved. Therefore, it is first checked which parts of the state are accessed by the next instruction, and then, these parts are *instantiated* by assigning all possible value combinations to them. For example, if an instruction reads input from an 8-bit I/O port, which is modeled using nondeterminism, 256 different successor states are created: one state for each possible assignment.

For each assignment, the effect of the next instruction is simulated and the atomic propositions are evaluated on the resulting state. The evaluation of atomic proposition takes place in the simulators, and hence, the model checking algorithm itself is independent of the underlying hardware platform. When model checking is finished, depending on the formula and the result, a counterexample or a witness is generated, which is displayed in assembly code, in C code, in the control flow graph of the program, and as a state-space graph. More details on the fundamentals of [MC]SQUARE are given by Schlich [11].

IV. SGDL AND THE SIMULATOR SYNTHESIS SYSTEM

None of the ADLs presented so far features support for nondeterminism, which is necessary to preserve an over-approximation of the behavior of the physical device. Therefore, we have developed a new ADL and a synthesis system suitable for creating simulators for use in model checking. Our new language, which is an extension of the *instruction set description language* used in the AVRora project [13], is called *State space Generator Description Language (*SGDL*)*. A first introduction into SGDL is given in [14].

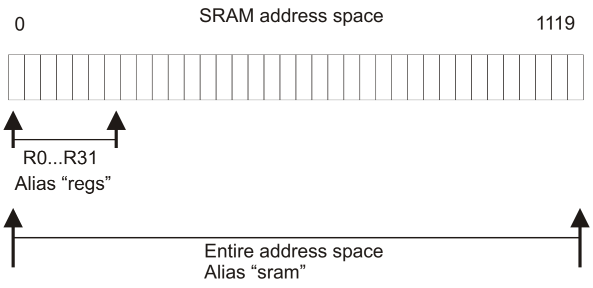

Figure 2. A memory with two aliases

A. *Description of Simulators Using* SGDL

The most important characteristics of an architecture, from the stance of instruction-accurate simulation, are the size and structure of memories, the instruction set, and the interrupt system. This section illustrates how these are modeled in SGDL. Other characteristics, which are also described in SGDL, but not detailed here, involve the generation of glue code, such as loading the program from the binary output generated by the developer's compiler.

1) Memory Description: All instructions operate on a memory model of the microcontroller. Describing memories in SGDL consists of two steps, *memory declaration* and *alias declaration*.

Memory declaration is equivalent to an array declaration in an imperative programming language. It creates a new identifier and allocates a specified amount of physical memory to store the simulated memory. However, in SGDL memory is never accessed directly using the memory array identifier. The reason for this design decision is that several platforms provide more than one way of addressing a memory location. An example is shown in Figure 2. This shows the memory layout of the SRAM address space of an Atmel ATmega16 microcontroller. The 32 general purpose registers are mapped into the main address space, followed by special function registers (not shown in the figure), and eventually, starting at address 96, the actual SRAM. Instructions accessing SRAM addresses 0 to 31 do not read from or write into the SRAM, but instead access the general purpose registers. Vice versa, instructions only dealing with registers do not only affect registers, but also the lower 32 SRAM addresses. For modeling such dependencies, we decided to introduce *aliases* to our language. Aliases separate memory access from memory declaration, and are semantically equivalent to typed pointers in imperative programming languages. The example in Figure 2 contains two aliases: `regs` is an alias for the lower SRAM addresses, and `sram` is an alias for the entire address space. Similarly, we can declare aliases for single bits inside addresses, for instance to provide easy access to frequently used flags such as the carry flag or the global interrupt enable flag, both of which are located in the status register.

Below is the SGDL representation of the memory mapping depicted in Figure 2:

```
memory array data_mem = {
 size = 1120
};

memory alias sram: memory = {
 block_size = 1, block_count = 1120,
 underlying = "data_mem",
 u_from = 0, u_to = 1119,
 op = "ubyte",
 description = "SRAM"
};

memory alias regs: memory = {
 block_size = 1, block_count = 32,
 underlying = "data_mem",
 u_from = 0, u_to = 31,
 op = "ubyte",
 description = "General Purpose Regs"
};
```

In this example, both aliases access *data_mem* in a bytewise fashion (*block_size=1*), but starting from / ending at different locations (*u_from*, *u_to*). Both interpret the content of each accessed memory cell as an unsigned byte value (*op=ubyte*). The value of the optional attribute *description* is used for creating a panel on the GUI for inspecting memory contents, which is used in counterexamples and for manual debugging.

2) Instruction set description: In the following example we illustrate the general principle of describing an instruction:

```
instruction "ADC" {
  encoding = GPRGPR where {
    opcode = 0b000111,
    r1 = rd,
    r2 = rr
  };
  operandtypes = {rd: GPR, rr: GPR};
  cycles = 1;
  execute = {
    $regs(rd) =
      performAddition($regs(rd),
        $regs(rr), $C);
  };
};
```

The instruction is called `ADC` and expects two operands of the (user-defined) types `GPR`. The binary encoding of this instruction is defined in the `encoding` section of the `instruction` element. SGDL allows developers to define patterns of frequently occurring encodings. This is useful if the instruction set can be grouped by adressing modes. In the above example, we use one pattern, `GPRGPR`. It is defined as follows:

```
format GPRGPR = { opcode[5:0],
  r2[4:4], r1[4:0], r2[3:0] };
```

This template is reused in all instructions using exactly two general purpose registers (GPR) as parameters. It takes care of the fact that in the instruction set of the real hardware the value indicating the second register, $r2$, is encoded as a so-called *split field*. That is, all instructions using this encoding contain a bit-pattern in which the first bit of register $r2$ is encoded first, then follow all 5 bits encoding register $r1$. Finally, the last 4 bits of register $r2$ are listed.

The instruction description also contains an associated semantics. The semantics describes the effect the instruction has on the simulated microcontroller when executed. It is usually described in detail in the manual or instruction set summary of the microcontroller. In the above example, the operation associated with the `ADC` instruction is an addition, adding two registers and a carry flag. The effect will be $r1 \leftarrow r1 + r2 + c$, where c is the status of the carry flag in the status register. To avoid redundancies, SGDL allows developers to define own methods, a feature which is used here: the actual calculation of $r1+r2+c$ is performed in the function `performAddition`. Hence, all variants of an add operation, such as add without carry, add with carry, add register and immediate, can all rely on the same implementation of `performAddition`.

3) Nondeterminism in instructions and interrupts: For model checking, it is essential that the state space generator creates an over-approximation of possible system behavior. In order to ensure this, the environment and also certain internal components have to be considered as *nondeterministic*, as their actual value at runtime cannot be predicted. In consequence, simulator developers must be able to specify for which components this is the case. The language elements we have introduced so far cover the same aspects as other ADLs do. Next, we introduce the elements necessary for defining nondeterministic behavior. Three elements are necessary, as described below.

To mark a location as nondeterministic, we prefix its name with a # and write a value to it. This special-purpose value is stored in a data structure called a *nondeterminism mask (nd* *mask)*. The value has to be 0 for all those bits of the location that are to remain deterministic, and 1 for all bits that are to become nondeterministic. This feature can be used to model dependencies between memory locations: if one register holds a certain value (for instance, port configuration registers or timer/counter activation), then another register (the I/O port or timer/counter register) will become nondeterministic. There is such a dependency on the ATmega16 between the data direction registers (DDRA to DDRD for ports A-D) and the port input registers (PINA to PIND). A bit set to 0 in DDRx implies that the pin with that number on port x is an input, i.e., in the SGDL model, its nd mask has to be set to 1. In the C-like assignment syntax of SGDL, this can be achieved in the following way:

```
#PINA = ~$DDRA;
```

This fetches the value of register DDRA (values are accessed by $), complements it, and writes it into the nd mask of register PINA (# access). Subsequent reads from PINA will then trigger an instantiation.

The second language element for dealing with nondeterminism is part of an instruction declaration. It is a signal for the simulator synthesizer to add code for instantiating the named locations before the instruction can be executed:

```
ndread = { $ioregs(imm) };
```

When the instruction accesses the memory alias `ioregs` at address `imm` to fetch its value, instantiation has to take place. This means that all nondeterministic bits in that location have to be assigned a deterministic value, which results in 2^n distinct values if n bits are marked as nondeterministic. In future implementations, we will use a symbolic technique for this, which will avoid the instantiation in some cases (see Sect. VI-C).

Finally, SGDL provides a detailed description mechanism for interrupts. In principle, interrupts are nondeterministic events that interfere with instruction execution. Either the event associated with an interrupt occurs, then it is possible (but not strictly necessary, as the interrupt might be deactivated) that simulation has to continue by entering the interrupt handler. Otherwise, the program continues at its current location (typically the main program). In order to preserve an over-approximation of program behavior, the SGDL description has to contain the following information about each of the interrupts of the platform:

1) A condition for checking whether the interrupt is active and can occur.
2) Code that will be executed when, during simulation, it is decided that the event leading to the interrupt has just occurred. Usually, on the real hardware, in such cases, some flag is automatically set by the hardware, even when the interrupt associated with it is not activated.
3) A condition for verifying that the event has occurred, used when deciding whether any interrupt handler has to be entered, and which, if any.
4) Code for actually entering the interrupt handler. The code here has to model the hardware behavior, such as looking up the active interrupt vector table, and modifying the program counter accordingly.
5) Code returning a priority. By considering the content of the memories when computing the return value, it is possible to correctly model architectures with configurable interrupt priorities (such as the MCS-51, where each interrupt can be assigned to one of two priority levels).

It is worth mentioning that each of these code sections is required. Combining any of them would lead to spurious counterexamples, which could not occur on the real hardware.

B. The Synthesis Framework

The synthesis framework consists of three major components: (1) input language parser, (2) intermediate representation and (3) factories. We will now describe each of these components in detail.

The input language parser forms the interface between the SGDL code created by the developers and the internal representation of the platform architecture, which we refer to as *intermediate representation*. It is based on the parser provided by the AVRora project, though we added several extensions of our own. The task of the parser is to read the input, check for syntax errors, and translate it into the intermediate representation.

Intermediate representation means the memory representation of the information present in the SGDL input file. It forms the basis of the synthesis phase, that is, it fulfills the same task as the abstract syntax tree (AST) in a compiler. Similar to a compiler, we perform several operations on the intermediate representation. The most important of these operations ensure that there are no semantic errors in the representation. For instance, instruction encoding must be unambiguous. If there are two different instructions with the same binary encoding, then we could not decide which instruction to create when we encounter that pattern in the instruction stream.

Factories are the backend of the synthesis framework. They process the intermediate representation and create the platform simulator as output. Currently, our synthesis framework generates Java classes because [MC]SQUARE itself is implemented in Java. Hence, the generated simulator requires a Java compiler, and it is not possible to generate the simulator on the fly without restarting [MC]SQUARE. The advantage of this approach is that the simulator itself will always be compiled and optimized by the compiler before it is executed, thus increasing performance.

C. The Synthesized Simulator

After the synthesis process has terminated, the generated simulator is available in the package structure of the [MC]SQUARE model checker. The synthesis tool automatically extends the simulator by a loader for the ELF format and a loader for the HEX file format. It also reconfigures options and GUI of the model checker such that the new simulator is immediately available for model checking and manual simulation.

V. Tool-based implementation of simulators

This section illustrates the results in describing different microcontroller platforms with SGDL, and model checking programs using the simulators synthesized from these descriptions.

A. Case Study: Atmel ATmega16

The ATmega16 is an 8 bit microcontroller from Atmel's AVR family of microcontrollers. The device has a RISC core, 131 instructions, 3 separate address spaces for program and data (i.e., it is a Harvard architecture), and makes heavy use of memory mapping (register bank and I/O mapped into the main address space for the SRAM). It has 21 different interrupt sources, but no priority for interrupts except their numbering.

1) Modeling the ATmega16 in SGDL*:* Details on the case study concerning the construction of an SGDL description of the ATmega16 can be found in [14]. For the sake of completeness and comparability to the newly created MCS-51 SGDL description, we only briefly resume the most important characteristics here.

The description for the ATmega16 was not created from scratch, but resulted from an extension of an existing AVR instruction set description that was part of the AVRora project. It served as a testbed during the development of the synthesis system. For this reason, the development time for the ATmega16 simulator has not been accurately measured. Also, it influenced some design decisions that turned out as being too hardware-specific with regard to the AVR when the MCS-51 simulator was created.

A total of 2150 lines of code (LOC) (in SGDL) was necessary to describe the ATmega16. From this description, the synthesizer generated the executable simulator consisting of 18350 LOC (in Java).

The generated simulator also includes a set of JUnit test case stubs for all instructions and methods declared in the SGDL file. These tests check whether the disassembler correctly decodes instructions, and whether it is possible to execute the instructions. The binary instruction encoding patterns to be tested are not generated automatically, as this would violate the principle from testing theory that a program may never be tested against itself. Hence, the developer has to fill in the patterns manually into the stubs (empty constants with names relating to the instruction they represent are generated for this purpose), but the testing is then conducted fully automatic by JUnit. Using these tests, we discovered two errors in the original AVRora instruction set description for the AVR family, which were based on missing bits in encodings for some instructions.

B. Case Study: Intel MCS-51

The MCS-51 is a family of microcontrollers from Intel. Its best-known member is the 8051, which was already presented in 1980, but is still widespread today. Just like the AVRs, it is an 8 bit microcontroller, but it only features 128 bytes of RAM (plus an optional another 128 bytes in variants like the 8052). The address space, however, covers at least 256 bytes, wherein the upper 128 bytes contain mappings of special function registers and some bits. The MCS-51 possesses a specialty, which are addressable single bits, and powerful bit-manipulating instructions. For this reason, there is very little need for specialized flags on this architecture, as the developer can easily realize custom flags in software. As to instructions, the family is based on a CISC architecture. The number of instructions reads 111, but, considering duplicate instructions due to different adressing modes, there actually are 256 different instructions.

1) Modeling the MCS-51 in SGDL*:* The MCS-51 SGDL description was not based on any previous work. Hence, we could accurately measure the effort for implementing a simulator using SGDL when implementing it. The effort was considerably lower than when manually implementing a simulator. [MC]SQUARE already contains a handcrafted simulator for this platform, which took the usual 6 months span. Using SGDL, on the contrary, the simulator was already operational (memories and instruction set description) after 23 hours. Programs could be loaded, instructions could be disassembled and executed, and their effects were visible on the GUI. The current status of the work on this simulator, after a total effort of only 40 hours, is that model checking is possible (all named registers of the device can be used as atomics in CTL formulas), debugging on the GUI is possible, and the interrupt system has been implemented. The effort of 40 hours already includes the time required for adapting the synthesis system wherever it was designed too AVR-specific, and for optimizing the performance in bottlenecks that have been discovered. As future simulators will benefit from these improvements as well, we suspect that the effort for implementing further platforms could be even lower.

As AVR and MCS-51 are fairly different (RISC vs. CISC architecture, sizes of memories), the successful implementation of the MCS-51 also proves that the SGDL approach is general enough to allow implementation of arbitrary platforms with acceptable effort.

VI. Abstraction

Abstraction refers to all techniques used for reducing the amount of infomation to be stored. Any abstraction, or abstraction technique, has to pursue two goals: first, reduce the number of states to be stored by dropping not required information, and second, do not sacrifice too much information. Dropping more information than is theoretically allowed might cause the model checker to miss errors, or might, in the better case, simply lead to spurious counterexamples (i.e., counterexamples that could never occur on the real system). Hence, special care is needed when creating new abstractions for simulators.

When applied carefully, abstractions can greatly assist the verifiction process. Systems containing billions of states, such that cannot be handled due to lack of memory, or taking

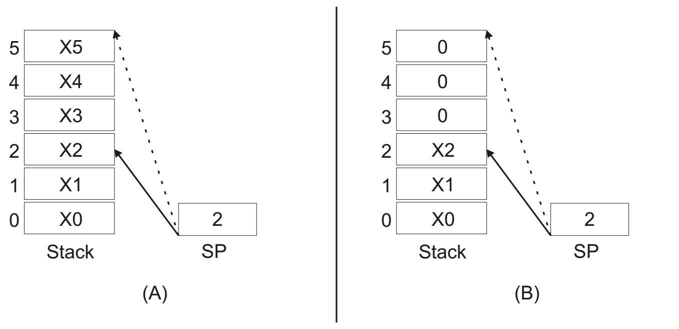

Figure 3. Stack management (A) without and (B) with Lazy Stack Evaluation

weeks to verify, can be reduced to systems of a few thousands states, which can be handled even by an average desktop computer within a matter of seconds. The existing handcrafted simulators have been equipped with such abstractions, each for itself as the abstractions themselves are typically hardware-specific.

A. Lazy Stack Evaluation

Lazy stack evaluation (LSE) is a very powerful abstraction that preserves the expressiveness of the abstracted system. In the handcrafted simulators, it has proven itself to be an invaluable abstraction, without which most programs would not be manageable for the model checker, and which is therefore active by default.

The idea behind lazy stack evaluation is shown in Figure 3. It is based on the premise that a program generated by a compiler will only access the stack of the hardware via `push` and `pop` operations, but never read or write directly from it. Whenever an element is taken from the stack by a `pop` operation, it will never be read again. So the stack pointer register (or registers, in case the platform provides more than one), which always points to the top of the stack, also points to the last element that actually needs to be stored. This observation is exploited by LSE. By resetting the elements beyond the top to a common reset value, it allows states that only differ in those irrelevant memory locations to be merged. In the example in Figure 3, this is depicted for a previous value of the stack pointer (SP) of 5. By 3 `pop` operations, the value of SP has been reduced to 2, allowing for the values X3, X4 and X5 to be discarded.

It is possible to guarantee a valid over-approximation even in the rare case the program should read a value beyond the location pointed to by SP. For this, locations are not only reset on `pop`, but also marked as nondeterministic. Vice versa, pushing a (deterministic) value on the stack will render a location deterministic again. Thus, should the program ever read beyond SP, instantiation will take place, generating all possible values for the location, including the original one before the reset.

Obviously, LSE can have little impact if the program scarcely modifies the stack pointer. Table I illustrates this for the test program called *Experimental Plant* (formula: AG TT, i.e. build full state space), which contains only 2 active

Table I
MODEL CHECKING RESULT COMPARISON FOR [MC]SQUARE ATMEGA16 SIMULATORS, TEST PROGRAM EXPERIMENTAL PLANT

Property	Synthesized	Synth. with LSE	Handcrafted
states stored	105,445	75,145	130,524
states created	115,846	78,476	135,949
transitions created	115,846	78,475	135,948
time	5.5 s	4.5 s	4.2 s

Table II
MODEL CHECKING RESULT COMPARISON FOR [MC]SQUARE ATMEGA16 SIMULATORS, TEST PROGRAM WINDOW LIFT

Property	Synthesized	Synth. with LSE	Handcrafted
states stored	426,239,648	3,763,032	129,030
states created	749,355,722	5,219,519	205,223
transitions created	749,355,721	5,219,518	138,982
time	5h 10 min	2 min 37 s	9.7 s

interrupts and no function calls except an initialization method, which is called only once. The opposite case is given when there are many function calls or even interrupts, which can occur in any ordering, and thus leave the stack in many different combinations of remaining return values. Under such circumstances, LSE can reduce up to 98% of the state space, as is shown in Table II.

Table II is a re-run of test case III from the case study in [14], illustrating the impact of the newly added support for LSE in synthetic simulators. The program used for these runs contains three active interrupts and a main program, resulting in 4 pseudo-parallel threads manipulating the stack. Compared to [14], the values for the column labeled *Synthesized* increased due to a necessary change in the modeling of interrupts (e.g., *states stored* increased from 413 million to 426 million).

Adding the support for LSE requires only the following modifications to the SGDL description:

- in `pop`:
  - store the value of the stack, which is about to be returned, in a temporary variable (instead of returning it immediately)
  - reset the value by writing a 0 to the SRAM location pointed at by SP
  - set all bits of the location as nondeterministic by writing 0x255 to its nd mask
  - return the value
- in `push`: reset the nd mask of the location where the new element is placed by writing a 0 to it, thus marking it as deterministic

Thus, the overall effort amounts to three new lines of code (reset value, set nd mask, and reset nd mask). It is possible to automate this by providing a generic version of `push` and `pop` to the simulator developer. However, the generic methods need to consider platform peculiarities such as endianness, word size, the direction in which the stack grows, whether SP points at the last occupied or the next free position, and the possibility of multiple stacks (as present for instance in the Renesas R8C). Hence, they would not necessarily relieve the developer due to their own complex handling.

The most important result from adding support for LSE in synthetic simulators is, however, not just the vast reduction of state space size caused by this specific abstraction technique. Instead, we consider the fact that also when using the synthesis tool for implementing simulators, it is possible to add abstraction. Hence, it is possible to consider further abstractions, some of which are illustrated in the next subsections.

B. Dynamic Path Reduction

The idea behind *path reduction* is to reduce long chains of states with only one successor each into a single step from the first state in the chain to the last. Apart from saving memory, this also has the advantage that the simulator state does not need to be stored in the state space after each step, which is very time-consuming (save and restore of simulator states consume up to 90% of the time in model checking with [MC]SQUARE, as memory is the bottleneck in modern computers). A possible disadvantage, which can occur depending on the program and the formula, is that states that have to be revisited during model checking may have to be recreated (when they are not stored). Hence, the typically much smaller state space size does not imply that the time required for model checking also shrinks. Instead, the time effort can even increase. Several conditions must be met for allowing states to be compacted by path reduction:

- no split-up due to instantiation of a nondeterministic value
- no active interrupts
- no change of any memory location relevant for the CTL formula
- no state occurs twice in the chain, i.e. no loop in the state space

Static path reduction [15], [16] uses a static analyis to derive the information about instantiation and active interrupts. The analysis is conducted prior to simulation. When in doubt about whether the condition will be fulfilled at a specific location, the information provided by the static analysis has to be that reduction is not possible. *Dynamic path reduction (DynPR)* does not require a static analysis, but simply checks for each created state during simulation whether the above conditions are still satisfied. If so, the state becomes part of the current chain. This results in additional effort at runtime, but is more accurate, as there can be no doubt about whether it is safe to add a state to a chain. Thus, dynamic path reduction is more effective than static path reduction, and can be implemented with far less effort. Hence, there is no reason for using the static version any more. Actual results from the handcrafted ATmega simulators, for which both versions have been implemented, underline this conclusion.

For the above reasons, we have not tried to create a generic version of static path reduction for synthetic simulators, but directly implemented the dynamic version. The checking for only a single successor (related to the first two conditions in the above list) is trivial, as we only need to verify, after creating the successors of a state, whether there is only a single successor. Loop detection in the state space (fourth condition) is not even hardware-dependent at all. The only obstacle we encountered was related to the third condition, that is, checking whether the step modifies any memory location relevant for the CTL formula. The problem is the same as for *Dynamic Delayed Nondeterminism* (cf. Sect. VI-C), detecting memory addresses accessed by an instruction. Fig. 4 illustrates our solution for this problem. The idea is to monitor write accesses to memories during successor state creation. If an access modifies a formula-relevant location, then the successor state which was just created cannot be added to the current chain. Instead, its predecessor (to which we need to keep a reference) is the final state in the chain. This approach causes the synthetic simulator to always perform one step more than necessary to discover formula modification. However, it eliminates any need for knowing the effects of instruction execution in advance, thus facilitating implementation.

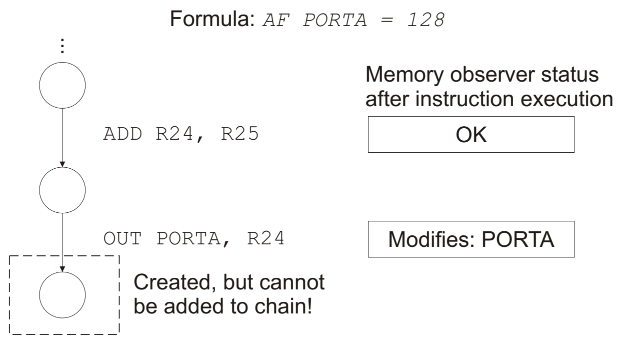
Formula: AF PORTA = 128
Memory observer status after instruction execution
ADD R24, R25
OK
OUT PORTA, R24
Modifies: PORTA
Created, but cannot be added to chain!

Figure 4. Principle used for Dynamic Path Reduction. The memory observer is notified on each write access, and computes whether $Written\ addresses \cap Addresses\ in\ formula = \emptyset$.

As the support for DynPR is completely independent of the SGDL description of the platform, the synthesizer can safely generate the necessary code for it. Thus, all synthetic simulators in [MC]SQUARE support it.

In order to evaluate the effect of dynamic path reduction, we have rerun our case studies from Tables I and II using the synthetic ATmega16 simulator. Table III shows the result for the Window Lift program. Generating the entire state space with LSE and DynPR active took approximately 4 minutes. The number of *created states* increased to 12,347,057, which is why the time required was not reduced. The key difference is that this result can also be achieved (in the same time) by an average desktop computer, whereas the state space without abstraction could only be built on a server equipped with 256 GB RAM. For the Experimental Plant program, we observed a similar reduction of the size of the state space, for which the number of stored states was reduced from (without abstraction) 105,445 to 2,221 (LSE+DynPR).

C. Dynamic Delayed Nondeterminism

Most of the simulators in [MC]SQUARE already feature an abstraction called *delayed nondeterminism* [11]. The intention of this technique is to avoid immediate instantiation when

Table III
COMPARISON OF EFFECTS OF ABSTRACTIONS IN SYNTHETIC SIMULATORS, TEST PROGRAM WINDOW LIFT

Abstraction used	Number of states stored
No abstraction	426,239,648
LSE	3,763,032
DynPR	40,734,991
LSE + DynPR	166,345

Table IV
COMPARISON OF ABSTRACTIONS FOR THE HANDCRAFTED ATMEGA16 SIMULATOR, TEST PROGRAM WINDOW LIFT

Abstraction used	Number of states stored
LSE only	1,660,560
LSE + Static DND	129,030
LSE + Dead Variable Reduction	159,693
LSE + DynPR	75,717
All available combined	701

accessing nondeterministic locations, i.e., postpone split-up of the computation path into several trajectories as long as possible. For this purpose, we allow internal registers of the simulated microcontroller to contain nondeterministic values. Thus, it becomes possible to copy the information about nondeterministic bits from one memory location to another, instead of immediately generating all possible bit patterns and creating distinct states for each. Only when the nondeterministic value is to be used in an operation writing to a location that has to remain deterministic, it will have to be instantiated. Until then, it will remain symbolic, hence avoiding an exponential blow-up in the number of nondeterministic bits. Examples of locations which have to remain deterministic are any locations used in the CTL formula, and also status registers (nondeterminism in status registers results in nondeterministic control flow, which is more difficult to handle than nondeterminism in the memory model only).

The complexity of adding this abstraction to synthetic simulators lies mainly in the classification of instructions, and in the description of memory locations that must always remain deterministic. For any instruction that might, for some actual parameters at runtime, write to a memory location that must remain deterministic, special code has to be generated. This code has to check whether the parameters are nondeterministic, and if so, instantiate before the instruction is executed. For all other instructions, the checking code should not be generated for performance reasons. Instead, nondeterminism should be propagated according to the intended behavior of the instruction. When manually implementing this abstraction for a simulator, performing these steps is simple. On the other hand, performing them *automatically* requires knowledge about which locations are read and written by an instruction. In theory, this would be possible by conducting static analyses of the `execute` section of each instruction (like Live Variable Analysis and Reaching Definition Analysis). However, SGDL is Turing-complete, allowing for recursive function calls from within `execute` sections. Hence, the analyses would have to be conducted not only intra-procedurally, but *inter-procedurally*.

In order to reduce the development effort, we have developed a new variant of delayed nondeterminism. The new technique is called *Dynamic Delayed Nondeterminism (DDND)*, as opposed to the existing technique, which we now refer to as *Static Delayed Nondeterminism (SDND)*. It is based on a description of locations that must remain deterministic, which has to be described by developers in the SGDL file. Unlike for SDND, this information will not be used by the synthesis tool for distinguishing different instruction classes. Instead, it will be propagated to the memory structure used by the simulator at runtime. During simulation, instructions invariantly have to check the memory locations they access for the need to instantiate. Compared to SDND, this will result in runtime overhead for each instruction, but will have the same effect with regard to the number of states that will not have to be created. The idea for DDND resulted from observations in Dynamic Path Reduction, which produces improvements similar to Static Path Reduction, but with far less effort and neglectible runtime overhead. It is not yet possible to state a similar result for DDND, as the technique has been developed and is based on principles that are known to work correctly (same as in SDND), but has not been implemented yet.

D. Dead Variable Reduction

Dead Variable Reduction (DVR) is a generalization of the principle used in LSE. Whenever there is a memory location (variable), a so-called *dead* location, that is not read again before it is overwritten, then there is no need to store this value until the next write. The value can be reset to 0. Thus, states that differ only in the value of such a *dead variable* can be merged into a single state, as this abstraction preserves liveness [17].

Discovering dead variables is possible by a static analysis called *Live Variable Analysis* [15], [16]. This type of analysis is also used by compilers to discover unnecessary assignments (when the value is not read after assigning it, then the assignment is *dead code* and can be eliminated). Some approaches exist which can conduct DVR without any static analysis, that is, the DVR is performed *on the fly* during state space building [18]. However, the problem for any on-the-fly approach is that it requires knowledge about the future. The question to be answered for any variable is whether the next access will be a read or a write. Therefore, approaches like the one in [18] usually require severe restrictions like the absence of nondeterminism in the system (without nondeterminism, the computation path cannot branch, i.e., there is only one possible future). Hence, for our purposes such an approach would be infeasible.

As shown in Table IV, DVR has a similar impact on the state space size as DND has. The program used for obtaining these values is the same as the one used for Tables II and III.

Implementing a generic DVR for synthesized simulators requires first of all a static analysis of the input program. The analysis has to identify all memory locations modified by the code. For this purpose, the synthesizer must provide information on the memory locations which are modified by each instruction, and compile them as a hardware-dependent static analysis into the generated simulator. However, deriving this information automatically requires either a static analysis of the SGDL code itself (as for SDND) or an explicit list of locations provided by the developer. The solution we used to circumvent this obstacle for dynamic path reduction (monitoring write accesses) is not applicable here, as it can only be used *during* simulation, not *prior to it*. Having gained the information about modified memory locations, adding the actual abstraction technique will then be rather simple: after each step, before attempting to store the new state in the state space, check which variables are dead, and reset them. In case this state is already present, do not store it again.

VII. Conclusion

This paper shows that automatically synthesized simulators are a competitive alternative to handcrafted simulators for binary code model checking. While the generation of hardware simulators from hardware descriptions has been long used in different fields, our work is new in that it highlights and approaches the need for automatic integration of abstraction techniques in order to tackle the state-explosion problem. As shown in the case study, only the smallest programs can be verified otherwise.

Clearly, the work calls for further investigation of abstraction techniques that can be integrated automatically, and also, the synthesis of hardware-specific static analyzers from SGDL code. Moreover, even though we believe that SGDL and the underlying synthesis system are general enough to handle different kinds of microcontroller platforms, we further want to investigate its applicability by synthesizing simulators for 16-bit microcontrollers such as the Renesas R8C/23.

Acknowledgment

This work was partly supported by a PhD scholarship from the German Research Foundation (DFG). It was also supported partly by the UMIC Research Centre, RWTH Aachen University.

References

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. San Diego, California, USA: ACM Press, June 7–14 2003, pp. 196–207.

[2] S. Bardin and P. Herrmann, "Structural testing of executables," in *ICST 08*. IEEE, 2008, pp. 240–249.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.

[4] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine, "Taxys: A tool for the development and verification of real-time embedded systems," in *Computer Aided Verification (CAV 2001)*, ser. LNCS, vol. 2102. Springer, 2001, pp. 391–395.

[5] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT, USA: ACM, Oct. 2009, pp. 207–220.

[6] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Embedded Software (EMSOFT 2008)*. ACM, 2008, pp. 255–264.

[7] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Design, Automation and Test in Europe (DATE '99)*. ACM, 1999, pp. 485–490.

[8] W. Qin, S. Rajagopalan, and S. Malik, "A formal concurrency model based architecture description language for synthesis of software development tools," in *Languages, Compilers, and Tools for Embedded Systems (LCTES '04)*. ACM, 2004, pp. 47–56.

[9] CoWare. CoWare Processor Designer. [Online]. Available: http://www.coware.com/products/processordesigner.php

[10] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV version 2: An opensource tool for symbolic model checking," in *Computer Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Springer, 2002, pp. 241–268.

[11] B. Schlich, "Model checking of software for microcontrollers," Dissertation, RWTH Aachen University, Aachen, Germany, June 2008. [Online]. Available: http://aib.informatik.rwth-aachen.de/2008/2008-14.pdf

[12] P. Leven, T. Mehler, and S. Edelkamp, "Directed error detection in C++ with the assembly-level model checker StEAM," in *Model Checking Software (SPIN 2004)*, ser. LNCS, vol. 2989. Springer, 2004, pp. 39–56.

[13] B. Titzer. AVRora. [Online]. Available: http://compilers.cs.ucla.edu/avrora/

[14] D. Gückel, B. Schlich, J. Brauer, and S. Kowalewski, "Synthesizing simulators for model checking microcontroller binary code," in *Proceedings of the 13th IEEE International Symposium on Design & Diagnostics of Electronic Circuits and Systems (DDECS 2010)*, 2010.

[15] K. Yorav and O. Grumberg, "Static analysis for state-space reductions preserving temporal logics," *Formal Methods in System Design*, vol. 25, no. 1, pp. 67–96, 2004.

[16] B. Schlich, J. Brauer, and S. Kowalewski, "Application of static analyses for state space reduction to microcontroller binary code," *Science of Computer Programming: Special Issue on FMICS 2007 & 2008*, 2010, to appear.

[17] M. Bozga, J.-C. Fernandez, and L. Ghirvu, "State space reduction based on live variables analysis," in *Static Analysis (SAS 1999)*, ser. LNCS, vol. 1694. Springer, 1999, pp. 164–178.

[18] J. P. Self and E. G. Mercer, "On-the-fly dynamic dead variable analysis," in *Model Checking Software (SPIN 2007)*, ser. LNCS, vol. 4595. Springer, 2007, pp. 113–130.