Automatic Derivation of Abstract Semantics From Instruction Set Descriptions

Dominique Gückel and Stefan Kowalewski

Embedded Software Laboratory
RWTH Aachen University
Ahornstr. 55
Aachen, Germany
<gueckel, kowalewski>@embedded.rwth-aachen.de

Abstract

Abstracted semantics of instructions of processor-based architectures are an invaluable asset for several formal verification techniques, such as software model checking and static analysis. In the field of model checking, abstract versions of instructions can help counter the state explosion problem, for instance by replacing explicit values by symbolic representations of sets of values. Similar to this, static analyses often operate on an abstract domain in order to reduce complexity, guarantee termination, or both. Hence, for a given microcontroller, the task at hand is to find such abstractions. Due to the large number of available microcontrollers, some of which are even created for specific applications, it is impracticable to rely on human developers to perform this step. Therefore, we propose a technique that starts from imperative descriptions of instructions, which allows to automate most of the process.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.3.4 Processors, F.3.2 Semantics of Programming Languages

Keywords and phrases Model Checking, Static Analysis, Hardware Description Languages

Digital Object Identifier 10.4230/OASIcs.SSV.2011.71

1 Introduction

Formal verification of software for embedded systems is crucial for multiple reasons. First of all, such systems are often used in safety-critical fields of applications, such as chemical plants, where failures of the controlling system may result in severe injuries or even fatalities. Furthermore, applying corrections after delivering a system to the customer may be inpracticable or costly, for instance in the case of devices embedded into cars. Such scenarios may be avoided by formal verification, for instance software model checking [4, 2].

1.1 Focus

The focus of our work is model checking and static analysis [5] of binary code for microcontrollers. For this purpose, we need to lift the given concrete semantics of the instructions of which the binary consists to their abstract counterparts in the respective domain. In the case of model checking, the sought-after abstract version of each instruction should be able to operate not only on conventional two-valued boolean logic, but on a variant of three-valued logic. This allows for certain abstractions to be applied, which can help avoid the state explosion problem. In the case of static analysis, the abstracted instruction should provide information on memory locations it reads and writes, plus on how executing it affects the control flow.

6th International Workshop on Systems Software Verification (SSV'11).
Editors: Jörg Brauer, Marco Roveri, Hendrik Tews; pp. 71–83

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Deriving abstract semantics from concrete semantics is a task that is usually performed manually, thus exploiting the knowledge of an expert in the associated fields. While this may be suitable for verification tools that are not likely to be modified very often, such as for high level languages, this is not applicable in the embedded domain, where a wide variety of different platforms is available to the developer. In case the platform is switched to a different microcontroller, which uses a different instruction set, the previous work on abstraction has to be done anew.

1.2 Approach

In order to reduce the necessary effort, we propose to conduct the abstraction on an already executable form of the instructions, that is, a description in an imperative programming language. In our setting, such a description can also be used to generate an instruction set simulator, which can build the state space for model checking programs for the described platform.

1.3 Contribution

In this paper, we make the following contributions:

- We describe how microcontroller instruction sets can be translated into a form that allows for automatic analysis of certain properties.
- Based on the results of these analyses, we can then derive abstract semantics that are more suitable for state space building than the concrete semantics. As an example, we detail how to obtain the necessary semantics for an abstraction called *delayed nondeterminism*, which can be used in model checking.
- We detail the generation of static analyzers for different platforms based on the aforementioned analyses.

1.4 Outline

The rest of this paper is structured as follows. In Sect. 2, we illustrate the tools we used in our contribution. Next, to motivate our work, we provide an example that illustrates the effects of an inappropriate modeling of instructions. The actual work on automatically deriving abstract semantics is contained in Sect. 4. The next-to last section focuses on related work (Sect.5), and Sect. 6 concludes this paper.

2 Preliminaries

In this section, we detail the environment of our contribution. First, we summarize the main features of the [MC]SQUARE model checker, which uses several of the abstraction techniques we are interested in. Next, we focus on a specific technique, called delayed nondeterminism. Finally, we present some features of a hardware description language that serves as a starting point for our automatic derivation of instruction semantics.

2.1 The [mc]square Assembly Code Model Checker

[MC]SQUARE [13] is an explicit-state model checker for microcontroller binary code. Given a binary program and a formula in Computation Tree Logic (CTL), [MC]SQUARE can automatically verify whether the program satisfies the formula, or create a counterexample in case the program violates the formula. Atomic propositions in formulas may be statements

about the values of general purpose registers, I/O registers, and the main memory. Currently, [MC]SQUARE supports the Atmel ATmega16 and ATmega128, Intel MCS-51, and Renesas R8C/23 microcontrollers. Furthermore, it can verify programs for Programmable Logic Controllers (PLCs) written in Instruction List (IL).

[MC]SQUARE builds state spaces for conducting the actual model checking by means of special simulators. These can execute the programs under consideration just like simulators provided by hardware vendors, by applying the semantics of instructions to a model of the system's memories, and simulating the effects of interrupts, I/O ports, and on-chip peripherals. The key difference, however, is that simulators in [MC]SQUARE support nondeterminism to model unknown values, and also provide certain abstractions. Nondeterminism is introduced into the system by I/O ports, timers, and interrupts. I/O ports communicate with the environment, of which we have to assume that it can show any behavior, i.e., any value might be present in I/O registers. Timers are modeled using nondeterminism because [MC]SQUARE deliberately abstracts from time, resulting in the value of timer registers to be unknown. Finally, interrupts are nondeterministic events because an active interrupt may occur or not occur, and both cases need to be considered for model checking. In case a nondeterministic bit has to be *instantiated* to a deterministic 0 or 1, the simulator performs the necessary step.

The state creation process in [MC]SQUARE operates as follows:

- Load a state into the simulator.
- Determine assignments needed for resolving nondeterminism.
- For each assignment
  - If the assignment indicates the occurrence of an enabled interrupt, simulate the effect of that interrupt. Otherwise, execute the current instruction.
  - Evaluate truth values of atomic propositions.
- Return resulting states.

Using and resolving nondeterminism creates an over-approximation of the behavior exhibited by the real hardware, allowing [MC]SQUARE to check for safety properties. Instantiation of n nondeterministic bits usually results in 2^n successor states (i.e., exponential complexity), which is why immediate instantiation of all nondeterministic bits is infeasible. Therefore, several abstraction techniques are implemented in [MC]SQUARE to prevent this. Within the scope of this paper, we focus on two of these: first of all, a technique called *delayed nondeterminism*, details on which are given in the next section, and second, on techniques that are enabled by static analyses. The latter is an optional preprocessing step performed before conducting the actual model checking, during which analysis results can be used to apply abstractions such as Dead Variable Reduction [17, 14].

2.2 Delayed Nondeterminism

An instantiation of nondeterministic bits results in an exponential number of successor states. Deterministic simulation triggers instantiation whenever an instruction accesses a nondeterministic memory cell, hence it cannot avoid the exponential blowup. However, at least on RISC-like load-store-architectures like the Atmel ATmega microcontrollers, the instruction in question usually only *copies* the content of the cell to some other cell, for instance a register. It does not modify the value or use it as an argument, as would an arithmetic or logic instruction. *Delayed nondeterminism* [11] is an abstraction exploiting this observation. Instead of immediately resolving the nondeterminism, a nondeterministic value is *propagated* through memory. Only when an instruction actually needs the deterministic

value, the cell is instantiated. As a result, all the paths starting at the original read instruction are not created at all. This approach proves particularly useful in case a value consisting of multiple nondeterministic bits is read, of which only a few bits are actually needed, for instance by an instruction testing a single bit.

2.3 Description of Microcontrollers Using SGDL

The concept of [MC]SQUARE requires the tool to be hardware-dependent. While this provides great accuracy as to hardware peculiarities, and also the ability to provide easy to understand counterexamples, it necessarily results in the obvious disadvantage of additional effort whenever adding support for a new platform. In order to compensate for this, [MC]SQUARE features an extensible architecture, and additionally contains a complete programming system for creating simulators in a high level language. The language is called SGDL, and a compiler for SGDL is part of [MC]SQUARE. SGDL is a hardware description language specifically tailored to describe microcontroller architectures, providing elements for describing entities such as instructions, memories, and interrupts. In the following, we only introduce those parts relevant for analyzing instruction semantics. Further details on SGDL are provided in [7, 6], and details on its precursor language from the AVRora project are available from [16].

▶ **Example 1.** Excerpt from the SGDL description of the Intel MCS-51

```
format OPCODE_IMM8_IMM8 = {opcode[7:0], imm8_1[7:0], imm8_2[7:0]};

subroutine performCJNE(leftVal:ubyte, rightVal:ubyte,
   target:SIGNED_IMM8) : void {
   if (leftVal != rightVal) {
      $pc = $pc + target;
      if (leftVal < rightVal) $CY = true;
      else $CY = false;
   }
};

instruction "cjne_acc_direct_rel" {
   encoding = OPCODE_IMM8_IMM8 where {opcode = 0b10110101};
   operandtypes = {imm8_1 : IMM8, imm8_2 : SIGNED_IMM8};
   instantiate = {};
   dnd instantiate = {};
   execute = {
      performCJNE($ACC, $sram(imm8_1), imm8_2);
   };
};
```

In the example, an instruction called `cjne_acc_direct_rel` is declared. The binary encoding of this instruction consists of an 8 bit wide opcode and two operands, each of which is also 8 bits wide. Within the scope of this instruction, these 8 bit operands are to be interpreted as signed 8 bit integers, using two's complement representation. The concrete semantics are described within the `execute` section of the `instruction` element. Global variables, i.e., the resource model of the simulated device, are accessed by prefixing the according identifier with a $ or a # (not in this example), whereas local variables are always accessed with neither. Function calls are also possible, in this example for externalizing the CJNE functionality,

which is shared by the different variants of the CJNE instruction (the MCS-51 instruction set contains four of these, each for a different addressing mode).

In case an instruction may encounter nondeterministic values in some addresses it accesses, the developer can indicate that the simulator should instantiate these by adding an `instantiate` entry to the instruction. Any address contained in the set will be instantiated. The `dnd instantiate` section has the same semantics, but is used only when the simulation type is set to *use delayed nondeterminism*.

Global memories in SGDL consist of two parallel structures to allow for nondeterminism. The first structure is the *value*, which is accessed using the aforementioned dollar symbol. The second structure is the *nondeterminism mask*. Both structures together represent values in ternary logics, with the semantics that a bit is nondeterministic iff its nondeterminism mask is set to 1. If the mask bit is set to 1, then the content of *value* becomes irrelevant, as logically, it could be either 0 or 1. Hence, generated simulators force it to 0, thus guaranteeing consistent states and additionally removing a potential distinguishing feature of states (which in some cases reduces the size of the state space).

A typical instruction set description in SGDL contains between 2.000 and 4.000 lines of code, depending on the number of instructions and overall complexity of the device.

2.4 Notations

▶ **Definition 2. Alphabet for ternary logics**
The alphabet for ternary logics is defined as $\Sigma := \{0, 1, n\}$. A word of length m over Σ^* is then a sequence of letters representing bits that are either explicitly 0, 1, or could be both.

▶ **Example 3.** The word $w := 000n\,0000$ can be instantiated to the explicit values $0000\,0000$ and $0001\,0000$.

▶ **Definition 4. Values of a memory cell**
Let x be a memory cell of m bits width. Then
- $\texttt{val}(x)$ denotes the content of x.
- $\texttt{ndm}(x)$ denotes the content of the nondeterminism mask of x.

`val` and `ndm` are bit vectors that can be combined to represent a value in ternary logic. Whenever a bit in $\texttt{ndm}(x)$ is set to 1, then that bit is considered to be $n \in \Sigma$, i.e., the content of $\texttt{val}(x)$ for that bit becomes irrelevant.

3 A Motivational Example

As an example, consider the following instruction, which is part of the instruction set of the Atmel AVR family of microcontrollers:

```
IN R0, TIFR
```

This instruction reads the value of the *timer interrupt flag register*, `TIFR`, and copies it into the general purpose register (GPR) R0. No flags are altered by this instruction. Accordingly, the semantics of the instruction, as depicted in the instruction set manual (ISM) are

$$\texttt{Rd} \leftarrow \texttt{I/O}$$

where `Rd` is a GPR, and `I/O` is an I/O register. Being an I/O register, `TIFR` may contain nondeterministic data. Hence, we either need to instantiate all nondeterministic bits immediately, or propagate this information into the destination, in this example `R0`. For simulation

(i.e., state space building), the optimal abstract semantics would be

$$\texttt{val}(\texttt{R0}) \leftarrow \texttt{val}(\texttt{TIFR}) \tag{1}$$

$$\texttt{ndm}(\texttt{R0}) \leftarrow \texttt{ndm}(\texttt{TIFR}) \tag{2}$$

To achieve the latter, we have several options:

1. Implement explicit code for copying. This approach requires a human developer to inspect the instruction semantics in the ISM, and implement the necessary code into the simulator.
2. Naive automatic approximation. Whenever an instruction depends on at least one nondeterministic input bit
   - Check if all output bits allow nondeterminism. To obtain the output bits without analyzing the instruction code, it suffices to execute the instruction once, then revert the resulting machine state to the original state.
   - If all output bits allow nondeterminism, set all of them to nondeterministic.
   - Else instantiate all input bits, and execute the instruction using the concrete semantics from the ISM.

While the first approach, using a developer, can always yield the optimal solution, it is also the most inappropriate one. Instruction sets usually consist of hundreds of instructions, and each of those has to be lifted from the concrete to the abstract. Furthermore, this is a very simple example, in which the developer can hardly introduce any mistakes. Other examples, like complex arithmetic instructions, can easily cause the developer to forget maybe the one or other flag bit, which may result in the state space generator containing a correct implementation for one simulation type (e.g., fully deterministic simulation based on the concrete semantics), and a faulty one for another type (e.g., delayed nondeterminism).

Compared to this, the proposed naive implementation of the automatic approach certainly has the advantage of far less manual effort. Moreover, it guarantees an over-approximation of instruction behavior, therefore preserving the model checker's ability to check safety properties. The disadvantage, however, is that it is grossly inaccurate. Consider the following machine state:

$$\texttt{val}(\texttt{R0}) = 0000\,0001 \tag{3}$$

$$\texttt{ndm}(\texttt{R0}) = 0000\,0000 \tag{4}$$

$$\texttt{val}(\texttt{TIFR}) = 0000\,0000 \tag{5}$$

$$\texttt{ndm}(\texttt{TIFR}) = 1000\,0000 \tag{6}$$

Executing the example instruction using the naive abstract semantics will change this to the following machine state:

$$\texttt{val}(\texttt{R0}) = 0000\,0000 \tag{7}$$

$$\texttt{ndm}(\texttt{R0}) = 1111\,1111 \tag{8}$$

$$\texttt{val}(\texttt{TIFR}) = 0000\,0000 \tag{9}$$

$$\texttt{ndm}(\texttt{TIFR}) = 1000\,0000 \tag{10}$$

That is, the source register, `TIFR`, retains its single nondeterministic bit, while the previously deterministic target register `R0` becomes completely nondeterministic. The consequences of this change depend entirely on the next instructions accessing `R0` (note that this need not necessarily be the immediately next instructions). In case the next instruction accessing `R0` is a bit test instruction such as `SBIC` (i.e., skip next instruction if bit is clear), only a single bit

may be instantiated. In this case, the naive approach would yield the desired result, which is to avoid instantiation until the value of nondeterministic bits is actually needed. However, in case the next instruction is a comparison, such as `CPI R0, 128`, the naive approach will result in an instantiation of 8 nondeterministic bits, yielding 256 successor states. Compared to this, the optimal approach would copy only 1 nondeterministic bit from TIFR to R0, thus the instantiation triggered by executing `CPI` would result in only 2 successors.

The disadvantage of the naive implementation is due to the fact that it marks all bits written by the instruction as nondeterministic. Such an approximation is overly pessimistic for `IN`, as there is a direct mapping of input bit i to output bit i in the equally wide registers `TIFR` and `R0`. For operations such as `ADDC Rd, Rr` (addition with carry), however, there is no such mapping. Instead, the value of a target bit may depend on the values of *several* bits in the input. Thus, without any additional knowledge about the actions performed by an instruction, the pessimistic assumption that any output may result, is actually an appropriate one. In the following sections, we illustrate a concept how to gain such information, and produce a smaller over-approximation.

4 Deriving Abstract Semantics

In this section, we focus on abstract semantics for delayed nondeterminism and static analysis. Throughout the section, we use the term *input* of an instruction as a synonym for the sets of locations read by it, and analogously the term *output* for the set of written locations.

4.1 Prerequisites

Certain invariants regarding memory locations must always hold in both the concrete and the abstract semantics of instructions, as they are needed to preserve expressiveness:

- Operands. We assume that operands are always deterministic. This is guaranteed by construction, as they are instantiated once by the disassembler.
- Addresses. Addresses must always be deterministic, as a nondeterministic address used in an instruction may result in any (visible) address to be read or written. Especially on devices with memory-mapped I/O, this could also have an impact on device behavior.
- Control flow. Any memory location relevant for control flow must remain deterministic. This applies to status registers, but not to general purpose registers. Nondeterministic control flow is undesirable because of a severely reduced expressiveness (e.g., a status register indicating that the last computation yielded a result that was zero, negative, and odd). The direct implication is that control-flow relevant instructions must always operate on deterministic data.
- Arithmetics and logics. Any computation involving a nondeterministic value yields a nondeterministic result. If the target of the assignment requires the result to be deterministic, then all variables involved in the computation have to be instantiated first.

4.2 Identifying the Control Flow Type

As pointed out in Sect. 4.1, all control-flow relevant operations require their input to be deterministic. The task at hand is therefore to separate jump and branch instructions from arithmetic/logic and data transfer instructions. Furthermore, for our second goal, generating an operative static analyzer, we do not only need to generate transfer functions for each instruction (i.e., an abstract semantics), but also establish a flow relation for creating the

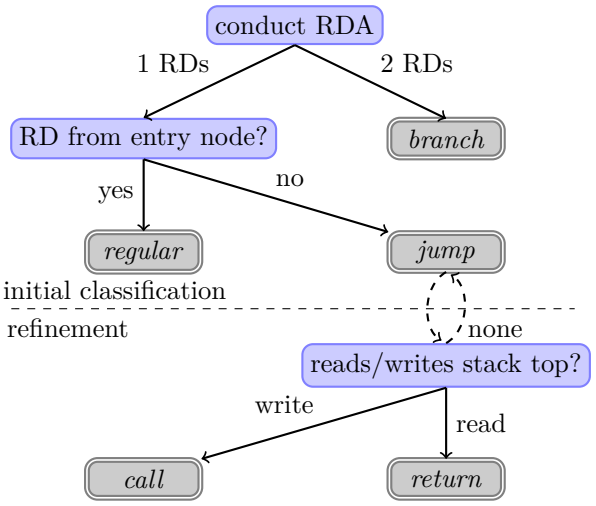

Figure 1 Classification strategy for control flow type.

control flow graph (CFG). The latter requires analyses to find out the number of succeeding instructions, and their addresses.

Both goals can be achieved by means of certain static analyses of the `execute` sections and subroutines in the SGDL description. As the language supports function calls, all analyses have to be conducted interprocedurally. Three analyses suffice: Reaching Definitions Analysis (RDA), Read Variable Analysis (RVA), and Written Variable Analysis (WVA). RDA is a standard textbook analysis [10], RVA is basically the collecting semantics of Live Variables Analysis (LVA) (i.e., an LVA with a constant empty *kill* function), and WVA is the counterpart of RVA with respect to written variables. The overall idea is to analyze write accesses to the program counter. Figure 1 illustrates the classification algorithm, which works as follows:

- Construct the control flow graphs for the current instruction and all called functions.
- All instructions implicitly increment the program counter by their own size, so insert one reaching definition (RD) into the entry node of the instruction CFG.
- Conduct the analyses.
- Classify instructions based on the number and origin of RDs reaching the exit node:
  - 1 RD from the entry node: regular instruction
  - 1 RD, but not from the entry node: program counter is inevitably overwritten with a single value, i.e., an unconditional *jump* instruction
  - 2 RDs: a conditional *jump*
- Refine the classification: *jump* instructions manipulating the stack could be *call* or *return* instructions, depending on whether they read / write the content of the program counter from / to the stack. Use RVA and WVA results to distinguish these.

The second step, obtaining the value written at runtime into the program counter, is part of the next section. Technically, it consists of two steps: first, use the RDA to locate assignments to the program counter, and second, backtrack and collect all subexpressions on the right hand side of such assignments. The resulting expression can then be evaluated at runtime on concrete instances of the instruction.

4.3 Analysis of Data Flow

An analysis of the data flow has to identify the effects of individual assignments to global variables. The goal of the analysis is to identify possible propagations of nondeterminism and also the opposite, variables that must not be nondeterministic when executing the instruction. Formally:

Let α be an instruction consisting of individual statements $\alpha_0, \ldots, \alpha_m$,

$$\begin{aligned}&\alpha_i \in \{\texttt{memidentifier}_\texttt{i}(\texttt{addr\_expr}_\texttt{i}) \leftarrow \texttt{expr}_i,\\ &\texttt{localvar\_identifier} \leftarrow \texttt{expr}_i,\\ &\texttt{call}\ \texttt{f}_\texttt{i}(\texttt{args}_\texttt{i})\},\end{aligned}$$

$$1 \leq i \leq m \quad (11)$$

Let $Addr_\alpha$ be the set of identifiers known to be used as an address within the scope of α, and initialize $Addr_\alpha := \emptyset$. Let $Var(expr)$ be the set of variables occurring in $expr$. Let $C \subseteq \mathbb{N} \times Addresses \times Addresses$ be a copy relation, wherein each entry is of the form $(instruction\ id, source, target)$.

▶ **Definition 5. Initial data flow analysis algorithm**
If α has been identified by the control flow algorithm as a jump, skip it. Else:

For all $\alpha_i \in \alpha$

- If $\alpha_i = \texttt{memidentifier}_\texttt{i}(\texttt{addr\_expr}_\texttt{i}) \leftarrow \texttt{expr}_i$:
  - $Addr_\alpha := Addr_\alpha \cup \texttt{Var}(\texttt{addr\_expr}_\texttt{i})$
  - $Addr_\alpha := Addr_\alpha \cup \texttt{Var}(\texttt{addr})$ for all addresses `addr` occuring in $\texttt{expr}_i$.
  - If $\texttt{expr}_i = \texttt{memidentifier}_j(\texttt{addr})$ for some memory identifier j and address `addr`, add an entry $(i, \texttt{memidentifier}_j(\texttt{addr}),$ $\texttt{memidentifier}_\texttt{i}(\texttt{addr\_expr}_\texttt{i})$ to the copy relation C
- If $\alpha = \texttt{call}\ \texttt{f}_\texttt{i}(\texttt{args}_\texttt{i})\}$: apply this algorithm to the called function to obtain a summary of function effects. Join resulting summary into analysis information of caller.

Next, *refine* the initial analysis results by collecting subexpressions referenced in expressions. The goal is to relate identifiers used as addresses back to the operands and global resources visible at the beginning of the instructions's `execute` block. This can be achieved by a backwards search through the CFG. In case the analysis should fail in this for a given expression (possible due to branches in the CFG, indicating the value for a subexpression is not unique), there are two possible continuations, depending on the type of the expression: if the expression is known to be used an an address, either in reading or in writing, we need to mark all identifiers used in the instruction for instantiation. Otherwise, if the expression is known to be used as the right-hand side (rhs) value in an assignment, we have to replace it by a value that is marked *completely nondeterministic*.

4.4 Synthesis of Abstract Semantics

Following completion of control and data flow analysis, we can generate abstract semantics for each instruction.

4.4.1 Delayed Nondeterminism

Instructions are considered as a list of individual assignments. For each of these, apply a translation rule. It is necessary to also add the original concrete semantics to the output

because it might be necessary, during execution, to revert to it, and in the process of that, instantiate all nondeterminism in the input. This can happen if at least one of the assignments tries to write to an address that has to remain always deterministic (cf. the requirements detailed in Sect. 4.1).

▶ **Definition 6. Translation rules for assignments**
Let $\alpha_i = \texttt{memidentifier}(\texttt{addr\_expr}) \leftarrow \texttt{expr}$.

Let $\texttt{Inst}_\alpha$ be the set of locations that have to be instantiated before executing α. Initially, $\texttt{Inst}_\alpha := \mathit{Var}(\texttt{addr\_expr})$.

Replace $\texttt{addr\_expr}, \texttt{expr}$ by the collected subexpressions, such that both expressions depend only on global variables and operands. If this is not possible because the analysis has failed, see below for a recovery strategy. Else, the abstract version $\widehat{\alpha_i}$ of α is defined by

- If the copy relation C created during analysis contains an entry $(i, \mathit{src}, \mathit{trgt})$, then $\widehat{\alpha_i} := [\mathit{val}(\mathit{src}) := \mathit{val}(\mathit{trgt}); \mathit{ndm}(\mathit{src}) := \mathit{ndm}(\mathit{trgt})]$
- Else (data is modified, so instantiate to generate concrete values)
  - $\texttt{Inst}_\alpha := \mathit{Inst}_\alpha \cup \mathit{Var}(\texttt{expr})$
  - $\widehat{\alpha_i} := \alpha_i$

Recovery strategy: for expressions whose composition cannot be analyzed, the obvious solution is to assign a nondeterministic value to the target. In case this is not desirable, for instance because the target is a frequently accessed or very wide register (i.e., many nondeterministic bits would be created), a fallback would be to instantiate every input of this instruction, and use the concrete semantics instead. Thus, no improved semantics is available for this instruction, but at least it is guaranteed that the abstraction would not actually *result in* state explosion instead of preventing it.

Using these translation rules yields the semantics of delayed nondeterminism. An obvious improvement concerns the condition for instantiation, which, in the above version, is *if any computation is performed, then instantiate all inputs.* Therefore, all arithmetic instructions will instantiate all of their inputs because they necessarily contain at least one such α_i. A more permissive condition exploits the computation rules for ternary logic:

- For all operators in the input language, i.e., $+, -, *, /, \ldots$, introduce new abstract versions $\widehat{+}, \widehat{-}, \widehat{*}, \widehat{/}, \ldots$. Semantics are those of their concrete counterparts, except that the abstract versions operate also on nondeterministic (n) bits. For instance, $0 \,\widehat{+}\, n = n \,\widehat{+}\, 0 = 1 \,\widehat{+}\, n = n \,\widehat{+}\, 1 = n$, $0 \,\widehat{*}\, n = 0, 1 \,\widehat{*}\, n = n$, and analogously for all other operators.
- For each rhs expression in an assignment, create an abstract syntax tree representation
- Conduct a tree pattern matching, as described by Aho et al. [1], and apply tree rewriting rules, to replace the operators in the expression by their abstract counterparts.

These advanced rules then leave only two cases for forced instantiation of all inputs: first, an address expression that cannot be discomposed into its components, and second, an attempted write to a location marked as *must always remain deterministic.*

4.4.2 Static Analysis

Using the results from the the control flow type analysis, it is possible to identify, for each instruction, the number of successors and their address, either absolute or relative. Therefore, given a program consisting of instances of these instructions, we can reconstruct the control flow graph from the disassembled binary representation of the program. Furthermore, the data flow analysis algorithm presented in the last section necessarily identifies read and

written memory locations, i.e., provides a starting point for generating transfer functions for analyses such as RDA and LVA.

[MC]SQUARE already provides a framework for static analysis, which can conduct analyses in case the developer provides a CFG and transfer functions for the named analyses. Therefore, the actual generation of an operative analyzer is reduced to the task of generating the necessary code from the existing analysis results.

5 Related Work

HOIST is a system by Regehr [12] that can derive static analyzers for embedded systems, in their case for an Atmel ATmega16 microcontroller. This is similar to our approach. The key difference is that they do not use a description of the hardware, but either a simulator or the actual device. For a given instruction that is executed on the microcontroller, HOIST conducts an exhaustive search over all the possible inputs, and protocols the effects on the hardware. These deduced transfer functions are then compacted into binary decision diagrams (BDDs), and eventually translated into C code. While this mostly automatic approach can provide very high accuracy in instruction effects, it certainly has the disadvantage of exponential complexity in the number of parameters for an instruction. Our approach does not depend on this, and is also automated, but the correctness of the results depends on the correctness of the description of the hardware. Moreover, HOIST is limited to analyzing ALU operations, whereas our analyzer, SGDL-STA, can analyze any kind of instruction.

Chen et al. [3] have created a retargetable static analyzer for embedded software within the scope of the MESCAL project [8]. Similar to our approach, they process a description of the architecture, which in their case is called a MESCAL Architecture Description (MAD). Automatic classification of instructions for constructing the CFG is apparently also possible in their approach, and they hint at that this is possible due to some attributes present in the MAD that allow identification of, for instance, the program counter. However, no further detail is provided on the ideas involved in classification. The generated analyzer is suitable for analyzing worst case execution time of certain classes of programs intended to run on the hardware.

Schlickling and Pister [15] also analyze hardware descriptions, in their case VHDL code. Their system first translates the VHDL input into a sequential program, before it applies well-known data flow analyses such as constant propagation analysis. These analyses are then used to prove or disprove worst case execution time properties of the hardware. In contrast to this, we concentrate on the way the resource model is altered by instructions, deliberately neglecting timing.

Might [9] focuses on the step from concrete to abstract semantics for a variant of lambda calculus. In their examples, they also relate their work to register machines, which, albeit a concept from theory, share some commonalities with real-world microcontrollers. They point out the similarities between the two semantics, and how to provide analysis designers with an almost algorithmic approach to lifting the concrete to the abstract. Hence, the foremost difference to our approach is that their contribution is certainly more flexible, as they rely on an expert. Compared to this, our approach is intentionally restricted to only a few abstractions, but for these, it is fully automated.

6 Conclusion

This paper shows that a single description of an instruction-set architecture, given as an implementation in a special-purpose imperative programming language, can serve as a starting point for generating several verification tools. We have shown how to switch from register transfer-level semantics based on concrete values to a partially symbolic technique, called delayed nondeterminism. To this end, we have described static analyses used on the imperative descriptions, by which the intention behind instructions becomes visible and ready for translation. Furthermore, these analyses can also be used to obtain a characterization of instructions needed for analyzing the code for the target platform.

The concepts developed in this contribution should be applicable not only to [MC]SQUARE and the SGDL system, but to any model checker interpreting assembly code. In order to verify the concepts, we have implemented a static analyzer for SGDL, called SGDL-STA. So far, we have successfully verified the ideas concerning classification of instructions into control flow classes. Classifying the instruction sets of both the ATmega16 and the MCS-51 microcontrollers can be achieved in less than 10 seconds. Additionally, we have used the analysis results for generating an operative static analyzer for the ATmega16 simulator, which enables a variant of Dead Variable Reduction [17] for this simulator. Hence, a direction for future work will be the implementation of the other concepts, especially the creation of the abstract semantics for delayed nondeterminism, and a comparison between the derived and the manually implemented versions of this abstraction technique.

Clearly, the results indicate that abstraction for hardware-dependent model checkers can, to a certain degree, be achieved automatically. Thus, it is not strictly necessary to have an expert in both model checking and embedded systems available, who is then to perform a fine-tuning of such tools. A practical implication of this improvement is that it might be possible for a non-expert to retarget a model checker to a new platform, at least in case the set of automatically derivable abstractions suffices. Therefore, we consider it necessary to conduct further research on other abstractions, and figure out to what extent it is possible to derive their semantics as well. Obvious directions for this include lifting the concrete semantics to interval semantics (i.e., the value of a memory cell is only known to be in an interval, instead of several distinct values), and easing our restrictions on nondeterministic control flow.

Acknowledgements We thank our former team member Christian Dehnert for his participation in our research, some of which was integrated into this contribution. Additionally, we thank the DFG Research Training Group 1298 "Algorithmic Synthesis of Reactive and Discrete-Continuous Systems" for funding parts of this work.

References

1 A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 2006.

2 C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

3 K. Chen, S. Malik, and D.I. August. Retargetable static timing analysis for embedded software. In *The 14th International Symposium on System Synthesis, 2001. Proceedings.*, pages 39 – 44, 2001.

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

5 P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL 77), Los Angeles, USA*, pages 238–252. ACM, 1977.

6 D. Gückel, J. Brauer, and S. Kowalewski. A system for synthesizing abstraction-enabled simulators for binary code verification. In *Industrial Embedded Systems (SIES 2010), Trento, Italy.*, 2010.

7 D. Gückel, B. Schlich, J. Brauer, and S. Kowalewski. Synthesizing simulators for model checking microcontroller binary code. In *13th IEEE International Symposium on Design & Diagnostics of Electronic Circuits and Systems (DDECS 2010), Vienna, Austria.*, 2010.

8 K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1523 –1543, 2000.

9 M. Might. Abstract interpreters for free. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 407–421. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-15769-1_25.

10 F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

11 T. Noll and B. Schlich. Delayed nondeterminism in model checking embedded systems assembly code. In *Hardware and Software: Verification and Testing (HVC 2007)*, volume 4899 of *LNCS*, pages 185–201. Springer, 2008.

12 J. Regehr and A. Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. *ACM SIGOPS Operating Systems Review*, 38(5):133–143, 2004.

13 B. Schlich. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany, June 2008.

14 B. Schlich, J. Brauer, and S. Kowalewski. Application of static analyses for state space reduction to microcontroller binary code. *Sci. Comput. Program.*, 76(2):100–118, 2011.

15 M. Schlickling and M. Pister. A framework for static analysis of VHDL code. In *Proceedings of 7th International Workshop on Worst-case Execution Time (WCET) Analysis*, 2007.

16 B. Titzer, J. Lee, and J. Palsberg. A declarative approach to generating machine code tools. Technical report, UCLA Computer Science Department, University of California, Los Angeles, USA, 2006.

17 K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. *Formal Methods in System Design*, 25(1):67–96, 2004.