Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 217 (2008) 133–150
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Proving Correctness of an Efficient Abstraction for Interrupt Handling

Gerlind Herberich,[1] Bastian Schlich,[1,2] Carsten Weise[1,3]

Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany

and

Thomas Noll[1]

Software Modeling and Verification Group
RWTH Aachen University
Aachen, Germany

Abstract

This paper presents an approach to the efficient abstraction of interrupt handling in microcontroller systems. Such systems usually operate in uncertain environments, giving rise to a high degree of nondeterminism in the corresponding formal models, which in turn aggravates the state explosion problem. Careful handling of nondeterminism is therefore crucial for obtaining efficient model checking tools. Here, we support this goal by developing a formal computation model and an abstraction method, called interrupt nondeterminism, which instantiates nondeterministic values only if and when this is required by the application code. It is shown how this symbolic technique can be integrated into our explicit CTL model checking tool [mc]square by introducing *lazy states*. A lazy state consists of explicit and symbolic parts and therefore, represents several concrete states. With regard to interrupt handling, we also give a simulation relation between the concrete and the abstract state space, thus establishing the correctness of our technique. Furthermore, a case study is presented in which three different programs are used to demonstrate the effectiveness of our method.

Keywords: Model checking, state space abstraction, simulation, interrupt handling

1 Introduction

Model checking is recognized by industry as a promising future tool for the analysis of embedded software (e.g., software for microcontrollers). Early model checkers as

[1] Email: "lastname"@cs.rwth-aachen.de

[2] partly funded by a DAAD-Doktorandenstipendium

[3] funded by the UMIC cluster of excellence: http://www.umic.rwth-aachen.de/

doi:10.1016/j.entcs.2008.06.046

SMV [4], Spin [8] and Uppaal [9] work on models described in their proprietary specification languages. Re-modeling existing systems in these proprietary formalisms is a huge effort. For existing systems, model checking of higher level programming languages (e.g., C, C++ or Java) is the more efficient approach, but when it comes to embedded software, many problems arise when model checking C programs.

Microcontroller programs written in C usually contain direct hardware accesses or embedded assembly statements. These constructs are not handled by existing C code model checkers (see [16]). Moreover, C code is first compiled into assembly code before it is deployed to the hardware. Hence, the C code is only an intermediate representation. The compiler could introduce errors that cannot be found in the original source code. In assembly code all errors introduced during the complete development process are present. Moreover, in contrast to C code, assembly code has a clean, formal and well documented semantics. Hence, model checking of assembly code (machine code) gets into focus of research, see [1,12,13,17,20].

However, when model checking assembly code state spaces tend to be bigger and the analysis is no longer hardware independent. In order to tackle these problems, we have developed [mc]square[4], which is a discrete, (mostly) explicit state, on-the-fly, Computation Tree Logic (CTL) [4] model checker. It is capable of model checking assembly code written for certain microcontrollers (ATMEL ATmega and Infineon XC167). We did not restrict the set of supported constructs. Hence, [mc]square can handle arbitrary assembly programs for those microcontrollers, supporting both low-level features such as direct or indirect memory access and source-level constructs such as recursion or functions. To address the disadvantage of being hardware-dependent, we developed an extensible architecture, which is described in [18]. To deal with the state explosion problem, we implemented different abstraction techniques in [mc]square. In this paper, we will show how prioritized interrupt levels can be used to abstract away from the concrete state of the interrupt bits of a microcontroller.

The basic idea of the abstraction technique is that in certain states of the processor, certain bits are irrelevant for the execution. These bits can be safely abstracted away by setting them to a *don't care* value, or better said to a *nondeterministic* value, which we will denote by $*$. This introduces so-called *lazy states*. A lazy state is a state that contains explicit and symbolic parts. As a consequence, a lazy state no longer represents a single state, but a set of states. Therefore, [mc]square combines explicit and symbolic techniques. This idea is used for several abstraction techniques within [mc]square, e.g., also for *delayed nondeterminism* (see [15]). In this paper, we focus on abstracting away from the specific values of the interrupt bits.

This paper is structured as follows: We start with the presentation of related work. Then, a basic introduction to [mc]square is given. Section 4 details nondeterminism for interrupts. The subsequent section presents our formal approach to modeling microcontrollers. As an example, the model of the ATMEL ATmega16 microcontroller is detailed. It is shown that nondeterminism of interrupts induces

[4] `http://www-i11.informatik.rwth-aachen.de/mc_square.html`

a simulation relation between the concrete trace of the system and the abstracted, nondeterministic traces, thus yielding an over-approximation of the real system behavior. However, while over-approximating the concrete behavior, it is true that for every trace in the abstract space, there is a concrete system which will exhibit this behavior. As interrupts depend on the behavior of an external environment which is not under the control of the processor, this means that all errors found in the over-approximation can in fact also be traced back to the real implementation. After that, a case study is summarized which demonstrates the effect of interrupt nondeterminism on the state space size. In the end a conclusion is drawn and some potential directions for future improvements are shown.

2 Related Work

Motivated by the observation that usually memory is the limiting factor in the application of model checking, many approaches have been developed to combat the state explosion problem (see [4] for an overview). The abstraction technique presented in this paper, *Interrupt NonDeterminism*, is dynamically applied at runtime. To the best of our knowledge, no comparable approach has been developed so far to control the effect of interrupts in modeling embedded systems.

There is, however, a verification method for concurrent systems called *narrowing* which is based on a similar idea, and which is described in [14]. Here, the states and transitions of the system are symbolically represented by terms and rewriting steps, respectively. Terms can contain variables to abstract from details of the system state which currently are not "interesting", but which can later be expanded by substitution steps if necessary. Thus, in some sense, variables correspond to the nondeterministic values in our approach.

Another direction of work which is worth mentioning is the concept of *lazy evaluation* in functional programming languages [10], which computes a function argument only if it is accessed in the function body. Moreover the paper [3] studies the implementation of nondeterministic choice in this setting and refers to the problem of copying nondeterministic values, which is also the reason for over-approximation in our model.

Symbolic simulation is another technique that is similar to the technique applied in [mc]square. Here, symbolic values are used in place of explicit values. In our approach parts of the states used can be symbolic, but whenever the simulator or the model checker needs to access symbolic parts of a state, these parts are instantiated, and hence, become explicit. All parts of a state that are not accessed remain symbolic. In [2], a symbolic simulator is used to verify hardware systems. Whenever an X (denoted by * in our approach) is accessed and a value is needed, new symbolic variables are added and simulation has to be repeated. In our approach a dynamic refinement is conducted. There are some approaches combining *explicit* and *symbolic executions* (cf. [6,22]), but these approaches employ explicit execution and symbolic execution in parallel.

Other model checkers that handle machine languages or languages that are similar to machine languages are *Java PathFinder* (*JPF*) [24], *StEAM* [11], and *Estes* [13], all being explicit model checkers as is [mc]square. JPF accepts Java bytecode and employs collapsing techniques for efficiently storing states. Our experiments have shown that such methods do not pay off in the case of [mc]square since its states have a less complex structure. Another difference is that JPF has to deal with concurrent processes and therefore employs abstraction techniques such as partial order reduction, which cannot be done in [mc]square. Moreover the memory model used within JPF makes it possible to apply symmetry reduction techniques. Again, this is not possible in [mc]square because the order of data within memory is important. StEAM model checks bytecode for the Internet C Virtual Machine. In this approach an existing Virtual Machine (VM) is monitored and model checking is conducted on the states created by this VM. Estes model checks assembly code for a certain processor. Similar to StEAM, it uses an existing VM (the GNU debugger) to create the state space. In our approach, we concentrate on the creation of the state space, that is, we concentrate on the domain-specific abstractions implemented within the simulator. We do not want to use existing simulators as we think that significant savings can be achieved by a tailored implementation. In contrast to Estes we abstract from time because the state explosion observed when temporal aspects are taken into account, i.e., real-time model checking (cf. [9]) is performed, is too big.

3 Introduction to [mc]square

This section gives an introduction to [mc]square, which is a discrete, (mostly) explicit state, on-the-fly, CTL model checker. “Mostly explicit” means that [mc]square uses explicit model checking algorithms, but combines them with symbolic techniques. We call the new states that combine explicit and symbolic parts *lazy states*. Whenever a symbolic part of a lazy state is accessed, the nondeterminism is resolved automatically. [mc]square works on assembly code written for certain microcontrollers (ATMEL ATmega and Infineon XC167). More information about [mc]square can be found in [17,18].

The process that is applied in [mc]square is depicted by Fig. 1. First, the user inputs the program as an Executable and Linking Format (ELF) file and the specification as a CTL formula. If the C code is available, the user can also provide the corresponding file. The formula is parsed and transformed into a formula object, which is utilized by the static analyzer and the model checker component. The ELF file is preprocessed and converted into an human readable assembly program.

Then, the static analyzer component starts inspecting the assembly program. During this analysis, it uses information from the formula object (registers, variables and memory locations used within the atomic propositions) to preserve validity of the results. In the first step of the static analysis, a Control Flow Graph (CFG) of the assembly program is created. This CFG is inter alia used by the counterexample generator to present counterexamples or witnesses. In the end, the static analyzer adds annotations to the assembly program which are used by the simulator to reduce

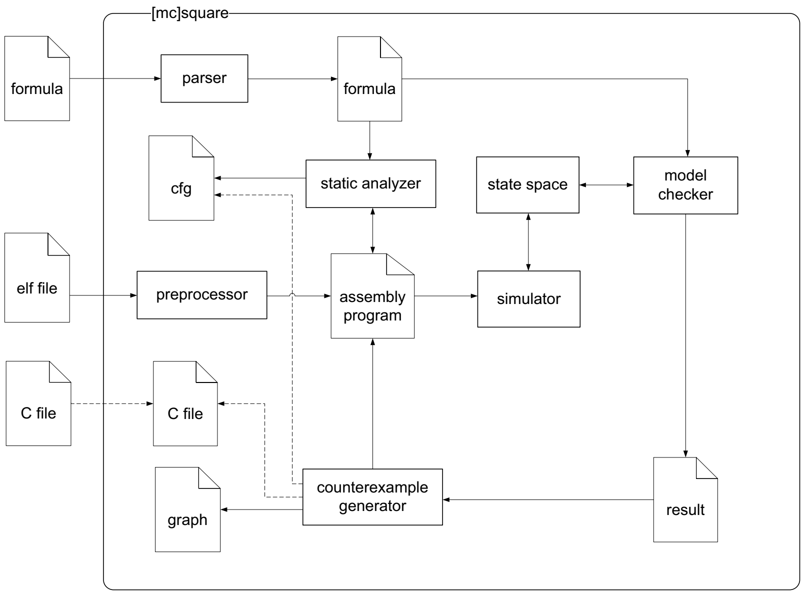

Figure 1. Process used in [mc]square

the size of the state space.

After that, model checking starts. First, the model checker requests the initial state from the state space generator. It checks this state for certain parts of the formula, and depending on the result of this check, it requests successor states of this state. Then, it again checks these states for specific parts of the formula. This process continues until a goal state is reached (proving or disproving the validity of the formula) or the complete state space is built. The model checking algorithm used is taken from [7]. A first version of this algorithm was presented in [23].

Whenever successors of a state are requested that are not created yet, the state space generator uses the simulator to on-the-fly create the needed states. To do so, it passes the current state to the simulator and calls a `step()` method. The simulator creates all possible successors of this state including, e.g., occurrences of interrupts, different input values from the environment etc. If, e.g., an instruction `IN R18 PINA` reads input from the environment into register `R18`, then all possible values might occur, which results in 256 successors states (all values between 0 and 255). Also, if a specific interrupt is enabled in a state of the system, then the interrupt might potentially occur, and thus both successor states – one taking the interrupt, one with the usual program execution step – must be considered by [mc]square.

As the last step of analysis, the counterexample generator derives a counterexample or a witness depending on the formula checked and the result of the model checking process. This counterexample/witness is then presented in the assembly code, in the C code, as a state space graph, or in the CFG of the assembly code. Hence, the user can choose the representation that suits his requirements best to find the error. As some abstraction techniques used in [mc]square only preserve

	i0	i1	i2	i3
source active	+	+	+	+
interrupt enabled	+	+	-	+
interrupt flag	*	*	*	*

Table 1
Four interrupts exemplifying nondeterministic interrupts.

ACTL[5], the user can use this representation to check whether a witness of an non-ACTL formula is a feasible one.

During state space building [mc]square uses different abstraction techniques to minimize the size of the state space. It is important to notice that all these abstractions lead to a safe over-approximation of the concrete state space, establishing a simulation relation between the concrete and the abstract states (and thus preserving the validity of ACTL formulae). One of these abstraction techniques is interrupt nondeterminism, which is detailed in this paper.

4 Interrupt Nondeterminism

Interrupt nondeterminism abstracts away from interrupts which are below the current interrupt level to reduce the state space for interrupt handling. Whenever an interrupt at level ℓ is taken, all interrupt flags for levels below ℓ can be set to the nondeterministic value $*$. When not using the abstraction, there are 2^s different possible combinations where s is the number of active interrupt sources. Abstraction reduces this number to $a - 1$ where $a \leq s$ is the number of enabled interrupts, which yields a drastical saving in terms of successor states.

However, even when interrupts are disabled or masked, it still is possible to check their values, or enable them again. In this cases, the analysis again must consider the actual values of the interrupt bits. Thus, we need to consider that interrupt bits are tested or set, and then make their value explicit again. This step is called the *instantiation* of a bit, and can occur in two ways: if the bit is tested, the simulation needs to take both cases into account: that its value is 0 or 1. Two successor states must be created, one for each possibility – or even more, if several bits are tested in parallel. If the bit is set to a specific value, then the bit is just set to this value in the successor state.

In Tab. 1 an example configuration of four interrupts is shown. The sources of all four interrupts are active, but only three of these interrupts are enabled. In a concrete simulation, all 16 value combinations shown in the left part of Tab. 2 are written to the interrupts flag registers. In contrast, only 4 combinations are written to the interrupt flag registers when using interrupt nondeterminism. These four combinations are shown in the right part of Tab. 2.

Note that if an interrupt bit occurs in the formula tested by the model checker, then that bit must be excluded from interrupt nondeterminism. This is a simple implementation issue not detailed in this paper.

[5] the universal fragment of CTL; see [5]

i0	i1	i2	i3	
0	0	0	0	→ no interrupt
0	0	0	1	→ interrupt 3
0	0	1	0	→ no interrupt
0	0	1	1	→ interrupt 3
0	1	0	0	→ interrupt 1
...	...	...	...	...
1	1	1	0	→ interrupt 0
1	1	1	1	→ interrupt 0

i0	i1	i2	i3	
1	*	*	*	→ interrupt 0
0	1	*	*	→ interrupt 1
0	0	*	1	→ interrupt 3
0	0	*	0	→ no interrupt

Table 2
Deterministic and nondeterministic evaluation of interrupts.

Here, we described handling of interrupts using a static interrupt priority table for the ATMEL ATmega microcontrollers. Nondeterministic interrupts also work using a dynamic interrupt priority table because at the time of interrupt handling, the priorities are fixed as the interrupt table can only be changed by instructions. Moreover whenever interrupts are handled, no instruction is executed at that moment.

5 The Formal Model

Delayed nondeterminism (DND) has been presented in [15] first, where it is proved that DND preserves a simulation relation. Here, we show that the restriction of this technique to interrupt handling, *interrupt nondeterminism*, also preserves simulation. Thus it is sound with respect to "path-universal" logics such as ACTL.

5.1 Basics of the Model

Our formal proof for the correctness of the nondeterministic approximation of interrupt handling uses a formal model defining the behavior of our system by a semantic function. The intuition behind the formal model is described in the section.

The model works on the following basic data types:

- the basic data in the model are bits $\mathbb{B} := \{0, 1\}$ and bytes $\mathbb{C} := \mathbb{B}^8$,
- to model nondeterminism, we use the "don't care" bit $*$, and extend the set of bits by this value: $\mathbb{B}_* := \mathbb{B} \cup \{*\}$, and similarly $\mathbb{C}_* := \mathbb{B}_*^8$.

The microcontroller consists of a control and a data space:

- The data space is modeled by an ordered set of addresses A, and an association of (nondeterministic) byte values to these addresses $v : A \to \mathbb{C}_*$. Here we assume w.l.o.g. that each address comprises m bytes, that is, $A = \{0, \dots, 2^{8m} - 1\}$. The set of all data spaces is denoted by V.
- As we often need to select specific bits from bytes, we use $A_{bit} := A \times \{0, \dots, 7\}$ as the address space for bits, and then interpret v as $v : A_{bit} \to \mathbb{B}_*$, where $v(a) = v(a, 7) \dots v(a, 0)$.
- The control (code) space is modeled by a set of locations Q, which is a linearly ordered set, and a mapping of the locations to instructions `Ins`. More details on

instructions can be found below.

The above model is a straightforward implementation of a Harvard Computer, where we allow certain bits to be *nondeterministic* in the model, i.e., in a specific snapshot we do not know the status of certain bits.

We will now define a transition relation, which describes the behavior of this machine. The basic idea is that the machine cycles through instructions, and that each instruction defines the read- and write accesses to the memory as well as the control location of the next instruction (which is the following location in the normal case, but a different one in case of jumps).

The machine we describe is used to control some environment. The communication with the environment happens via ports, which are lines sending bits into the environment or receiving bits from the environment. We assume the usual memory-mapped I/O, so the ports are special addresses within the memory of the machine. The setting of the input ports is described by an environment specification `env`, which is a sequence of guarded assignments to the ports. The guarded assignments are executed each time before an instruction is carried out.

Further, the machine has a set of interrupt routines. The interrupt routines are described by the *interrupt handler* `IH`, which is a sequence of guarded assignments, where each guarded assignment has a continuation, i.e. a location to which the machine will jump when the interrupt routines are triggered. The guarded assignments describe the interrupt logic of the machine: the guard specifies when the interrupt is taken (e.g. when a certain interrupt is enabled and the interrupt flag is set) and the assignments model the interrupt logic, e.g. storing of the current program counter on the system stack happens here. Further, the interrupt routines are prioritized, i.e. they are executed in a specific order from highest to lowest priority, and the routine with the highest priority and an enabled guard is executed first.

The interrupt handler is executed after the environment specification and before the respective instruction. This way, a synchronized, clock-cycle triggered execution of the machine is guaranteed.

We now define the syntax and semantics of assignments, guards and expressions. An assignment takes the form $x \leftarrow e$, where x is an *address expression* and e is a *value expression*. Let Z bet the set of all assignments.

For address expressions, we have $x ::= a \mid a\!\downarrow + d \mid a[b]$ where $a \in A$, $b \in \{0, \ldots, 7\}$, and $d \in \mathbb{Z}$. The first case is a direct byte access, the second is an indirect byte access with displacement d, the third is accessing the bth bit at address a. Let X be the set of all address expressions.

A value expression is either an address expression, or an operation on value expressions $e ::= op(e_1, \ldots, e_n)$. Possible domains are $\mathbb{B}$ and $\mathbb{C}$ both for argument and result. Let E be the set of all value expressions.

A guard is a test on the equality of two value expressions, boolean combinations thereof or negation: $g ::= e_1 == e_2 \mid g_1 \wedge g_2 \mid g_1 \vee g_2 \mid \neg g_1$, where e_i are value expressions and the g_i are guards. Let G be the set of all guards.

5.2 Semantics in the Deterministic Case

We can now give a semantics for the behavior of the microcontroller for the deterministic case.

The semantics $[\![e]\!] : V \to \mathbb{C} \cup \mathbb{B}$ of a value expression is defined by $[\![a]\!](v) := v(a)$, $[\![a{\downarrow} + d]\!](v) := v(a' + d)$ where a' denotes the address which is referred to by the m bytes stored at a: $a' := v(a) \cdot 2^{8(m-1)} + \ldots + v(a + m - 1) \in A$, $[\![a[b]]\!](v) := v(a, b)$, and $[\![op(e_1, \ldots, e_n)]\!](v) = op([\![e_1]\!](v), \ldots, [\![e_n]\!](v))$.

Let $v\{a/e\}$ be the valuation v' where $v'(a) = e$ and $v'(a') = v(a')$ else. Then assignments have the semantics $[\![x \leftarrow e]\!] : V \to V$ where $[\![a \leftarrow e]\!](v) = v\{a/[\![e]\!](v)\}$, $[\![a{\downarrow}+d \leftarrow e]\!](v) = v\{a'+d/[\![e]\!](v)\}$ where again $a' := v(a) \cdot 2^{8(m-1)} + \ldots + v(a+m-1)$, and $[\![a[b] \leftarrow e]\!](v) = v\{(a, b)/[\![e]\!](v)\}$ (assuming type correctness).

Guards have type $[\![g]\!] : V \to \mathbb{B}$, and are defined as $[\![e_1 == e_2]\!](v) = 0$ if $[\![e_1]\!](v)$ not equal $[\![e_2]\!](v)$, and $[\![e_1 == e_2]\!](v) = 1$ else. The boolean operators work as usual: $[\![g_1 \wedge g_2]\!](v) = [\![g_1]\!](v) \wedge [\![g_2]\!](v)$ etc.

On sets, we use the usual operations $*$ and $+$ meaning all finite sequences resp. all non-empty finite sequences. Let $GA := G \times Z^*$ be the set of guarded assignments, and $GAC := GA \times Q$ the set of guarded assignments with a continuation.

A guarded assignment is enabled when its guard is 1. The semantics of an enabled guard assignment is the execution of the assignment, otherwise the guard assignment does not change the microcontroller's memory. Formally,

$$[\![g : asn]\!](v) = \begin{cases} v & \text{if } [\![g]\!](v) = 0 \\ [\![asn]\!](v) & \text{if } [\![g]\!](v) = 1 \end{cases}$$

The Environment handler `env` $\in GA^+$ has the form $g_1 : asn_1;\ \ldots\ ; g_n : asn_n$. Its semantics is the consecutive execution of the enabled assignments:

$$[\![g_1 : asn_1;\ \ldots\ ; g_n : asn_n]\!](v) = [\![g_n : asn_n]\!]([\![g_1 : asn_1;\ \ldots\ ; g_{n-1} : asn_{n-1}]\!](v)),\ n \geq 2$$

A guarded assignment with continuation induces a mapping $[\![.]\!] : Q \times V \to Q \times V$, defined as

$$[\![g : asn, q']\!](q, v) = \begin{cases} (q, v) & \text{if } [\![g]\!](v) = 0 \\ (q', [\![asn]\!](v)) & \text{if } [\![g]\!](v) = 1 \end{cases}$$

The interrupt handler `IH` $\in GAC^+$ has the form `IH` $= g_1 : asn_1, q_1\ >\ \ldots\ >\ g_n : asn_n, q_n$. We say that an interrupt is *triggered* if its guard is enabled. The function *trigger* $: GAC^+ \to \mathbb{N}$ defines which interrupt is taken:

$$trigger(\mathtt{IH}) = \begin{cases} 0 & \text{if } \forall i.[\![g_i]\!](v) = 0, \\ j & \text{if } [\![g_j]\!](v) = 1, \forall i < j.[\![g_i]\!](v) = 0 \end{cases}$$

and then $[\![\mathtt{IH}]\!](v) = [\![g_j : asn_j, q_j]\!](q, v)$ if $trigger(\mathtt{IH}) = j$.

The instruction handler `Ins` $\in Q \times GAC^+$ has the form $q :: g_1 : asn_1, q_1\ >\ \ldots\ >\ g_n : asn_n, q_n$, and will also execute the first enabled assignment:

$$\begin{aligned}&[\![q :: g_1 : asn_1, q_1 \ > \ \ldots \ > \ g_n : asn_n, q_n]\!](v) := \\ &\quad [\![g_1 : asn_1, q_1 \ > \ \ldots \ > \ g_n : asn_n, q_n]\!](q, v)\end{aligned}$$

The microcontroller MC is then the tuple $(A, Q, \texttt{Ins}, \texttt{env}, \texttt{IH}, q_0, v_0)$ with start state $q_0 \in Q$ and initial memory $v_0 \in V$. It defines a transition relation on $Q \times V$ by $(q, v) \to (q', v')$ with $v_1 = [\![\texttt{env}]\!](v)$,

$$(q', v') = \begin{cases} [\![\texttt{IH}]\!](q, v_1) & \text{if } \mathit{trigger}(v_1) > 0 \\ [\![\texttt{Ins}]\!](q, v_1) & \text{if } \mathit{trigger}(v_1) = 0 \end{cases}$$

and initial state (q_0, v_0). We will call this behavior of MC the *concrete* behavior.

5.3 Nondeterministic Interrupts

When modeling all possible behaviors of an environment, we would need to take into account all possible settings of the interrupt bits at any time. However, when an interrupt of level ℓ is taken, the lower prioritized interrupts are not important anymore. Thus we can reduce the number of states in the model checker by replacing all interrupt bits with a lower priority by the nondeterministic bit $*$.

To formalize this, we now introduce the nondeterministic transition relation $\to_{nd}$. This will be done by lifting the deterministic transition relation to nondeterministic evaluations $v : A \to \mathbb{C}_*$.

We start with introducing nondeterminism into our model, using the function $\text{ndet} : (A_{bit} \to \mathbb{B}_*) \times 2^{A_{bit}} \to (A_{bit} \to \mathbb{B}_*)$, which increases the level of nondeterminism:

$$\text{ndet}(M, v) := w \text{ where } w(a, b) := \begin{cases} v(a, b) & \text{if } (a, b) \notin M \\ * & \text{if } (a, b) \in M \end{cases}$$

Assume we have interrupt levels $1, \ldots, n$. Then we have also a set of interrupt bits $IB_1, \ldots, IB_n \in A_{bit}$. Let $\text{IL}(\ell) := \{IB_1, \ldots, IB_{\ell-1}\}$. Let $\texttt{IH}$ be some interrupt handler with $[\![\texttt{IH}]\!](q, v) = (q, v')$ in the deterministic case. Then $[\![\texttt{IH}]\!]_{nd}(q, v) := (q', \text{ndet}(\text{IL}(\ell), v')$ in the nondeterministic case, if $\mathit{trigger}(\texttt{IH}) = \ell > 0$, and thus interrupt nondeterminism is introduced into the model at this point.

The next step is to lift the semantics of expressions, assignments and guards to nondeterministic valuations, where all bits which are important for the evaluation are still deterministic. To formalize this notion, we defined the set of bits which are read in an expression, assignment, resp. guard. Let tested : $A \cup E \cup Z \cup G \to 2^{A_{bit}}$

be defined as

$$\begin{aligned}
\text{tested}(a) &:= \{(a,0),\ldots,(a,7)\}\\
\text{tested}(a{\downarrow}+d) &:= \text{tested}(a) \cup \text{tested}(v(a))\\
\text{tested}(a[b]) &:= \{(a,b)\}\\
\text{tested}(op(e_1,\ldots,e_n)) &:= \text{tested}(e_1) \cup \ldots \cup \text{tested}(e_n)\\
\text{tested}(a \leftarrow e) &:= \text{tested}(e)\\
\text{tested}(e_1 == e_2) &:= \text{tested}(e_1) \cup \text{tested}(e_2)\\
\text{tested}(e_1 \wedge e_2) &:= \text{tested}(e_1) \cup \text{tested}(e_2)\\
\text{tested}(e_1 \vee e_2) &:= \text{tested}(e_1) \cup \text{tested}(e_2)\\
\text{tested}(\neg e) &:= \text{tested}(e)
\end{aligned}$$

Obviously, only the bits in tested(e) are needed to evaluate the expression e for some valuation v. So for valuations v which are deterministic in tested(e), the above definitions can be re-used without changes. Let $\rightarrow_x$ be the transition relation induced by this.

Now for a valuation which is nondeterministic in bits needed for evaluation, we will simply instantiate the necessary bits to all possible values. This is formalized by the function: $\text{det} : 2^{A_{bit}} \times (A_{bit} \rightarrow \mathbb{B}_*) \rightarrow 2^{(A_{bit} \rightarrow \mathbb{B}_*)}$, which is capable of decreasing the nondeterminism of a valuation:

$$\begin{aligned}
\text{det}(M,w) := \{v \in A_{bit} \rightarrow \mathbb{B}_* \mid \forall (a,b) \in M. v(a,b) \neq * \text{ and}\\
\forall (a,b) \in A_{bit}. w(a,b) \neq * \implies v(a,b) = w(a,b)\}.
\end{aligned}$$

Further, nb(w) is the set of nondeterministic bits in w, i.e.

$$\text{nb}(w) := \{(a,b) \mid w(a,b) = *\}.$$

Now a semantic function ϕ with $[\![e]\!](v) = v'$ can be extended to a nondeterministic valuation w by setting $[\![e]\!]_{nd}(w) := \{[\![e]\!](v) \mid v \in \text{det}(M,w)\}, M = \text{tested}(e) \cap \text{nb}(w)$, and similarly $[\![e]\!]_{nd}(q,w) := \{[\![e]\!](q,v) \mid v \in \text{det}(M,w)\}$. Note that our semantics now yields a set of valuations as result.

The nondeterministic transition function can then be defined as $(q,w) \rightarrow_{nd} (q',w')$ iff $(q',w') \in [\![MC]\!]_{nd}(q,w)$.

Note that we can lift the definition of tested to the transition relation by the following rules, yielding a notion of tested for our transition relation.

- For consecutive execution of statements, the tested sets need to be joined,
- for statements with priority, only take into account the statements until the first executed.

So whenever we have a deterministic transition $(q,v) \rightarrow (q',v')$, there is a set $M \subseteq A_{bit}$ of the bits referred to by the transition. For all $N \subseteq A_{bit} \backslash M$, the valuation

$\hat{w} := \mathrm{ndet}(N, v)$ has all the necessary information for the deterministic relation, and thus we can also apply the definition, yielding a transition $(q, \hat{w}) \rightarrow_x (q', \hat{v})$. As this transition results from reading and writing the same bits as for $(q, v) \rightarrow (q', v')$, there must be $N' \subseteq N$ and $\hat{v} = \mathrm{ndet}(N', v')$. It must not be $N = N'$, as nondeterministic bits can be assigned a value by the transition.

5.4 Modeling the ATMEL ATmega16

Now we will show how to use the general framework to model the ATMEL ATmega16 microcontroller. The ATMEL ATmega16 has a 16K flash memory for program code, which corresponds to the location set Q. All special purpose register of the ATMEL are embedded into the data space, which has an address length of $m := 2$. As we focus on interrupt handling here, we will briefly describe the interrupt handling of the ATMEL and identify the special purpose registers within A that are essential for the interrupt handling.

The ATMEL has 21 different interrupts, each on an interrupt level of its own. The interrupts range over a non-maskable reset, externally generated interrupts, timer interrupts and internally generated interrupts caused, e.g., by completion of specific operations.

Each interrupt apart from the non-maskable reset can only occur when at least three conditions are met:

- interrupts are globally enabled, i.e., the *global interrupt enable* flag `I` in the status register `SREG` must be set,
- the enable interrupt bit of the specific interrupt must be set,
- the interrupt flag of the specific interrupt must be set.

The enable interrupt bit and the interrupt flag are found in different registers depending on the type of the interrupt. For timer interrupts, they are stored in the `TIMSK` and `TIFR` registers, for external interrupts in the `GICR` and the `GIFR` registers, and similar for the rest of the interrupts.

In order to model check real systems, where we have no control over the environment of our microcontroller, we must assume that input ports can have any value, and that interrupts might occur at any time.

This is done by introducing nondeterminism for certain bits in our storage, namely for the input port bits of our system and the interrupt flags. The environment `env` is used to set these bits to $*$ before an instruction is carried out. Due to space limitations, we cannot give the full definition of the environment for the ATMEL, but we will illustrate it by an example with one of the built-in timers and with one of the external interrupts.

For the example, we use the function $nd : \mathbb{B}_* \rightarrow \mathbb{B}_*$ defined as $nd(0) := *, nd(1) = 1, nd(*) = *$. If an interrupt flag is already set, then this function leaves it set. If the interrupt flag is not set, or nondeterministic, then it could be set in this step, and thus must be made nondeterministic.

A timer interrupt for Timer 0 can only occur if the clock for Timer 0 is selected.

The mode of the timer is set via the CS02/CS01/CS00 bits of the TCCR0 register.

The external interrupt 2 can only occur if a certain port is selected as the source for this interrupt. The port's setting is controlled via bit DDB in the port register DDRB. Thus we have:

$$\begin{aligned}&\texttt{TCCR0}[\texttt{CS02}] == 1 \lor \texttt{TCCR0}[\texttt{CS01}] == 1 \lor \texttt{TCCR0}[\texttt{CS00}] == 1 :\\&\quad \texttt{TIFR}[\texttt{TOV0}] \leftarrow nd(\texttt{TIFR}[\texttt{TOV0}]);\\&\texttt{DDRB}[\texttt{DDB2}] == 0 : \texttt{GIFR}[\texttt{INTF2}] \leftarrow nd(\texttt{GIFR}[\texttt{INTF2}]); \ldots\end{aligned}$$

As said before, the interrupt only occurs if the global interrupt enable bit in the status register, the bit in the interrupt mask and the interrupt flag are set. The following interrupt handler checks this first for the timer interrupt, then for the external interrupt.

$$\begin{aligned}&\texttt{SREG}[\texttt{I}] = 1 \land \texttt{TIMSK}[\texttt{TOIE0}] = 1 \land \texttt{TIFR}[\texttt{TOV0}] = 1 :, 18\downarrow\ >\\&\texttt{SREG}[\texttt{I}] = 1 \land \texttt{GICR}[\texttt{INT2}] = 1 \land \texttt{GIFR}[\texttt{INTF2}] = 1 :, 36\downarrow > \ldots\end{aligned}$$

As can be seen, interrupt vectors are stored at memory locations 18 and 36 respectively, and the interrupt at 18 has priority over the other one.

5.5 The Simulation Proof

In order to reduce the effort in model checking which comes from introducing nondeterminism, we will show that for the interrupts, it is safe to make all interrupt bits below the current interrupt level nondeterministic. Assuming three interrupt levels stored in the lower three bits of the interrupt register, this means we can identify all states where these bits have the values $100, 110, 101, 111$, which is a reduction of a factor 4 for the interrupt handling. In the general, we gain a factor or 2^n for n interrupt levels, so for the ATMEL we gain a factor of up to 2^{20}.

To show this, we look at a fixed microcontroller $MC = (A, Q, \texttt{Ins}, \texttt{env}, \texttt{IH}, q_0, v_0)$. The interrupt handler and the environment handler are defined as sketched above, thus implementing the interrupt handling model of the ATMEL. The instruction space can hold an arbitrary program.

This MC induces a deterministic transition system $\rightarrow$ and a nondeterministic transition system $\rightarrow_{nd}$. Intuition tells us that the deterministic transition system is the real behavior of the microprocessor when running the loaded program, and if the environment behaves as modeled. The nondeterministic transition system in contrast models how the simulation of the processor is implemented in [mc]square.

To show that the nondeterministic transition system is similar to the deterministic one, formally we show that there is a simulation relation $\leq_\sigma$: $(Q \times V) \times (Q \times V_{nd})$ such that whenever $s \leq_\sigma s_{nd}$ and $s \rightarrow s'$, then there is s'_{nd} such that $s_{nd} \rightarrow_{nd} s'_{nd}$ and $s' \leq_\sigma s'_{nd}$.

The relation for which we show the result is just relating states which are more nondeterministic, i.e. for which there is a set $M \subseteq A_{bit}$ such that $w = \mathrm{ndet}(M, v)$. For those we claim that $(q, v) \leq_\sigma (q, w)$ is a simulation of (q, w) by (q, v).

So assume we have $(q, v) \to (q', v')$. Further let M such that $w = \mathrm{ndet}(M, v)$. If there is no interrupt involved, then we have $(q, w) \to_{nd} (q', w')$ for some M, N with $w = \mathrm{ndet}(M, v), w' = \mathrm{ndet}(N, v')$. By definition of $\leq_\sigma$, we have $(q', v') \leq_\sigma (q', w')$.

If an interrupt at level ℓ occurs, then we have $(q, w) \to_{nd} (q', w')$ for some M, N with $w = \mathrm{ndet}(M, v), w' = \mathrm{ndet}(\mathrm{IL}(\ell), \mathrm{ndet}(N, v'))$. Again, obviously we also have $(q', v') \leq_\sigma (q', w')$.

So nondeterministic interrupts are an overapproximation of the concrete system, simulating all behaviours of the concrete system. Additionally, as any value for an interrupt can occur at any time, it is also clear that all states reachable in the nondeterministic system are reachable in the deterministic system.

6 Case Study

This case study was first described in [15]. Here, we summarize the important details and reconsider the results under the aspect of DND for interrupts. In [15] only DND for values was considered. DND for interrupts was activated in all runs.

The case study was conducted on a laptop equipped with a Intel Core Duo CPU at 2.33 GHz, 4 GB main memory, and a hard disk with a capacity of 100 GB. [mc]square is completely written in Java, and hence, every operating system can be used. All programs used in this case study were developed by students during lab courses, exercises, diploma theses, or their working time. None of these programs was intentionally written to be model checked. All programs were run on the ATMEL ATmega16 microcontroller. Details about this case study can be found in [15].

As DND for interrupts cannot be deactivated in [mc]square because its operation is essential for the model checking of programs using interrupts, we can only show the differences obtained by DND of values. These three programs all use interrupts. Without using DND for interrupts the number of states would be considerably higher and model checking of these programs would not be possible.

Table 3 presents the outcome of this case study for the three programs. The first column shows the name of the program. In the second column it is indicated which abstraction techniques were used (here: *DND for interrupts only*, *DND for interrupts and for values*, and all abstraction techniques). The option all abstraction techniques includes DND, *path reduction*, and *dead variable reduction*. The column *# states stored* represents the number of different states stored in the state space. In contrast, the column *# states created* shows the number of all states created during building of the state space, including revisits. *Size [MB]* gives the size of the state space in main memory, and *Time [s]* shows the total time needed for building the state space including all preparatory steps (e.g., preprocessing, parsing, and static analyses) and model checking the formula **AG** true. We chose this formula because it builds the complete state space, and it does not influence the abstraction techniques. In case a formula is chosen that makes an assumption about a certain memory location (e.g., register, I/O register, or variable), the abstraction techniques would no longer work on this memory location. As the formulas are different for

Program	**Options used**	**# states stored**	**# states created**	**Size [MB]**	**Time [s]**
plant	DND IR	801,616	854,203	240	23.19
	DND	188,404	195,955	57	4.39
	all	11,524	222,636	3.5	3.02
traffic light	DND IR	35,613	38,198	10	0.78
	DND	10,004	10,520	2.73	0.24
	all	523	13,069	0.21	0.17
window lift	DND IR	10,100,400	11,196,174	2,049	416.98
	DND	323,450	444,191	96	9.09
	all	10,699	463,129	3.26	7.43

Table 3
Effect of delayed nondeterminism on the state space size.

each program, the influence on each program would be different. Therefore, a fair comparison of the state space sizes and the effect of the abstraction techniques would not be possible.

We can give some comments about the size of the state space when not using DND of interrupts. When using DND for interrupts, only enabled interrupts are fired by writing only possible value combinations into the flag registers. When not using DND for interrupts, all interrupts would be fired that have an active interrupt source by writing all value combinations into the flag registers.

The *plant* program consists of 225 lines of assembly code and uses two interrupts and one timer. The *traffic light* consists of 155 lines of assembly code and uses the same number of timers and interrupts as the *plant* program. The *window lift* has 289 lines of assembly code and uses again two interrupts and one timer. As all program use the same number of timers and interrupts, we only detail one of them.

The *plant* program uses one timer interrupt and one external interrupt. When using DND for interrupts, at most three combinations are written to the flag registers: timer interrupt occurred, external interrupt occurred, and no interrupt occurred. This is only done when the corresponding interrupts are enabled. When not using DND for interrupts more combination are written. In this case all interrupts are fired that have an active interrupt source. The interrupt source for the timer used in the *plant* program is actually the source for two different timer interrupts. The second timer interrupt is not used in this program but without using DND for interrupts, it would be fired. As all value combinations are written into the flag registers, at least nine combinations would be written. These nine combination would be created in every line of the program where the sources of the interrupts are active. The sources are active in almost all parts of the program including interrupt routines (in interrupt routines, other interrupts are usually deactivated). If nondeterminism of values is involved (e.g., input from the environment) additionally, the number of resulting states from these values would be multiplied with the number of interrupt value combinations. The same notes apply for the program *traffic light* and *window lift*.

DND has a major influence on the size of the state spaces. The influence of DND for values can be seen in Tab. 3 and is described in [15]. The influence of DND for interrupts cannot be seen in the table. Before we implemented DND for interrupts, we could hardly check a program using more than one interrupt. Now, we can usually check programs with up to five interrupts. DND for interrupts typically has a bigger influence on the size of the state space than DND for values has. In the previous paragraph, we gave an impression about the sizes of state spaces of these three programs when not using DND. [21] presents a case study where we model checked microcontroller programs, which were used to do a speed measuring for a model car. These programs used up to 5 interrupts and had up to 5000 lines of assembly code. Without using DND, [mc]square was not able to model check these programs. Using DND it was possible to model check them.

7 Conclusion & Future Work

In this paper delayed nondeterminism for interrupts, which is an abstraction technique implemented in [mc]square, was detailed and it was proven that DND for interrupts preserves a simulation relation. This is an important result as DND for interrupts cannot be deactivated by the user because this abstraction technique is too essential for [mc]square. Without this abstraction techniques, even small programs using more than one interrupt could not be model checked. As [mc]square is a CTL model checker, simulation is needed to preserve the validity of formulas. The DND of values preserves a simulation relation (see [15]) and hence, the validity of ACTL formulas is preserved. Nevertheless, DND of values can be deactivated by the user if the over-approximation is too coarse.

DND is an abstraction technique that introduces lazy states into [mc]square. A lazy state is a state that is mostly explicit but has symbolic parts. These symbolic parts remain symbolic until they are required in a computation. The moment they are required, they are lazily instantiated. Thereby, the approach used in [mc]square is no longer completely explicit but partly explicit and partly symbolic. DND has a significant influence on the size of state spaces. Without this abstraction technique, [mc]square could not model check most programs it can check using DND. As seen in Sect. 6 DND can be used together with other abstraction techniques implemented in [mc]square.

In the future, we want to investigate if we can establish a bisimulation relation for DND for values. The copying of values destroys the bisimulation relation. If we introduce instances of nondeterminism and copy these instances, instantiation such an instance would have an effect on all the instances and preserve the bisimulation. However, we have to observe the effects on the size of the state space and the number of different nondeterminism instances. Another thing that we want to implement is a model checking algorithm for a three-valued logic. This would make it possible to make propositions about registers used within the DND abstraction technique.

Summarizing, we think that this is a promising approach to analyze software for embedded systems. [mc]square can already handle programs of interesting size. Delayed nondeterminism is an abstraction technique that helps to tackle the state explosion problem. It can be combined with other techniques implemented in [mc]-square (e.g., path reduction and dead variable reduction). This technique can also be used for model checking software for many other microcontrollers. As we have experienced with delayed nondeterminism or path reduction (cf. [19]), there are abstraction techniques which perform better when model checking assembly code. Hence, we will focus future research on domain specific abstraction techniques.

References

[1] Balakrishnan, G., T. Reps, D. Melski and T. Teitelbaum, *WYSINWYX: What you see is not what you execute*, in: *Verified Software: Theories, Tools, Experiments* (2007), to appear.

[2] Bryant, R. E., *A methodology for hardware verification based on logic simulation*, Journal of the ACM **38** (1991), pp. 299–328.

[3] Clark, A., *A lazy non–deterministic functional language* (2000), http://www.dcs.kcl.ac.uk/staff/tony/docs/LazyNonDetLanguage.ps.

[4] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," The MIT Press, Cambridge, Massachusetts, 1999.

[5] Emerson, E. A., *Temporal and modal logics*, in: *Handbook of Theoretical Computer Science, vol. B*, Elsevier, 1990 .

[6] Godefroid, P., N. Klarlund and K. Sen, *DART: directed automated random testing*, SIGPLAN Not. **40** (2005), pp. 213–223.

[7] Heljanko, K., *Model checking the branching time temporal logic CTL*, Research Report A45, Helsinki University of Technology (1997).

[8] Holzmann, G. J., "The Spin Model Checker: Primer and Reference Manual," Addison-Wesley, 2003.

[9] Larsen, K. G., P. Pettersson and W. Yi, Uppaal *in a Nutshell*, Int. Journal on Software Tools for Technology Transfer **1** (1997), pp. 134–152.

[10] Launchbury, J., *An natural semantics for lazy evaluation*, in: *Proc. 20th ACM Symp. on Principles of Programming Languages (POPL '93)* (1993), pp. 144–154.

[11] Leven, P., T. Mehler and S. Edelkamp, *Directed error detection in C++ with the assembly-level model checker StEAM*, in: *Model Checking Software (SPIN)*, Lecture Notes in Computer Science **2989** (2004), pp. 39–56.

[12] Mehler, T., "Challenges and Applications of Assembly-Level Software Model Checking," Ph.D. thesis, Universität Dortmund (2005).

[13] Mercer, E. G. and M. D. Jones, *Model checking machine code with the GNU debugger*, in: *SPIN Workshop on Model Checking of Software*, LNCS **3639** (2005), pp. 251–265.

[14] Meseguer, J. and P. Thati, *Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols*, ENTCS **117** (2005), pp. 153–182.

[15] Noll, T. and B. Schlich, *Delayed nondeterminism in model checking embedded systems assembly code*, in: *Proc. of 3rd Annual Haifa Verification Conf. (HVC 2007)*, LNCS (2007), to appear.

[16] Schlich, B. and S. Kowalewski, *Model checking C source code for embedded systems*, in: T. Margaria, B. Steffen and M. G. Hinchey, editors, *Proc. IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation (IEEE/NASA ISoLA 2005)*, NASA/CP-2005-212788 (2005), pp. 65–77.
URL http://www-i11.informatik.rwth-aachen.de/schlich.html

[17] Schlich, B. and S. Kowalewski, *[mc]square: A model checker for microcontroller code*, in: T. Margaria, A. Philippou and B. Steffen, editors, *Proc. 2nd Int'l Symp. Leveraging Applications of Formal Methods, Verification and Validation (IEEE-ISoLA 2006)*, 2006, to appear in: IEEE proceedings.

[18] Schlich, B. and S. Kowalewski, *An extendable architecture for model checking hardware-specific automotive microcontroller code*, in: E. Schnieder and G. Tarnai, editors, *Proc. 6th Symp. Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)* (2007), pp. 202–212.

[19] Schlich, B., J. Löll and S. Kowalewski, *Application of static analyses for state space reduction to microcontroller assembly code*, in: *Proc. 12th Int'l Workshop Formal Methods for Industrial Critical Systems (FMICS 2007)*, LNCS (2007), to appear.

[20] Schlich, B., M. Rohrbach, M. Weber and S. Kowalewski, *Model checking software for microcontrollers*, Technical Report AIB-2006-11, RWTH Aachen University (2006).
URL http://aib.informatik.rwth-aachen.de/2006/2006-11.pdf

[21] Schlich, B., F. Salewski and S. Kowalewski, *Applying model checking to an automotive microcontroller application*, in: *Proc. IEEE 2nd Int'l Symp. Industrial Embedded Systems (SIES 2007)* (2007), to appear.

[22] Sen, K., D. Marinov and G. Agha, *CUTE: a concolic unit testing engine for C*, in: *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (2005), pp. 263–272.

[23] Vergauwen, B. and J. Lewi, *A linear local model checking algorithm for CTL*, in: *Proc. 4th Int'l Conf. on Concurrency Theory (CONCUR '93)*, Lecture Notes in Computer Science **715** (1993), pp. 447–461.

[24] Visser, W., K. Havelund, G. Brat, S. Park and F. Lerda, *Model checking programs*, Automated Software Engineering Journal **10** (2003).