# Reduction of Interrupt Handler Executions for Model Checking Embedded Software

Bastian Schlich[1], Thomas Noll[2], Jörg Brauer[1], and Lucas Brutschy[1]

[1] Embedded Software Laboratory, RWTH Aachen University
Ahornstr. 55, 52074 Aachen, Germany
[2] Software Modeling and Verification Group, RWTH Aachen University
Ahornstr. 55, 52074 Aachen, Germany

**Abstract.** Interrupts play an important role in embedded software. Unfortunately, they aggravate the state-explosion problem that model checking is suffering from. Therefore, we propose a new abstraction technique based on partial order reduction that minimizes the number of locations where interrupt handlers need to be executed during model checking. This significantly reduces state spaces while the validity of the verification results is preserved. The paper details the underlying static analysis which is employed to annotate the programs before verification. Moreover, it introduces a formal model which is used to prove that the presented abstraction technique preserves the validity of the branching-time logic CTL*-X by establishing a stutter bisimulation equivalence between the abstract and the concrete transition system. Finally, the effectiveness of this abstraction is demonstrated in a case study.

## 1 Introduction

Embedded systems frequently occur as part of safety-critical systems. Full testing of these systems is often not possible due to fast time to market, uncertain environments, and the complexity of the systems. Model checking has been recognized as a promising tool for the analysis of such systems. A major problem for the application of model checking is the state explosion. When model checking embedded-systems software, interrupts are a major challenge. They are important as many features of embedded systems are implemented using interrupts, but they have a considerable impact on the size of the state space. Whenever they are enabled, they can interact with the main program and influence the behavior of the overall system.

To make model checking applicable to embedded systems software, we developed a model checker for microcontroller assembly code called [MC]SQUARE [1]. This model checker works directly on the assembly code of the program and automatically applies abstraction techniques such as delayed nondeterminism [2] and delayed nondeterminism for interrupts [3] to tackle the state-explosion problem. This paper describes a new abstraction technique called *interrupt handler execution reduction* (IHER), which is based on the idea of partial order reduction

(POR). It reduces the number of program locations at which the possible execution of interrupt handlers (IHs) has to be considered. This can greatly reduce state spaces built during model checking.

The idea behind IHER is similar to the one behind POR (cf. Sect. 6), but the algorithms used are different due to the fact that the pseudo parallelism introduced by IHs significantly differs from concurrent threads in its asymmetry. Threads can block other threads and control can nondeterministically change between threads. IHs, however, can only interrupt the main program, but they cannot be interrupted by the main program. The instructions of the main program have to be executed whereas the execution of IHs is usually nondeterministic. Moreover in [MC]SQUARE, IHs are required to be executed atomically because if IHs can mutually be interrupted, a stack collision will eventually occur as the stack that stores the return addresses is bounded on real microcontrollers. Hence, we model IHs as atomic actions in IHER. Consequently, as IHs do not necessarily terminate due to loops or usage of microcontroller features, we cannot guarantee termination of atomic actions. In contrast, in POR it is assumed that atomic actions always terminate.

The contribution of this paper is twofold. We have developed a static analysis framework for microcontroller assembly code that forms the basis for IHER. Furthermore, we have developed a dynamic part that applies IHER during model checking. The static analysis identifies program locations at which the execution of IHs can be prevented because they do not influence the (visible) behavior of the system or software, respectively. During model checking, the execution of IHs is blocked at these locations. As we will see, this abstraction technique guarantees a stuttering bisimulation equivalence between the concrete and the abstract transition system. Therefore, it preserves the validity of CTL*-X [4] formulas.

The paper is structured as follows. First, [MC]SQUARE is introduced in Sect. 2. Then, Sect. 3 explains the general idea of our abstraction technique and details the applied algorithms. Section 4 presents a formal model and gives a sketch of the proof that the abstraction technique presented in this paper actually preserves a stuttering bisimulation equivalence. The effectiveness of the technique is demonstrated in the case study described in Sect. 5. Related work, particularly with respect to POR, is presented in Sect. 6.

## 2 [MC]SQUARE

[MC]SQUARE [1] is a model checker for microcontroller assembly code. It can verify code for five different microcontrollers, namely ATMEL ATmega16 and ATmega128, Infineon XC167, Intel MCS-51, and Renesas R8C/23. It accepts programs given in different binary-code file formats such as ELF or Intel Hex Format and, additionally, it reads the corresponding C code if it is available. [MC]SQUARE processes specifications given in CTL [4], which can include propositions about general purpose registers, I/O registers, general memory locations, and the program counter. (Depending on the applied abstraction techniques,

propositions about the program counter may be disallowed.) If debug information is available, specifications can also include propositions about C variables.

[MC]SQUARE uses explicit model checking algorithms, but the states are partly symbolic. That is, they do not represent single concrete states but sets of concrete states, and are introduced by abstractions of the microcontroller memory. In [MC]SQUARE, we have modeled different abstractions of the memory that vary with respect to the degree of abstraction. Beside these memory-oriented methods, we have also implemented several general purpose abstraction techniques such as dead variable reduction and path reduction [5]. It is important to notice that [MC]SQUARE always creates an over-approximation of the behavior shown by the real microcontroller. Depending on the applied abstraction techniques, [MC]SQUARE preserves the validity of CTL, the universal fragment of CTL (ACTL) [4], or ACTL-X, which refers to ACTL without the *next* operator.

Figure 1 shows the model checking process that is applied by [MC]SQUARE. First, the binary code, the C code (if available), and the formula are parsed and transformed into their internal representations. Then, the static analyzer is executed and the program is annotated using information from the assembly code, the debug information, and the CTL formula. These annotations are later used by the simulator to reduce the state space.

The static analyzer performs several analyses as described by Schlich [1]. A major challenge for the analysis of assembly code are indirect references to the memory. As most of these are caused by stack-handling operations, a stack analysis is employed to determine the values of the stack pointer in order to restrict the memory regions that can be accessed [6]. Other indirect references are rarely used. To generate an over-approximation, we assume in these cases that indirect references can access the complete memory of the microcontroller.

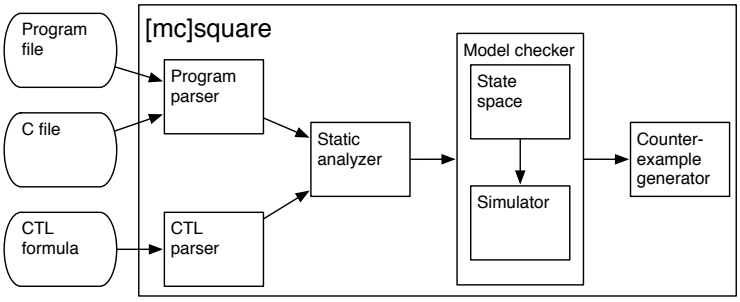



**Fig. 1.** Model checking process applied in [MC]SQUARE

In the next step, [MC]SQUARE performs model checking. Currently, we have implemented two different algorithms: one for checking invariants, and one on-the-fly CTL model checking algorithm described by Heljanko [7]. The model checker requests states from the state space. If successors of a state are not yet created, the state space uses the simulator to generate them on-the-fly.

The simulator natively handles nondeterminism and creates an over-approximation of the behavior shown by the real microcontroller. Within the simulator component, we have modeled the different microcontrollers. Creation of successors is done by means of interpretation. A state is loaded into the model of the microcontroller, and then all its possible successors are generated. A state can have more than one successor because interrupts can occur while executing the program and because input can be read from the environment or from devices with nondeterministic behavior such as timers.

If the model checker refutes the property under consideration, the counterexample generator creates a counterexample, which is also optimized. That is, loops and other unneeded parts are removed to ease its comprehension. The counterexample is presented in the assembly code, in the C code (if available), in the control flow graph of the assembly code, and as a state space graph.

## 3 Reduction of Interrupt Handler Executions

The execution of IHs has a significant impact on state space sizes when model checking microcontroller programs. Interrupts introduce pseudo parallelism in microcontroller programs as they can possibly occur at every program location where they are enabled (cf. Sect. 1). Thus, state spaces can grow exponentially with the number of interrupts used. Similarly to the observation that led to the partial order reduction technique (cf. Sect. 6), we observed that the execution of IHs does not always influence the behavior of a program. In the following, an abstraction technique is described that reduces the number of locations where IHs have to be considered. First, the general idea is presented, then, details of the applied analysis are given, and in the end, the application of this technique is demonstrated using an example.

### 3.1 General Idea

We have developed an abstraction technique called *interrupt handler execution reduction*, which reduces the number of IH executions by blocking IHs at program locations where there is no dependency between certain IHs and the program. There is a dependency if either one influences the other or the visible behavior of the program is changed. An IH influences a program location if it, for example, writes a memory location that is accessed by the program location. Here, an access refers to both a reading or writing reference to a memory location. On the other hand, a program location influences an IH if it, for instance, enables or disables interrupts. The visible behavior of the program is changed by a visible action if a memory location is written that is used in an atomic proposition (AP). The same applies for dependencies between IHs.

When using this abstraction technique, propositions about the program counter are not allowed because the program counter is changed at all program locations, and therefore, IHs could never be suppressed. In the analysis, the execution of IHs is assumed to be atomic, and therefore, IHs are treated as single

instructions. Our idea, however, can easily be extended to the case that IHs are interruptible by treating each IH the same way as the main process is treated. As IHs can possibly contain divergent loops, termination of IHs cannot be guaranteed. To preserve the validity of specifications with respect to our abstraction technique, divergent behavior has to be observable both in the concrete and the abstract model (see Sect. 4). Hence, IHs have to be executed at least once between two visible actions.

We additionally require that an interrupt can occur arbitrarily often at a single program location because at this location it has to mimic all possible behaviors to create an over-approximation of the real behavior. On the real hardware, for some microcontrollers such as the ATMEL ATmega16, the execution of an IH is always followed by the execution of an instruction of the program. Allowing an arbitrary number of occurrences adds additional behavior and thus leads to an over-approximation, but it again reduces state spaces.

The IHER technique comprises two parts: a static analysis that annotates the program, and a dynamic part that uses the annotations during model checking to suppress the execution of IHs where they do not need to be considered. The next section details the static analysis and the last section provides an example.

### 3.2 Static Analysis

As a prerequisite for determining program locations where IHs can be blocked, [MC]SQUARE employs a sequence of different context-sensitive static analyses and combines their results as detailed by Schlich [1]. First, the control flow graph (CFG) of the program is built and all program locations are annotated with the sets of live variables, reaching definitions, and the status of interrupt registers, that is, the information whether certain interrupts are enabled or disabled. During these analyses, information about the stack is used to limit the over-approximation. As their results potentially influence each other, these analyses are conducted within a loop until a fixed point is reached. Using the information that was obtained in this way, the analysis for the IHER abstraction technique is applied. It consists of the following four steps:

1. Detect dependencies between IHs
2. Detect dependencies between program locations and IHs
3. Refine results
4. Label blocking locations

In the following, these four steps are detailed.

**Detect Dependencies between IHs.** In the first step of the analysis, dependencies between IHs are identified. This is formalized by the relation $\bowtie \subseteq \mathsf{IH} \times \mathsf{IH}$ where $i,\ j \in \mathsf{IH}$ *depend on each other*, denoted $i \bowtie j$, if one of the following conditions holds:

– one enables or disables the other,

– one writes a memory location accessed by the other, or
– one writes a memory location used in an AP.

This relation is obviously symmetric. If one IH enables or disables another IH, all possible interleavings between both are relevant. Therefore, not only the enabled/disabled IH has to be executed if the enabling/disabling IH is executed but also vice versa as otherwise behavior could get lost. This also applies if one IH writes a memory location that is accessed by another IH. Note that in the last condition only one IH is mentioned. Thus, if there is one IH that writes a memory location that is used in an AP, all IHs depend on each other. An IH that writes an AP is related to all IHs including those that do not write APs because its execution could be prevented by a non-terminating IH. This includes the case of two IHs that both write APs: they are related because they both alter the visible behavior of the program, and thus, all their possible interleavings are relevant.

The transitive-reflexive closure of $\bowtie$ is denoted by $\bowtie^*$ and induces a partitioning of $\mathsf{IH}$. This partitioning is used in the following way. Whenever one of the IHs has to be executed, all other IHs in its equivalence class have to be executed as well. The algorithm to compute the dependency relation performs a nested iteration over all IHs based on the conditions described above.

**Detect Dependencies between Program and IHs.** In the second step of the analysis, [MC]SQUARE determines dependencies between the program and the IHs and identifies program locations where interrupts have to be executed. There exists a dependency between a program location and an IH if either one influences the other or the program behavior is visibly changed. The latter is the case if an instruction or an IH writes memory locations used in APs.

To detect the dependencies between the program and the IHs, [MC]SQUARE marks specific program locations with the following two labels: *execution* and *barrier*. The label *execution* implies that there exists a dependency between the preceding program location and an IH, and thus, this IH needs to be executed eventually. The label *barrier* denotes that there exists a dependency between that program location and an IH, and therefore, this IH needs to be executed before the instruction at that location is executed. Otherwise, visible behavior could get lost. In the later refinement step, label *execution* can be moved until a label *barrier* is reached.

Let program location $k$ be a direct predecessor of program location $l$. Formally, for each $i \in \mathsf{IH}$, $l$ is labeled with $\mathit{execution}_i$ if one of the following conditions is satisfied:

– $k$ enables or disables $i$,
– $k$ writes a memory location that is accessed by $i$, or
– $k$ writes a memory location that is used in an AP.

These conditions are similar to the conditions for dependencies between IHs. If $k$ enables or disables an IH, this IH has to be executed eventually to exhibit

the changed behavior. The same applies if $k$ writes a memory location that is accessed by an IH. As interrupts are deactivated in the initial program location, a program location has to enable interrupts before they can influence the program or change the visible behavior of the program. Note that in the last condition only $k$ is mentioned and not a specific IH. If $k$ writes an AP, each IH has to be executed afterwards because the execution of an IH could either prevent the execution of another instruction that writes an AP or the IH could itself write an AP. If $l$ is labeled with $\mathit{execution}_i$, it is also labeled with $\mathit{execution}_j\ \forall j \in [i]_{\bowtie^*}$ because all IHs of the same equivalence class have to be executed at the same location.

For each $i \in \mathsf{IH}$, a program location $l$ is labeled with $\mathit{barrier}_i$ if one of the following conditions holds:

- $i$ writes a memory location that is accessed by $l$,
- $l$ enables or disables $i$,
- $l$ writes a memory location that is accessed by $i$, or
- $l$ writes a memory location that is used in an atomic proposition.

The first condition is different from the conditions for label $\mathit{execution}_i$. If $i$ writes a memory location that is accessed by $l$, $i$ has to be executed before $l$ is executed because otherwise a possibly changed behavior could get lost: the execution of $i$ after the execution of $l$ could no longer influence the execution of $l$. This condition shows the asymmetry between the program and the IHs. The remaining conditions are duals of the conditions for label $\mathit{execution}_i$. They are used to guarantee that a possibly changed behavior is finally considered. If $l$ is labeled with $\mathit{barrier}_i$, it is also labeled with $\mathit{barrier}_j\ \forall j \in [i]_{\bowtie^*}$.

**Refine Results.** In the refinement step, [MC]SQUARE tries to reduce state spaces further by moving $\mathit{execution}_i$ labels until their execution is actually required. This is possible because in the previous step, [MC]SQUARE only locally labeled program locations where IH behavior was changed, but did not check whether their changed behavior actually influences the program. During refinement, the context is taken into account. An IH and all dependent IHs do not have to be executed if all behavior relevant to the specification and the program is created through their execution at another program location. Therefore, it is sufficient to execute IHs at only one of these locations. In the refinement step, [MC]SQUARE moves labels $\mathit{execution}_i$ forward until one of the following conditions holds:

- a program location labeled with $\mathit{barrier}_i$ is reached,
- a loop entry is found, or
- a loop exit is found.

This further reduces state spaces by postponing the execution of IHs until required. The label $\mathit{execution}_i$ cannot be moved over a program location labeled with $\mathit{barrier}_i$ because it either influences the next instruction or the next instruction influences its behavior, and it has not yet been executed. Furthermore,

a label $execution_i$ is not moved into a loop because this would possibly increase the size of the state space. Moreover, it is not moved out of a loop because loop termination cannot be guaranteed and divergent behavior has to be preserved in the abstract system.

**Label Blocking Locations.** In the last step, all program locations are labeled with IHs that can be blocked at the corresponding program location. An IH can be blocked at a program location if its execution is not required. Thus, a program location is labeled with $blocking_i$ if it is not labeled with $execution_i$.

### 3.3 An Example

To illustrate the IHER abstraction technique, we give an example. We apply this analysis to the program shown in Fig. 2(a) and the IH presented in Fig. 2(b). In the main program, interrupts are first enabled and then some calculations are executed on registers `r1`, `r2`, and `r3`. The IH accesses only register `r1` and doubles its value. No atomic propositions are used in this example.

| | | |
|---|---|---|
| $l_0$ | `SEI` | enable interrupts |
| $l_1$ | `LD r2,5` | $r_2 \leftarrow 5$ |
| $l_2$ | `ADD r1,r2` | $r_1 \leftarrow r_1 + r_2$ |
| $l_3$ | `MOV r2,r1` | $r_2 \leftarrow r_1$ |
| $l_4$ | `MOV r3,r2` | $r_3 \leftarrow r_2$ |
| $l_5$ | `CLI` | disable interrupts |
| $l_6$ | `RJMP -1` | self loop |

(a) Main program

| | | |
|---|---|---|
| $i_0$ | `ADD r1,r1` | $r_1 \leftarrow r_1 + r_1$ |
| $i_1$ | `RETI` | return |

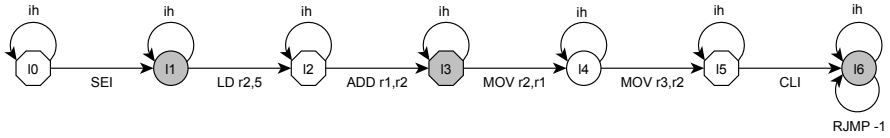
(b) Interrupt handler

**Fig. 2.** Assembly code of the main program and the interrupt handler (excerpt)

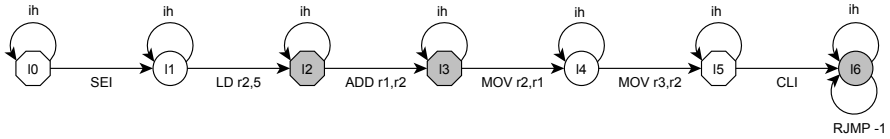
In this example, only one IH is used, and therefore, the first step of the analysis can be omitted. In the second step, we label the program locations with *execution* and *barrier*. Here, we omit the indices for clarity. The resulting labeled CFG is depicted in Fig. 3(a). White circles represent program locations without labels, white octagons represent locations labeled with *barrier*, and grey nodes represent program locations labeled with *execution*. Edges are labeled with the corresponding instruction or IH respectively.

Locations $l_1$, $l_3$, and $l_6$ are labeled with *execution* because their preceding instructions influence the IH. Locations $l_2$ and $l_3$ are labeled with *barrier* as the IH influences the current instruction. Hence, the IH has to be executed not later than at these locations. Locations $l_0$ and $l_5$ are labeled with *barrier* because interrupts are enabled or disabled by the respective instruction.

In the refinement step, *execution* labels are moved forward until either a *barrier* label or a loop is reached. The result of this step for the program is shown in Fig. 3(b). Here, only the *execution* label of $l_1$ is moved to $l_2$ because $l_2$ is a barrier. The *execution* label in $l_6$ cannot be moved due to the self loop.



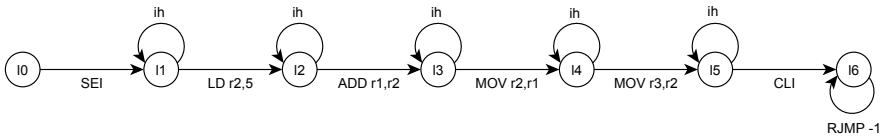

(a) Before refinement



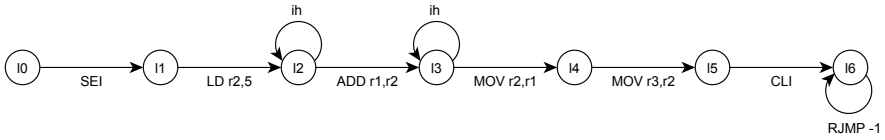

(b) After refinement

**Fig. 3.** CFGs of the code shown in Fig. 2

These labels are then translated into blocking locations. In Fig. 4 the differences in IH execution with and without IHER are shown. Figure 4(a) shows that without applying IHER, the IH is executed at five program locations because interrupts are disabled in $l_0$ and $l_6$. The application of IHER leads to the execution of the IH at only two locations as depicted in Fig. 4(b).




(a) Without IHER




(b) Using IHER

**Fig. 4.** Comparison of IH executions for program shown in Fig. 2

## 4 Formal Model & Correctness Proof

This section introduces the formal model on which the correctness proof of our abstraction technique is based. It is defined in two steps: (1) The syntactic structure of the microcontroller program is represented by its CFG, which consists of the program locations connected by control flow edges. Here, each edge carries an action label and a Boolean expression. The former represents the execution of either a single machine instruction or of a complete IH. The latter acts as a guard controlling the execution of, for instance, conditional branching instructions or IHs in dependence of the memory state. Note that single instructions of IHs are not considered as we assume their execution to be atomic. (2) Semantics is involved by associating with every action a mapping on the data space of the program, that is, the memory contents. This gives rise to a *labeled transition system* in which each state is given by a program location and a data state, where the latter represents the contents of general-purpose registers, I/O registers, and memory locations. Thus, the correctness proof boils down to showing that the original and the reduced CFG yield equivalent labeled transition systems.

### 4.1 The Formal Model

Formally, the CFG of the program is given by $G = (L, l_0, A, B, \longrightarrow)$ where

- $L$ is a finite set of *program locations*,
- $l_0 \in L$ is the *initial location*,
- $A$ is a finite set of *actions*,
- $B$ is a finite set of *guards*, and
- $\longrightarrow \subseteq L \times A \times B \times L$ is the *control flow relation* (where each entry is represented as $l \xrightarrow{a,b} l'$ with $l, l' \in L$, $a \in A$, and $b \in B$).

The introduction of guards allows to model, e.g., conditional branching instructions by two transitions with the same action and different guards, which indicate the outcome of the evaluation of the condition.

The semantics of a CFG is determined by associating with every action $a \in A$ a mapping $[\![a]\!] : D \to 2^D$, and with every guard $b \in B$ a mapping $[\![b]\!] : D \to \mathbb{B}$. Here, $D$ stands for the *data space*, that is, the finite set of *memory states* of the program. Interpreting $[\![a]\!](d)$ as a set of memory states allows us to model the non-deterministic nature of certain instructions, such as reading operations on input registers. Each of these sets is required to be non-empty and finite. A set is a singleton if the respective action is deterministic.

Applying this semantics to the given CFG $G$ yields a *labeled transition system* $T(G) = (S, s_0, A, \Longrightarrow, P, \lambda)$, which is defined as follows:

- $S := L \times D$ is the finite set of *states*,
- $s_0 := (l_0, d_0) \in S$ is the *initial state* where $d_0 \in D$ stands for the initial data state,
- $A$ is the finite set of *actions* (as before),

– $\Longrightarrow \subseteq S \times A \times S$ is the *transition relation*, given by: whenever $l \xrightarrow{a,b} l'$ in $G$ and $d \in D$ such that $[\![b]\!](d) = \mathsf{true}$, then $(l,d) \overset{a}{\Longrightarrow} (l',d')$ for every $d' \in [\![a]\!](d)$,
– $P$ is a finite set of *atomic propositions*, and
– $\lambda : S \to 2^P$ is the *property labeling*.

### 4.2 Correctness of the Abstraction

As explained in Sect. 3, IHER reduces the state space of the system by blocking IHs at program locations where they are independent of the main program. In other words, it removes certain transitions from the CFG (but keeps all locations), leading to a reduced graph $G_\sharp = (L, l_0, A, B, \longrightarrow_\sharp)$ with $\longrightarrow_\sharp \subseteq \longrightarrow$. According to the previous definition, $G_\sharp$ then yields a reduced labeled transition system $T(G_\sharp) = (S_\sharp, t_0, A, \Longrightarrow_\sharp, P, \lambda_\sharp)$ with $S_\sharp \subseteq S$, $t_0 = s_0$, $\Longrightarrow_\sharp \subseteq \Longrightarrow$, and $\lambda_\sharp = \lambda|_{S_\sharp}$.

We will now establish the correctness of our abstraction technique by showing that the original and the reduced transition system are equivalent. More concretely we will see that $T(G)$ and $T(G_\sharp)$ are related by a *divergence-sensitive stutter bisimulation*, implying that our abstraction mapping preserves the validity of formulas in CTL*-X [8].

We begin with the definition of a *stutter bisimulation* [9], which is a binary relation $\rho \subseteq S \times S_\sharp$ such that $s_0 \rho t_0$ and, for all $s \rho t$,

– $\lambda(s) = \lambda_\sharp(t)$,
– if $s \overset{a}{\Longrightarrow} s'$ with $(s',t) \notin \rho$, then there exists a path $t \overset{a_0}{\Longrightarrow}_\sharp u_1 \overset{a_1}{\Longrightarrow}_\sharp \ldots \overset{a_{n-1}}{\Longrightarrow}_\sharp u_n \overset{a_n}{\Longrightarrow}_\sharp t'$ with $n \geq 0$, $s \rho u_i$ for every $i \in \{0,\ldots,n-1\}$, and $s' \rho t'$, and
– if $t \overset{a}{\Longrightarrow}_\sharp t'$ with $(s,t') \notin \rho$, then there exists a path $s \overset{a_0}{\Longrightarrow} u_1 \overset{a_1}{\Longrightarrow} \ldots \overset{a_{n-1}}{\Longrightarrow} u_n \overset{a_n}{\Longrightarrow} s'$ with $n \geq 0$, $u_i \rho t$ for every $i \in \{0,\ldots,n-1\}$, and $s' \rho t'$.

Thus, a stutter bisimulation requires equivalent states to be equally labeled, and every outgoing transition in one system must be matched in the other system by a transition to an equivalent state, but allowing some transitions that are internal to the equivalence class of the source state. Note that action labels are not important here.

In our application, a stutter bisimulation $\rho \subseteq S \times S_\sharp$ between the original and the reduced system can inductively be defined as follows:

1. $s_0 \rho t_0$,
2. if $s \rho t$, $a \in A$, $s' \in S$, and $t' \in S_\sharp$ such that $s \overset{a}{\Longrightarrow} s'$, $t \overset{a}{\Longrightarrow}_\sharp t'$, and $\lambda(s') = \lambda_\sharp(t')$, then $s' \rho t'$, and
3. if $s \rho t$, $a \in A$, and $s' \in S$ such that $s \overset{a}{\Longrightarrow} s'$ and $t$ has no $\overset{a}{\Longrightarrow}_\sharp$-successor, then $s' \rho t$.

This definition handles the following three cases: (1) it relates the initial states, (2) it relates states that are reachable from stutter-bisimilar states in both systems via the same machine instruction or via the same (non-blocked) IH, and

(3) it relates a state that is reachable via some IH in the original system with the state in the reduced system where this IH is blocked.

The following arguments show that $\rho$ is indeed a stutter bisimulation; details are omitted for lack of space. First, whenever $s\rho t$ with $s = (l, d) \in S$ and $t = (l_\sharp, d_\sharp) \in S_\sharp$, then $l = l_\sharp$ and $\lambda(l, d) = \lambda_\sharp(l_\sharp, d_\sharp)$. This is obvious in cases 1 and 2 of the definition of $\rho$, and also valid in 3 as the blocked IH returns to the same program location (implying $l = l_\sharp$), and must be invisible with respect to the atomic propositions (implying $\lambda(l, d) = \lambda_\sharp(l_\sharp, d_\sharp)$).

Second, the remaining requirements of a stutter bisimulation follow from the observation that, whenever $s\rho t$ (where $s \in S$ and $t \in S_\sharp$),

- if $s \stackrel{a}{\Longrightarrow} s'$ with $(s', t) \notin \rho$, then case 3 cannot apply as $s'\rho t$ otherwise. Hence, there exists $t' \in S_\sharp$ such that $t \stackrel{a}{\Longrightarrow}_\sharp t'$. For at least one of these states, it must be true that $\lambda(s') = \lambda_\sharp(t')$ (since $\lambda(s) \neq \lambda_\sharp(t)$ otherwise, contradicting $s\rho t$), and hence $s'\rho t'$;
- if $t \stackrel{a}{\Longrightarrow}_\sharp t'$ with $(s, t') \notin \rho$, then again case 2 must apply with $s \stackrel{a}{\Longrightarrow} s'$ and $s'\rho t'$.

The last step in our correctness proof consists of showing that both the original and the reduced transition system exhibit the same behavior with respect to non-terminating computations. Formally, a state $s \in S$ in a labeled transition system $(S, s_0, A, \Longrightarrow, P, \lambda)$ is called *$\rho$-divergent* with respect to an equivalence relation $\rho \subseteq S \times S$ if there exists an infinite path $s \stackrel{a_1}{\Longrightarrow} s_1 \stackrel{a_2}{\Longrightarrow} s_2 \stackrel{a_3}{\Longrightarrow} \ldots$ such that $s\rho s_i$ for all $i \geq 1$. The relation $\rho$ is called *divergence-sensitive* if, for every $s_1\rho s_2$, $s_1$ is $\rho$-divergent iff $s_2$ is $\rho$-divergent.

Again, it can be shown that the stutter bisimulation $\rho \subseteq S \times S_\sharp$ as defined above is also divergence-sensitive, the essential arguments being that non-terminating computations only occur in the form of cycles (as the state space is finite), and that our abstraction technique never completely blocks the execution of an IH in a loop, and therefore preserves divergence. This completes the proof that our abstraction technique is correct with respect to formulas in CTL*-X.

## 5 Case Study

This section describes a case study conducted with [MC]SQUARE using the IHER technique. We analyzed five programs for the ATMEL ATmega16 to evaluate the performance of our abstraction method. All programs were written by students during lab courses or diploma theses and have previously been used to evaluate the impact of other techniques developed for [MC]SQUARE. A more thorough description of the analyzed programs is given by Schlich [1]. Note that for all programs, delayed nondeterminism (DND) [2] is used, which affects state space sizes by delaying the instantiation of nondeterministic values until their concrete value is required. This way, [MC]SQUARE can handle programs of up to 4 billion (symbolic) states. The larger programs used in this case study could not be checked without DND.

The differences in state space sizes with and without IHER are presented in Table 1. It shows the numbers with and without dead variable reduction (DVR) enabled, which reduces state spaces by removing unused variables. Here, the formula AG `true` was checked as it requires the creation of the complete state space.

Two different versions of a controller for a powered window lift used in a car were analyzed, one of which containing defects caused by missing protection of shared variables, and a second one where those errors were fixed. Both programs consist of 290 lines of assembly code and use two interrupts and one timer. Depending on the applied static analysis techniques, the state space sizes are reduced by between 64% and 82%. The second program controls a fictive chemical plant. It consists of 225 lines of assembly code. One timer and two interrupts are used. The IHER technique reduced the state space by approx. 98%. The last program implements a four channel speed measurement with a CAN bus interface. It consists of 384 lines of assembly code. The state spaces were reduced by approx. 89%.

**Table 1.** Number of states stored by [MC]SQUARE

| Without DVR | Default | Time [s] | IHER | Time [s] | Reduction |
|---|---|---|---|---|---|
| `window_lift.elf` (error) | 316,334 | 6.25 | 64,164 | 7.32 | 80% |
| `window_lift.elf` (fixed) | 129,030 | 2.81 | 23,852 | 6.04 | 82% |
| `plant.elf` (error) | 123,699,464 | 3,428 | 2,161,624 | 43 | 98% |
| `plant.elf` (fixed) | 75,059,765 | 1,956 | 1,327,715 | 25 | 98% |
| `can.elf` | 147,259,483 | 3,917 | 16,187,483 | 392 | 89% |
| With DVR | Default | Time [s] | IHER | Time [s] | Reduction |
| `window_lift.elf` (error) | 111,591 | 6.73 | 28,153 | 8.21 | 75% |
| `window_lift.elf` (fixed) | 23,013 | 5.52 | 14,919 | 6.89 | 64% |
| `plant.elf` (error) | 123,699,464 | 3,513 | 2,161,624 | 42 | 98% |
| `plant.elf` (fixed) | 75,059,765 | 1,940 | 1,327,715 | 25 | 98% |
| `can.elf` | 147,259,483 | 3,954 | 16,187,483 | 394 | 89% |

These results show that the IHER abstraction technique greatly reduces state spaces for a number of different programs. This is still true in the presence of other abstraction techniques such as DVR, meaning that both can be combined. The magnitude of improvement depends on various factors such as the number of interrupts used, dependencies between IHs and instructions, dependencies between IHs, the overall structure of the program, and the property to be verified.

## 6 Related Work

In the past, much work has been carried out to limit the state explosion in model checking resulting from concurrent activities in a system. A prominent technique is partial order reduction (POR) [10,11,12], which tries to reduce the

number of possible orderings of concurrent actions that need to be analyzed for model checking. This reduction is based on two important notions, namely, *independence* and *visibility*. Here, the first characterizes the commutativity of two actions, meaning that the execution of either of them does not disable the other and that executing both in any order always yields the same result. The second notion, visibility, refers to the property that the execution of an action does not affect the (in)validity of the formula to be checked. Together, both properties allow to reduce a transition system by only exploring a subset (the *ample set*) of all transitions enabled in a given state. A general overview of concepts related to POR is given by Valmari [13].

As pointed out in the introduction, our technique differs from POR in the following way. POR works in the context of concurrent threads while IHER works in the context of sequential programs and pseudo-parallelism introduced by IHs. Threads differ from IHs in that the interleaving between different threads is nondeterministic. For IHs only their occurrence is nondeterministic, that is, either they occur at a program location or they do not occur at a program location. Threads can be interrupted at any location since control can change nondeterministically between all threads. An IH, however, can interrupt the main program, but the main program cannot interrupt an IH. This means that an IH has to be executed completely until execution of the main program can continue. The same asymmetry applies in case that IHs can interrupt other IHs. Due to this asymmetry, we have to account for additional dependencies between the main program and IHs. In POR, all atomic actions are guaranteed to terminate. Since we represent IHs as atomic actions which can contain non-terminating loops, atomic actions are not guaranteed to terminate in our setting.

IHs could be modeled using threads. Techniques for converting interrupt-driven programs into equivalent programs using threads have been developed by Regehr and Cooprider [14]. This modeling can be done on source code level, but it involves some challenges. The peculiarities of interrupts vary between different microcontrollers. On some microcontrollers, IHs are non-interruptible while IHs can be interrupted on other microcontrollers. In some architectures interrupts have no priorities, in other architectures they have fixed or even dynamic priorities. This approach can, however, not be used for microcontroller assembly code as there is no thread model for microcontroller assembly code.

Kahlon et al. [15] developed an extension for partial order reductions using sound invariants. In their approach, the product graph of a concurrent system is iteratively refined, and statically unreachable nodes are removed. In contrast to our approach, only a context-insensitive static analysis is performed.

The notion of independent actions based on Lipton's theory of reduction [16] was introduced by Katz and Peled [17]. Our definition of dependencies between the main process and IHs can also be seen as an extension of Lipton's theory where, in addition to the dependencies that are induced by accesses to shared variables, also the control dependencies imposed by enabling and disabling interrupts are taken into account.

Recently, Elmas et al. [18] have described a proof calculus for static verification of concurrent programs using shared memory. In this approach, the concept of atomicity is used for computation of increased atomic code blocks, which are then, in contrast to our approach, verified sequentially.

A static analysis based on Petri nets to capture causal flows of facts in concurrent programs was proposed by Farzan and Madhusudan [19], but it only implements a restricted model of communication and synchronization compared to our setting. Another approach by Lal and Reps [20] adapts static analyses for sequential programs and extends them to work in a concurrent setting while our approach embodies specific analyses for concurrency. Other approaches, such as the work by Qadeer and Rehof [21] or Lal et al. [22], tackle the state explosion by imposing an upper bound on the number of context-switches, which is not possible in our setting.

## 7 Conclusion & Future Work

In this paper, we have presented a new abstraction technique called *interrupt handler execution reduction*, which is based on partial order reduction. It reduces state spaces by blocking the execution of interrupt handlers at certain program locations during model checking. It preserves the validity of CTL*-X and, as shown in the case study presented in Sect. 5, can significantly reduce state spaces. Two ingredients are needed for implementing this abstraction technique: a static analysis and a dynamic part executed during model checking. The static analysis determines program locations where interrupt handlers can be blocked. The model checking part then prevents the execution of the corresponding interrupt handlers at these program locations.

In the future, we want to improve the static analysis that is used for this abstraction technique. Currently, we rely on a coarse analysis of pointer variables. In many cases, our analysis has to over-approximate the set of possible address values. A more precise pointer analysis would improve the results of other static analyses such as live variable and reaching definitions analysis as well.

Another candidate for improvement is the refinement phase. From our point of view, there is no optimal static solution to this problem. We think that better heuristics can be found if termination of certain loops can be determined. Given this, we could postpone the execution of interrupt handlers beyond these loops.

## References


1. Schlich, B.: Model Checking of Software for Microcontrollers. Dissertation, RWTH Aachen University, Aachen, Germany (June 2008)
2. Noll, T., Schlich, B.: Delayed nondeterminism in model checking embedded systems assembly code. In: Hardware and Software: Verification and Testing (HVC 2007), Haifa, Israel. Volume 4899 of LNCS., Springer (2008) 185–201
3. Herberich, G., Noll, T., Schlich, B., Weise, C.: Proving correctness of an efficient abstraction for interrupt handling. In: Systems Software Verification (SSV 2008). Volume 217 of ENTCS., Elsevier (2008) 133–150

4. Emerson, E.A.: Temporal and Modal Logics. In: Handbook of Theoretical Computer Science. Volume B. The MIT Press (1991) 995–1072
5. Yorav, K., Grumberg, O.: Static analysis for state-space reductions preserving temporal logics. Formal Methods in System Design **25**(1) (2004) 67–96
6. Brauer, J., Schlich, B., Reinbacher, T., Kowalewski, S.: Stack bounds analysis for microcontroller assembly code. In: 4th Workshop on Embedded Systems Security (WESS 2009), Grenoble, France, ACM (2009) To appear.
7. Heljanko, K.: Model checking the branching time temporal logic CTL. Research Report A45, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland (May 1997)
8. Browne, M., Clarke, E., Grumberg, O.: Characterizing finite kripke structures in propositional temporal logic. Theor. Comput. Sci. **59**(1-2) (1988) 115–131
9. van Glabbeek, R., Weijland, W.: Branching time and abstraction in bisimulation semantics. Journal of the ACM **43**(3) (1996) 555–600
10. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Computer Aided Verification (CAV 1990), New Brunswick, USA. Volume 531 of LNCS., Springer (1990) 176–185
11. Holzmann, G.J., Peled, D.A.: An improvement in formal verification. In: Formal Description Techniques VII. IFIP International Federation for Information Processing, Springer (1995) 197–211
12. Peled, D.: Ten years of partial order reduction. In: 10th Int. Conf. on Computer Aided Verification (CAV '98). Volume 1427 of LNCS. (1998) 17–28
13. Valmari, A.: The state explosion problem. In: Petri Nets. Volume 1491 of LNCS., Springer (1996) 429–528
14. Regehr, J., Cooprider, N.: Interrupt verification via thread verification. Electronic Notes in Theoretical Computer Science **174**(9) (2007) 139–150
15. Kahlon, V., Sankaranarayanan, S., Gupta, A.: Semantic reduction of thread interleavings in concurrent programs. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS 2009), York, UK. Volume 5505 of LNCS., Springer (2009) 124–138
16. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Communications of the ACM **18**(12) (1975) 717–721
17. Katz, S., Peled, D.: Defining conditional independence using collapses. Theoretical Computer Science **101**(2) (1992) 337–359
18. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: Principles of Programming Languages (POPL 2009), Savanna, USA, ACM (2009) 2–15
19. Farzan, A., Madhusudan, P.: Causal dataflow analysis for concurrent programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), Braga, Portugal. Volume 4424 of LNCS., Springer (2007) 102–116
20. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: Computer Aided Verification (CAV 2008), Princeton, USA. Volume 5123 of LNCS., Springer (2008) 37–51
21. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS 2005), Edinburgh, UK. Volume 3440 of LNCS., Springer (2005) 93–107
22. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS 2008), Budapest, Hungary. Volume 4963 of LNCS., Springer (2008) 282–298