Effective Verification for Low-Level Software with Competing Interrupts

LIHAO LIANG, TOM MELHAM, and DANIEL KROENING, University of Oxford
PETER SCHRAMMEL, University of Sussex
MICHAEL TAUTSCHNIG, Queen Mary University of London

Interrupt-driven software is difficult to test and debug, especially when interrupts can be nested and subject to priorities. Interrupts can arrive at arbitrary times, leading to an exponential blow-up in the number of cases to consider. We present a new formal approach to verifying interrupt-driven software based on symbolic execution. The approach leverages recent advances in the encoding of the execution traces of interacting, concurrent threads. We assess the performance of our method on benchmarks drawn from embedded systems code and device drivers, and experimentally compare it to conventional approaches that use source-to-source transformations. Our results show that our method significantly outperforms these techniques. To the best of our knowledge, our work is the first to demonstrate effective verification of low-level embedded software with nested interrupts.

1. INTRODUCTION

Interrupts are a key design primitive for embedded software that interacts closely with hardware. The interrupt mechanism enables timely response to outside stimuli in a power-efficient way. Interrupts are common in all styles of computing platforms, including safety-critical embedded software, low-power mobile platforms, and high-end information systems.

But interrupt-driven code is difficult to engineer. Device drivers, typical examples of software that use interrupts heavily, are known as the "fault-hotspots" of the Linux Kernel [Chou et al. 2001; Palix et al. 2011]. The crux of the problem is the non-determinism inherent in systems that use interrupts; the hardware can divert control to the interrupt service routine (ISR) at any time, resulting in surprising interactions between the code that is interrupted and the ISR. The problem is exacerbated by *interrupt nesting*, where the ISR itself can be preempted by interrupts with higher priority.

Most existing approaches to validating interrupt-driven software rely on testing. But testing is particularly unreliable in the case of nested interrupts, as the number of possible interleavings—i.e. interspersions of interrupt arrivals into a run of the code—grows exponentially in the number of interrupts that occur. Bugs are therefore easily missed, and any errors that are observed are hard to reproduce.

This work is supported by EPSRC EP/H017585/1, a gift from Intel Corporation for research on *Effective Validation of Firmware*, by *VeTeSS: Verification and Testing to Support Functional Safety Standards*, Artemis Joint Undertaking 295311, ERC project 280053 "CPROVER", the H2020 FET OPEN 712689 SC^2 and SRC contracts no. 2012-TJ-2269 and 2016-CT-2707.

Authors' addresses: L. Liang, T. Melham and D. Kroening, Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; P. Schrammel, School of Engineering and Informatics, University of Sussex, Falmer, Brighton BN1 9QT, UK; M. Tautschnig, School of Electronic Engineering and Computer Science, Queen Mary University of London, Mile End Road, London E1 4NS, UK.

DOI: http://dx.doi.org/10.1145/0000000.0000000

The concurrent nature of interrupt generation and handling and the sheer size of the search space suggest a formal approach to validation. Model Checking, in particular, shines when applied to problems with subtle corner cases. Several model-checking based approaches to the problem have therefore been explored. A standard technique is to instrument the source code of the program by inserting conditional calls to the interrupt procedure, in effect mimicking the semantics of the hardware interrupt. The instrumented program is then passed to a conventional analyser for sequential programs. The approach is straightforward to implement, and is in principle viable in the case of nested interrupts as well. Sophisticated instances of this type of approach use over-approximating analyses, akin to partial-order reduction, to determine program locations where the call to the ISR can be omitted safely—for example when only variables that are not shared with the ISR are read or written [Schlich et al. 2009; Bucur and Kwiatkowska 2011].

The interleaving semantics of programs with nested interrupts closely resemble the semantics of programs with multiple threads. In particular, it is clear that the behaviours under interrupt semantics are a subset of the behaviours under thread semantics. This suggests an analysis using Model Checkers for concurrent programs, such as SPIN [Holzmann 1997]. False alarms caused by the over-approximation can be addressed by means of suitable instrumentation. We discuss this approach in Sec. 4.

The key contribution of this paper is a new alternative approach. It leverages recent advances in SAT-based analysis of concurrent systems. The central idea is to collect the set of possible events (reads or writes) in a system comprising an interrupt-driven program and its ISRs, and construct from these events a formula that relates their relative ordering by means of a symbolic *happens-before* relation. The formula is then passed to a modern SAT or SMT-solver together with a constraint that specifies the set of error states. If the solver determines the formula is satisfiable, an error trace giving the exact sequence of events can be extracted from the satisfying assignment.

The method has been shown to be effective for interleaving semantics of threads, including multi-core memory models with relaxed consistency [Alglave et al. 2013]. CBMC, a tool implementing this idea, was awarded the Gold medal in the "ConcurrencySafety" category in the 2017 software verification competition. The work on memory models reported by Alglave et al. [2013] explores a weakening of the thread semantics to capture the additional behaviours in the multi-core case. By contrast, our contribution in this paper is to employ a specially designed *strengthening* of the interleaving semantics of threads, in the form of a symbolic encoding by a partial order that more precisely captures the subset of interleavings required to model nested interrupts faithfully.

The novel encoding we present here provides the first demonstrated method for effective verification of software with nested interrupts. This work builds on—but also substantially extends—the framework of Alglave et al. [2013] to address this entirely different application area, demonstrating the essential generality of this framework. We provide a theoretical analysis of this new extension in Sec. 5.5, and experimental results that explore its effectiveness in section 6.

2. NESTED INTERRUPTS

Interrupt semantics is very platform-dependent. On some microcontrollers (e.g. ARM Cortex-M, AT89CXX), priorities or priority groups can be configured, whereas others use fixed priorities (e.g. AVR1305). There are architectures where lower-priority interrupts, when enabled, can preempt higher-priority ones (e.g. ATMEL256). The primary way to implement priorities is by enabling and disabling interrupts appropriately, e.g. by means of API functions such as AVR's `sei` and `cli` instructions.

In this paper, we target the following interrupt semantics. When entering an ISR, all interrupts are masked by the hardware and thus ignored. The ISR can then change

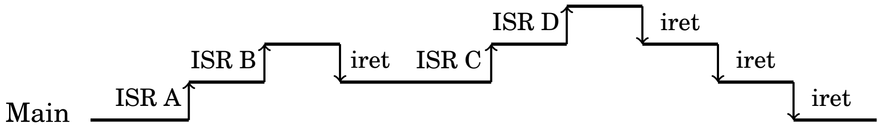


Fig. 1: A complex scenario of nested interrupts

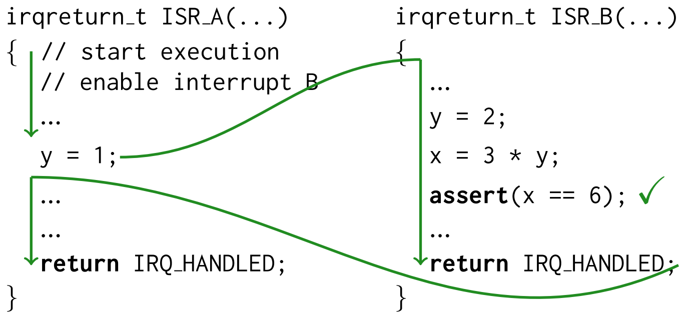
which specific interrupts are globally enabled or disabled by issuing a command, e.g. by setting particular bits in a global control register. The interrupt mask can then be lifted, allowing whatever interrupts are enabled at that point to subsequently preempt the execution of the ISR. As an example consider Fig. 1, where interrupt a is preempted by interrupts b and c, which is in turn preempted by interrupt d. The state recording which interrupts are enabled and which are disabled is global, and so any changes made by an ISR to this state remain in force when it completes execution.

Note that interrupts with priorities are a special case of this semantics, where the interrupts enabled during execution of an ISR are only those with higher priority. The handling of lower-priority interrupts is thus suspended until the handling of those with higher priority is completed. So, in principle, our semantics covers the vast majority of microcontrollers. For the modelling approaches we discuss later in this paper, however, we propose more direct ways in which priorities can be handled.

We illustrate the semantics of programs with nested interrupts through the simple code fragment in Fig. 2. In this example, the interrupt handler `ISR_A` enables the interrupt handled by `ISR_B`. This permits the execution of `ISR_A` to be preempted by an interrupt that is then serviced by `ISR_B`.

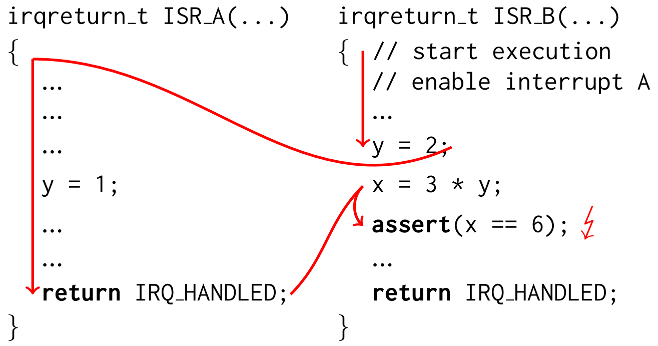
In this work, we focus on verification of safety properties, such as user-defined assertions. An example occurs in the code of `ISR_B` in Fig. 2. Suppose interrupt `ISR_A` enables interrupt 2, as shown, and interrupt `ISR_B` does not enable any interrupts. Then the property `(x == 6)` in `ISR_B` always holds because the execution of `ISR_B` cannot be interrupted. But if interrupts are enabled incorrectly—as depicted in Fig. 3—the assertion `(x == 6)` will be violated. The ISR `ISR_B` can be interrupted and the invoked ISR can change the value of `y` just before the assignment to `x` in `ISR_B` is executed.

We note that an interleaving semantics in which ISRs are treated as threads would admit an assertion-violating execution in the example of Fig. 2. This is because in a multi-threaded program, between statements `y = 2` and `x = 3 * y` of `ISR_B`, the control can return to `ISR_A` which then updates the value of `y`. Consequently, tools for multi-threaded software verification, naïvely applied, can report an assertion violation that is not possible in any actual execution of the program. This motivates the need for techniques that model the semantics of interrupts more precisely than threads do. The



```
irqreturn_t ISR_A(...)          irqreturn_t ISR_B(...)
{ // start execution            {
  // enable interrupt B           ...
  ...                             y = 2;
  y = 1;                          x = 3 * y;
  ...                             assert(x == 6); ✓
  ...                             ...
  return IRQ_HANDLED;             return IRQ_HANDLED;
}                               }
```

Fig. 2: Motivating example



```
irqreturn_t ISR_A(...)      irqreturn_t ISR_B(...)
{                           { // start execution
  ...                         // enable interrupt A
  ...                         ...
  ...                         y = 2;
  y = 1;                      x = 3 * y;
  ...                         assert(x == 6);
  ...                         ...
  return IRQ_HANDLED;         return IRQ_HANDLED;
}                           }
```

Fig. 3: Motivating example (modified)

program analysis principles discussed in this paper for interrupt handling also apply to the analysis of prioritised, preemptive task scheduling policies.

3. INTERRUPT SEQUENTIALISATION

The first conventional approach we compare our new method with is a source-to-source transformation that translates a program with interrupts into a sequential program, similar to the approach proposed by Kidd et al. [2010]. This allows us to analyse interrupt-driven code by leveraging existing verification tools for sequential programs.

Interrupts can happen at any program point where they are enabled. An intuitively simple way to model this as a sequential program is to insert a *non-deterministic call* to the handler of an interrupt wherever it is enabled. Semantically, a non-deterministic call is a function call that—independently—either is or is not invoked each time program control passes through the point at which it occurs. This models the contingent arrival, at that point of program execution, of interrupts that are generated externally.

We extend this idea to encompass the arrival and handling of nested interrupts. In this case, the transformation is realised by the insertion of (unconditional) calls to a certain *scheduling function*, instead of to a fixed interrupt handler. The scheduling function non-deterministically invokes handlers for the interrupts that are enabled. An optimisation resembling partial-order reduction is then used to reduce the number of calls to the scheduling function. This retains an invocation of the scheduler only in certain program locations, namely just before any program statement that has potential run-time interactions with interrupts through shared variables.

The basic idea is well-known [Bucur and Kwiatkowska 2011]. To the best of our knowledge, however, our transformation is the first to combine generalisation through a scheduling function that models potentially nested arrivals of diverse interrupts with an optimisation similar to partial-order reduction to verify low-level software with *nested* interrupts. We also provide experimental results (Sec. 6).

3.1. Modelling Nested Interrupts using a Scheduler

We define a scheduling function together with two global variables: `count` is a bound on the number of interrupt arrivals that can occur during program execution, and the Boolean `irq_enabled[i]` indicates whether interrupt i is currently enabled. At the beginning of the interrupt-driven code, `count` is set to a positive integer and the array `irq_enabled` is initialised to zeros. Then `irq_enabled[i]` is set to 1 right after any statement in the original code that explicitly enables interrupt i.

The scheduling function in Fig. 4 works as follows. First, if `count` is already zero the scheduler does nothing and exits. Otherwise, up to `count` interrupts might still occur.

```
void schedule_irq(void) {
  int j = count;
  int irq;

  for (int i = 0; i < j; i++) {
    // non-deterministically generate candidate irq
    irq = nondet_int() ;
    assume(irq >= 0 && irq < num_irqs);

    // if irq is enabled, contingently generate the interrupt
    if (count != 0 && irq_enabled[irq] && nondet_bool()) {
      count--;

      // invoke the ISR, preempting the calling program
      switch (irq) {
        case 0: ISR_A(); break;
        case 1: ISR_B(); break;
        ...
        default: ;
      }
    }
  }
}
```

Fig. 4: Scheduling function

To decide which ones do occur, at this point of the program execution, the scheduler loops count times and, in each iteration, makes a non-deterministic choice among all possible interrupts. If the chosen interrupt is enabled, according to irq_enabled, and count has not reached zero, the scheduler makes a non-deterministic choice between doing nothing (to simulate that this interrupt does not occur) and calling the corresponding interrupt handler, decrementing count. The variable count is checked in the conditional inside the loop because the ISRs may themselves be interrupted and so have invocations of the scheduler within them that also decrement count.

In the special case where interrupt priorities are explicit, a global array irq_pty is used to store the priority of each interrupt. When the scheduler generates a candidate interrupt, it is considered further only if its priority is strictly higher than the currently-executing interrupt (if any) stored in a global variable curr_pty. Then if the scheduling function decides to invoke an interrupt handler, it stores the current interrupt priority in a variable local to the scheduler, and updates curr_pty to the priority of the interrupt being handled, in the same way as Kidd et al. [2010]. Then it can restore curr_pty to the previous priority upon return from the invoked handler.

Although in our experiments we bound the number of interrupt arrivals, the sequential translation can be extended to handle an unlimited number of interrupt arrivals. To do so, we eliminate the count variable and replace the main loop with an infinite one. Each time through the loop, the scheduler makes a non-deterministic choice between exiting and generating another candidate interrupt. The resulting code might then be analysed by an unbounded model checker.

3.2. Reducing Scheduler Calls

In the naïve approach, interrupt arrivals are modelled by inserting a call to the scheduler at every boundary between instructions in the program to be translated, including the code of each of its interrupt handlers. But this will almost certainly make formal analysis computationally intractable. We therefore use an optimisation that strongly resembles *partial-order reduction* [Godefroid 1994; Godefroid and Wolper 1991; Katz and Peled 1988; Peled 1994; Clarke et al. 2001] to lessen the number of calls to our scheduling function, while preserving the semantics given by the naïvely-transformed program. As with partial-order reduction, the idea is to reduce the number of permutations of a sequence of individual state transitions that need to be considered, when the order of the transitions does not affect the result of formal analysis.

Our optimisation does a light-weight static analysis of the program to determine calls to the scheduler that can be omitted. Consider a point in the original program between two program statements, P and Q. Informally, this call may be omitted if nothing an invocation of the scheduler might do could interact via shared state with anything the statement Q does. More precisely, we can omit this call if:

— P and Q do not enable or disable any interrupts, and
— neither any interrupt enabled between P and Q, nor any nested interrupts that may be invoked within this interrupt's execution, have variables shared with Q that are written by Q and read by the interrupts, or vice versa.

In essence, the state transitions effected by the scheduler and by Q are independent, and so any invocation of the scheduler can be deferred until after the execution of Q.

4. MODELLING INTERRUPTS WITH THREADS

The second source-to-source transformation we use for comparison generates multi-threaded code and was suggested by Regehr and Cooprider [2007]. This method verifies program properties for specific, user-defined interrupt arrival scenarios, rather than modelling all possible patterns of interrupt arrival. The method requires us to know which interrupts are enabled at every point of program and ISR execution. We then choose a collection of locations, at each of which we insert some instrumentation that models the potential for a specific enabled interrupt to arrive at that or any subsequent point of program execution. These locations can be in the ISRs as well as in the main program. Interrupt re-entrance is not allowed, but a particular interrupt can arrive and be handled multiple times in succession.

This technique uses threads to model interrupts. The basic setup is as follows:

(1) We start the main program as a thread.
(2) At any point in the original code where an interrupt is enabled, we may insert code that explicitly spawns a further thread that executes that interrupt's ISR. We endow each such thread with a unique and distinct copy of its ISR function body, so that we can distinguish multiple arrivals of the same interrupt.
(3) We instrument the code so that a thread modelling an interrupt arriving and being handled starts executing only if its corresponding interrupt is enabled.
(4) Our instrumentation also ensures that execution of an enabled ISR preempts execution of the main program or the ISR that enabled it, but not vice versa.

Using threads this way means that an ISR can start running any time after the thread that models it is spawned, by the interleaving semantics of multi-threaded programs. Note that when we model multiple invocations of the same ISR, we create separate function bodies for each one. This allows them to have different instrumentations.

To ensure that only the threads of enabled ISRs can preempt those of enabling ones or the main program, we assign a *depth* to each thread according to the following rule. If thread j is spawned by thread i, then the depth of thread j is greater than that of thread i. If two threads are spawned by the same ISR thread (or the main program), then their depth is the same. We then instrument the code so that execution of all threads with smaller depth is blocked by any execution of a thread with greater depth.

Just as in the sequential transformation of Sec. 3, since we instrument at the source code level, we treat each program statement as the smallest atomic block in any interleaving of the execution of program threads. Function calls in the ISRs are inlined, because two ISRs may call the same function and may therefore need different instrumentations. We further assume the existence of a global Boolean array `irq_enabled[i]` that records whether each interrupt i is currently enabled or not. We suppose the state of this array is maintained either by the programming language or explicitly by the programmer. At the beginning of ISR i's function body, we insert the atomic statement below to make sure ISR i starts executing only if interrupt i is enabled.[1]

```
atomic { assume(irq_enabled[i]); }
```

This ensures that a thread modelling an interrupt, spawned at a point in the program where the interrupt is enabled, will not begin execution at some later time if the interrupt has in the meantime been explicitly disabled.

Another global array `thread_running` records the currently executing thread, along with all the threads the executions of which have begun but have been suspended. The code for each thread i is instrumented with assignments that set `thread_running[`i`]` to `1` initially and to `0` when the thread completes. We then insert at every boundary between statements throughout the code for thread i the following atomic statement

```
atomic { assume(thread_running[j_0]==0 && ... && thread_running[j_n]==0); }
```

where threads j_0 to j_n are all those threads, excluding thread i itself, the depths of which are greater than or equal to that of thread i. This assume statement ensures that the program statement it precedes is executed only if no thread of greater or equal depth is running—i.e. execution of this ISR has not been interrupted or no other ISRs that cannot be preempted by this ISR are currently running.

In the special case where interrupts have explicit priorities, we take an approach similar to that explained in Sec. 3. A global array `irq_pty` is used to store the priority of each interrupt. All threads that model an arrival of the same interrupt will have the same priority. We then use a slightly modified form of instrumentation, in which the atomic assume statement is as shown above but threads j_0 to j_n are all the threads, excluding thread i, the *priorities* of which are greater than or equal to that of thread i. That is, we simply replace 'depth' by 'priority'.

With all this instrumentation, the resulting translated program can be checked by verification tools for concurrent programs, such as CBMC [Alglave et al. 2013], ESBMC [Morse et al. 2014], or CSeq [Inverso et al. 2015].

5. ENCODING INTO PARTIAL-ORDER CONSTRAINTS

We now present the primary contribution of this paper, a new symbolic encoding for interrupt-driven programs with interrupt nesting. The validation methodology and modelling approach are the same as in Sec. 4: the user verifies properties for specific interrupt arrival scenarios, which are modelled by spawning ISR threads at selected points in the program. As with the approach in Sec. 4, we can model multiple arrivals of the same interrupt by creating separate copies of the ISR function body for each one. By doing so, we can decide statically the depth of each invocation of ISRs. But instead of instrumenting the source code to constrain the resulting concurrency interleavings, we encode the concurrent program execution and express the interleaving constraints directly by a SAT/SMT formula.

[1] In many verification tools, atomic sections are supported by a pair of constructs `__VERIFIER_atomic_begin()` and `__VERIFIER_atomic_end()` that indicate the beginning and the end of an atomic section, respectively. The atomic assume statement can be realised by using an atomic section and the `__VERIFIER_assume` primitives.

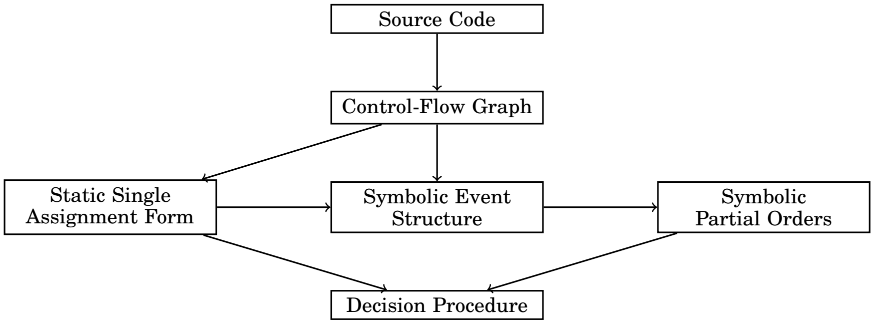


Fig. 5: Overview of encoding into partial-order constraints

The idea is to translate a concurrent program into atomic memory read/write events, and then describe the interleavings of these events that can occur as a symbolic partial order expressed by a formula. The overall encoding is a formula of the form $SSA \wedge PORD$, where conjunct SSA encodes the data and control structure of the program, in the style expounded by Alglave et al. [2013], and conjunct $PORD$ is a predicate that over-approximates the set of all possible interleavings of ISRs that can occur during execution of the program. (We discuss the over-approximation in Sec. 5.5, after presenting the algorithms.) As outlined in Sec. 2, the program may contain assertions for verification, which will be included in SSA. A satisfying assignment to the variables in the formula corresponds to a potential error trace in the interrupt-driven program, identifying an assertion violation. An overview of our method is shown in Fig. 5.

We emphasize that our partial-order encoding for nested interrupts is not to be confused with partial-order *reduction*, which was mentioned in Sec. 3.2.

5.1. Static Single Assignment Form

The first conjunct SSA in our encoding is built in the same way as is employed by Alglave et al. [2013], which is based on a variant of the logical encoding of the *static single assignment* (SSA) form of Clarke et al. [2003]. In SSA, each occurrence of a program variable is annotated with an index; assignments can then be turned into equalities, with the indices yielding distinct variables in the resulting equation. For example, the assignment `x := x + 1` results in the equality $x_1 = x_0 + 1$. Unique indices are also used in loop unwinding; for example, iterating `x := x + 1` twice gives $x_1 = x_0 + 1$ and $x_2 = x_1 + 1$. Each equation has a *guard*, which is the disjunction over the conjunctions of the branching conditions that occur along all the program paths leading to the assignment statement. So guards encode the control flow of the program. For more information, including the treatment of pointers, see Clarke et al. [2003].

For a concurrent program, a distinct index is used for each occurrence of a shared variable. Consequently, the connection between reads and writes across concurrent components of the program is broken. Reading a shared variable is then non-deterministic, over-approximating the behaviours of the program. For example, in the program given in Fig. 6, `ISR_B` reads the value of global variable `x` and uses it to compute `b`. Depending on whether `ISR_B` starts its execution before `ISR_A`, `x` in `ISR_B` can get value x_1 from `ISR_A` or x_0 from the initialisation step. As both scenarios are possible, we use a new index x_2 to capture both cases in the encoding. The construction of $PORD$ then includes constraints to enforce consistency of corresponding reads and writes [Alglave et al. 2013]. The following two subsections elaborate on how to construct $PORD$.

```
volatile unsigned x = 0, y = 0;             x_0 = 0  ∧  y_0 = 0
volatile unsigned a = 0, b = 0;             a_0 = 0  ∧  b_0 = 0

void* ISR_A(void* arg) {
  x = 1;                                    x_1 = 1
  a = y + 1;                                a_1 = y_1 + 1
}

void* ISR_B(void* arg) {
  async: ISR_A(); // spawn ISR_A thread
  y = 1;                                    y_2 = 1
  b = x + 1;                                b_1 = x_2 + 1
}

void main() {
  async: ISR_B(); // spawn ISR_B thread
  assert(!(a == 1 && b == 1));              a_2 = 1  ∧  b_2 = 1
}
```

Fig. 6: An example of the static single assignment form for concurrent programs

$x_0 = 0 \;\wedge\; y_0 = 0$	(m1) W x 0	(m2) W y 0
$a_0 = 0 \;\wedge\; b_0 = 0$	(m3) W a 0	(m4) W b 0
$x_1 = 1$	(A1) W x 1	
$a_1 = y_1 + 1$	(A2) R y y_1	(A3) W a a_1
$y_2 = 1$	(B1) W y 1	
$b_1 = x_2 + 1$	(B2) R x x_2	(B3) W b b_1
$a_2 = 1 \;\wedge\; b_2 = 1$	(m5) R a a_2	(m6) R b b_2

Fig. 7: Symbolic memory events for the program in Fig. 6

5.2. Symbolic Events

Once the SSA encoding SSA is obtained, we construct a collection of *symbolic memory events* based on the occurrences of variables being read or written. A symbolic memory event has four parts: a unique identifier, a direction (read or write), the memory location that is read or written, and a value, which may be given by a symbolic expression containing variables.

Consider the example in Fig. 7. The SSA equation $a_1 = y_1 + 1$ arises from the statement `a = y + 1` in `ISR_A`. This generates two symbolic memory events: (A2) R y y_1 and (A3) W a a_1. Event A2 reads memory location `y` (i.e. the variable `y` in the original program) with symbolic value y_1. Event A3 writes to the memory location `a` with symbolic value a_1, which is equal to $y_1 + 1$ as given in the SSA equation.

5.3. Partial-Orders for Nested Interrupts

Using the symbolic memory events just described, we construct a symbolic partial order as a formula $PORD$ that over-approximates the set of all possible interleavings of reads and writes that can occur in any execution of the original program. The motivation for making an over-approximation is to reduce the size of $PORD$ by using data dependencies to define control dependencies in program executions. As will be seen in Sec. 5.5, this is still precise for non-nested interrupts—but it may signal false alarms in some more complex scenarios. We believe, however, that it is still precise enough in practice, which is confirmed in our experimental studies in Sec. 6.

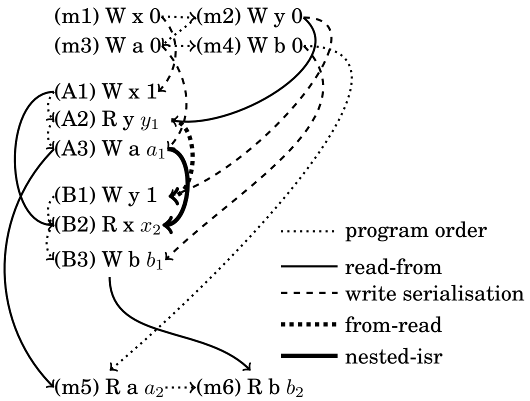


Fig. 8: Event graph for the example program in Fig. 6

Our encoding is an extension of the framework explained by Alglave et al. [2012]. It is based on the idea of an *event graph* of reads and writes to shared memory locations. An event graph represents a set of candidate executions of the program with a particular set of control-flow choices for each thread, in which memory reads and writes happen in a certain partially-constrained order between different threads. There may be many different event graphs consistent with a given program, each describing different interrupt-arrival scenarios. The nodes of an event graph are the symbolic memory events of the program. For clarity of presentation, we sometimes just write the unique identifier component of a symbolic memory event as a node, it being understood that the other three components are obtainable from the identifier they are associated with. The edges of an event graph model a particular *global happens-before* order of the memory events that occur across the main program and the ISR threads.

There are five kinds of edges in an event graph that indicate specific types of ordering relationship between events: *program order* (po), *write serialisation* (ws), *read-from* (rf), *from-read* (fr), and *nested-isr* (ni). An event graph is said to be *valid* if po $\cup$ ws $\cup$ rf $\cup$ fr $\cup$ ni is acyclic—i.e., there are no cyclic dependencies among the read and write events in the event graph. One valid event graph for the example program introduced in Sec. 5.2 is given in Fig. 8. This event graph corresponds to the arrival and handling of interrupt A during the handling of interrupt B, just after execution of the first assignment statement in ISR_B.

We now explain each relation in detail. We write $\mathrm{thread}(e)$ to denote the thread that generates memory event e, whether of an ISR or the main program. Note that memory events e and e' may belong to the same ISR, but this does not necessarily mean that $\mathrm{thread}(e) = \mathrm{thread}(e')$, as they may belong to two different copies of the ISR function body—i.e. two different threads. We define $\mathrm{depth}(e)$ to be the execution depth of the thread that contains e, the same way we define depth in Sec. 4.

- The relation *program order* (po) is defined by the textual order of instructions within the code for each thread, or the main program. It relates only memory events that belong to the same thread. In the example event graph given in Fig. 8, the events within each handler and the main program are ordered as indicated by the numeric suffix of their identifiers.
- The relation *write serialisation* (ws) is the union of a set of total orders, each of which orders all the write events that access a specific memory location. For example, in the event graph in Fig. 8, ws = {(m1, A1), (m2, B1), (m3, A3), (m4, B3)}. The ws

relation represents an order in which the write events to each memory location occur during an execution of the program and the modelled interrupts.

— The relation *read-from* (rf) relates each read event to a unique write event from which it is assumed that the read gets its value. More precisely, for all read events (r) R x a there is unique write event (w) W x b which accesses that same memory location x such that $(w, r) \in$ rf. As will be seen later, each such edge in the graph adds a constraint $a = b$ to the overall partial-order encoding—because, of course, r must read the same value that was written by w. In our example event graph, read event A2 reads the value of memory location y that is written by write event m2.

— The relation *from-read* (fr) contains certain read-write event pairs (r, w) that access the same memory location and thereby constrain the read r to happen before the write w. It is constructed as follows. Suppose there is a pair (w, r) in the *read-from* relation rf. That is, it is assumed that r reads the value written by w. Now consider any write w' that occurs *after* w, according to the write serialisation relation ws. The read event r must happen *before* the write event w', for otherwise r would have gotten its value from w' instead of w. We express this constraint by having $(r, w') \in$ fr. In our example, (m2, A2) $\in$ rf and (m2, B1) $\in$ ws, so (A2, B1) $\in$ fr.

Finally, we define the *nested-isr* (ni) relation. This represents constraints on the order of memory events that arise due to the program semantics of interrupt-handling. Recall from Sec. 2 that all interrupts are masked by the hardware when an ISR is entered. The ISR can then selectively enable or disable individual interrupts, before lifting the mask. If ISR i enables interrupt j, which subsequently arrives during the execution of ISR i, the execution of ISR j must finish before the execution of ISR i can resume. The *nested-isr* relation captures this semantics.

Formally, the relation *nested-isr* (ni) is defined as follows. For all memory events e, e' such that $\mathrm{thread}(e) \neq \mathrm{thread}(e')$ and $\mathrm{depth}(e) \leq \mathrm{depth}(e')$, if $(e', e) \in$ ws $\cup$ rf $\cup$ fr, then for all events e'' such that $(e', e'') \in$ po, we put $(e'', e) \in$ ni. Intuitively, suppose e' is in a thread executing ISR j and e' happens before e, which is in a thread with equal or lower depth that is executing ISR i. This means that ISR i has been interrupted by ISR j, which generates memory event e' before execution of ISR i has reached memory event e. Since ISR j must exit before control is returned to ISR i, any memory events e'' in the execution of ISR j that occur after e' in program order must also happen before e. So we have $(e'', e) \in$ ni. In our example, interrupt A has arrived and is handled during the execution of `ISR_B`, indicated by (A1, B2) $\in$ rf. Then `ISR_A` must complete its execution before the control goes back to `ISR_B`, so we have (A3, B2) $\in$ ni. This is depicted as a solid, bold edge in Fig. 8.

5.4. Algorithms for Computing Constraints for Symbolic Partial Orders

We now elaborate on the algorithms for computing the formula $PORD$, which characterises a set of possible interleavings among the ISRs and the main program under analysis. We can view $PORD$ as the set of all valid event graphs that are consistent with the structure of the program and with the semantics of the interrupt scenarios with which we have instrumented the program. As we shall see later, $PORD$ is a slight over-approximation of this set; that is, it admits some event orderings that cannot occur in any real program execution. This will be explained in detail in Sec. 5.5.

Algorithms 1–3 take as input the set of all the symbolic events of a program with nested interrupts and its program order po. They produce as output sets of constraints that characterise the four relations between symbolic events in a valid event graph for the program. The formula $PORD$ is the conjunction of all these constraints. Each algorithm computes a set of constraints for one of the relations *write serialisation* (ws), *read-from* (rf), or *from-read* (fr). In addition, each algorithm computes one part

Algorithm 1: Constraints for *write serialisation*

```
Input  : A set of symbolic events and program order po
Output : Sets of constraints $C_{ws}$ and $C_{ni\_ws}$
1 $C_{ws} \leftarrow \emptyset$; $C_{ni\_ws} \leftarrow \emptyset$
2 $writes \leftarrow \{(\alpha, \{w_1 \ldots w_n\}) \mid w_i$ is a write event $\wedge\ \mathrm{mloc}(w_i) = \alpha\}$
3 for $\alpha$, $W$ s.t. $(\alpha, W) \in writes$ do
4     for $w, w' \in W$ s.t. $\mathrm{thread}(w) \neq \mathrm{thread}(w')$ do
5         $C_{ws} \leftarrow C_{ws} \cup \{w \rightarrow w' \vee w' \rightarrow w\}$
6         if $\mathrm{depth}(w) \geq \mathrm{depth}(w')$ then
7             $C_{ni\_ws} \leftarrow C_{ni\_ws} \cup \{(w \rightarrow w' \wedge \mathrm{grd}(w) \wedge \mathrm{grd}(w')) \Rightarrow \bigwedge_{e \in \mathrm{last}(w)} e \rightarrow w'\}$
8         if $\mathrm{depth}(w) \leq \mathrm{depth}(w')$ then
9             $C_{ni\_ws} \leftarrow C_{ni\_ws} \cup \{(w' \rightarrow w \wedge \mathrm{grd}(w') \wedge \mathrm{grd}(w)) \Rightarrow \bigwedge_{e \in \mathrm{last}(w')} e \rightarrow w\}$
```

of the constraints for the *nested-isr* (ni) relation—namely those constraints on ni that arise from the *other* constraints that the algorithm computes. Our algorithms generate the same constraints for relations ws, rf, and rf as in Alglave et al. [2013]. The novel extension we have made in this work to adapt this framework to the new setting of interrupts is to use these constraints to generate further constraints for the ni relation.

In what follows, we write $\mathrm{mloc}(e)$ for the memory location of event e, $\mathrm{val}(e)$ for the symbolic value of event e, and $\mathrm{grd}(e)$ for the guard of the SSA equation from which e is derived. For every event e, we introduce an integer variable ck_e, which is the *clock* of e. Given two memory events e and e', our encoding will then express that event e happens before event e' with the constraint $ck_e < ck_{e'}$. We write $e \rightarrow e'$ as a shorthand for the formula $\mathrm{grd}(e) \wedge \mathrm{grd}(e') \Rightarrow ck_e < ck_{e'}$.

Algorithm 1 computes the constraints C_{ws} on the relation *write serialisation* (ws) and constraints C_{ni_ws} on the part of the *nested-isr* relation that derives from event orderings in ws. Specifically, it adds, for each pair of write events w and w' that access the same memory location and do not belong to the same thread (lines 2–3), the constraint that either $(w, w') \in$ ws or $(w', w) \in$ ws (line 5). Because of the definition of $e \rightarrow e'$, the disjunction is exclusive. So a satisfying assignment of values to the integer clock variables in this constraint constitutes a set of (guarded) total orders on all write events that access each location α.

If a write w happens before another write w' and $\mathrm{depth}(w') \leq \mathrm{depth}(w)$, then according to our definition of the *nested-isr* relation, all memory events e after w in the program order must also happen before w'. This is expressed by $\bigwedge_{e \in \mathrm{last}(w)} e \rightarrow w'$ in line 7, where

$$\mathrm{last}(e) = \{e' \mid (e, e') \in \mathrm{po} \wedge \neg\exists e''.\mathrm{thread}(e') = \mathrm{thread}(e'') \wedge (e,' e'') \in \mathrm{po}\}$$

Note that we state only the last events e in the program order of the ISR that contains w. This is an optimisation, as all memory events between w and any e in program order will happen before w' by transitivity. Symmetrically, we also impose a similar constraint in cases where $\mathrm{depth}(w) \leq \mathrm{depth}(w')$ (line 9).

Algorithm 2 computes constraints C_{rf} on the relation *read-from* (rf) and constraints C_{ni_rf} on the part of the *nested-isr* relation deriving from event orderings in rf. In addition to specifying orders between write and read events, it connects the symbolic *value* of each read event to the value written by one of the possible writes. For each pair of a write w and a read r that access the same memory location (lines 2–7), we introduce a fresh Boolean variable s_{wr} (line 9) to indicate that r in fact does read the value written by this w (line 10) as well as say that w must happen before r (line 11). Technically, this is achieved by enforcing $\mathrm{val}(r) = \mathrm{val}(w)$ in line 10. The vector of all the s_{wr} variables

Algorithm 2: Constraints for *read-from*

```
Input  : A set of symbolic events and program order po
Output : Set of constraints $C_{rf}$, $C_{ni\_rf}$
1  $C_{rf} \leftarrow \emptyset$; $C_{ni\_rf} \leftarrow \emptyset$
2  $reads \leftarrow \{(\alpha, \{r_1 \ldots r_n\}) \mid r_i$ is a read event $\wedge\ \mathrm{mloc}(r_i) = \alpha\}$
3  $writes \leftarrow \{(\alpha, \{w_1 \ldots w_n\}) \mid w_i$ is a write event $\wedge\ \mathrm{mloc}(w_i) = \alpha\}$
4  for $\alpha$, $R$, $W$ s.t. $(\alpha, R) \in reads \wedge (\alpha, W) \in writes$ do
5      for $r \in R$ do
6          $tmp \leftarrow \emptyset$
7          for $w \in W$ do
8              if $(r, w) \notin po$ then
9                  $tmp \leftarrow tmp \cup \{s_{wr}\}$
10                 $C_{rf} \leftarrow C_{rf} \cup \{s_{wr} \Rightarrow (\mathrm{grd}(w) \wedge \mathrm{grd}(r) \wedge \mathrm{val}(w) = \mathrm{val}(r))\}$
11                 $C_{rf} \leftarrow C_{rf} \cup \{s_{wr} \Rightarrow w \rightarrow r\}$
12                 if $\mathrm{thread}(w) \neq \mathrm{thread}(r) \wedge \mathrm{depth}(w) \geq \mathrm{depth}(r)$ then
13                     $C_{ni\_rf} \leftarrow C_{ni\_rf} \cup \{(w \rightarrow r \wedge \mathrm{grd}(w) \wedge \mathrm{grd}(r)) \Rightarrow \bigwedge_{e \in \mathrm{last}(w)} e \rightarrow r\}$
14         $C_{rf} \leftarrow C_{rf} \cup \{\mathrm{grd}(r) \Rightarrow \bigvee_{s \in tmp} s\}$
```

Algorithm 3: Constraints for *from-read*

```
Input  : A set of symbolic events and program order po
Output : Set of constraints $C_{fr}$, $C_{ni\_fr}$
1  $C_{fr} \leftarrow \emptyset$; $C_{ni\_fr} \leftarrow \emptyset$
2  $reads \leftarrow \{(\alpha, \{r_1 \ldots r_n\}) \mid r_i$ is a read event $\wedge\ \mathrm{mloc}(r_i) = \alpha\}$
3  $writes \leftarrow \{(\alpha, \{w_1 \ldots w_n\}) \mid w_i$ is a write event $\wedge\ \mathrm{mloc}(w_i) = \alpha\}$
4  for $\alpha$, $R$, $W$ s.t. $(\alpha, R) \in reads \wedge (\alpha, W) \in writes$ do
5      for $(w, w') \in W \times W$, $r \in R$ s.t. $w \neq w' \wedge \mathrm{thread}(r) \neq \mathrm{thread}(w')$ do
6          $C_{fr} \leftarrow C_{fr} \cup \{(s_{wr} \wedge w \rightarrow w' \wedge \mathrm{grd}(w')) \Rightarrow r \rightarrow w'\}$
7          if $\mathrm{depth}(r) \geq \mathrm{depth}(w')$ then
8              $C_{ni\_fr} \leftarrow C_{ni\_fr} \cup \{(s_{wr} \wedge w \rightarrow w' \wedge \mathrm{grd}(w')) \Rightarrow \bigwedge_{e \in \mathrm{last}(r)} e \rightarrow w'\}$
```

introduced in this part of the algorithm constitutes a symbolic index into all the possible ways in which reads get their values from writes. Note that the disjunction in line 14 technically allows a read event to get its value from multiple writes. However, with constraints of relation *from-read* described in Algorithm 3, this will be prevented. To see this, suppose $(w, r), (w', r) \in$ rf and $(w, w') \in$ ws. Then we have $(r, w') \in$ fr, forming a cycle between r and w'. This is not a valid execution. The constraints related to nested interrupts are generated in lines 12–13 in the same way as in Algorithm 1.

Finally, **Algorithm 3** constructs constraints C_{fr} for the relation *from-read* (fr) and constraints C_{ni_fr} for the part of the *nested-isr* relation derived from event orderings in fr. For every read-write pair (r, w) that accesses the same memory location α (lines 2–4), we consider all other writes w' that accesses this memory location too and happen after w (expressed by $w \rightarrow w'$ line 6). Recall that in Algorithm 2 we introduce the Boolean variable s_{wr} to say that $(w, r) \in$ rf. So if r does indeed get its value from w, then we require $r \rightarrow w'$ to hold. That is r must happen before w', for otherwise it would get its value from w'. The constraints related to nested interrupts are generated in lines 7–8 in a way similar to those in Algorithms 1 and 2.

In the special case where interrupts have explicit priorities, we take an approach similar to those used in the program transformation methods discussed in the previous sections. A global array irq_pty is used to record the priority of each interrupt, and we define the priority of a memory event $\mathrm{pty}(e)$, to be the priority of the ISR function body it belongs to. Then to handle explicit priorities in our encoding, we only need to replace any occurrence of $\mathrm{depth}(e)$ with $\mathrm{pty}(e)$ in our algorithms.

Finally, it is worth discussing a precise encoding that represents exactly the set of all possible interleavings of ISRs that can occur during program execution. It is a straightforward modification of the partial-order encoding, where the constraints for the *nested-isr* (ni) relation consider events that can access *different* memory locations. By contrast, the ni relation of the partial-order encoding only considers events that access the same memory locations in the ws, rf, and the fr relations. More precisely, for each pair of ISRs i and j such that the execution depth of ISR i is smaller than or equal to that of ISR j, the *nested-isr* (ni) constraints of the precise encoding are as follows: for each memory event e of ISR i and the first memory events e' of ISR j, which can either access the same or different memory locations, if e' happens before e, then for all memory events e'' after e' in ISR j, e'' must happen before e, encoded as $(e' \to e \wedge \mathrm{grd}(e') \wedge \mathrm{grd}(e)) \Rightarrow \bigwedge_{e'' \in \mathrm{last}(e')} e'' \to e$. Although the edge (e', e) may not belong to any of the partial-order relations, the verification condition $e' \to e \wedge \mathrm{grd}(e') \wedge \mathrm{grd}(e)$ is checked by the underlying solver used in an algorithm that computes the constraints. Since the precise encoding relates events that access the same and different memory locations, it produces constraints that are potentially larger than those of the partial-order encoding. In Sec. 6, we experimentally compare these two encodings.

5.5. Analysis of the Algorithms

Our encoding does not relate events in the same ISR call as they are already related by program order, except for *read-from* as a read event can get its value from a write event in the same ISR call. The (asymptotic) size of our encoding is, in the worst case, quadratic for *write serialisation* and *read-from*, and cubic for *from-read*, in the maximum number of memory events for a *single* address. Note that the encoding produced by our algorithm is by no means minimal in terms of size. For example, line 13 of Algorithm 2 can, in principle, exclude cases where all of the following hold: w and r do not belong to the same ISR call, depth(r) $\leq$ depth(w) is true, and w is the last write event in the program order s.t. the conjunction of guards $\mathrm{grd}(w)$ and $\mathrm{grd}(r)$ is satisfiable. However, in each iteration, we would need to check whether $\mathrm{grd}(w) \wedge \mathrm{grd}(r)$ is satisfiable, which can be quite expensive. We need to balance the size of the encoding and the computational cost of producing it.

As discussed in Sec. 2, executions of an interrupt-driven program with n interrupts should satisfy the following: for any pair of ISRs i and j, if ISR i preempts the execution of ISR j, then ISR i must finish its execution before the control can return to ISR j. In terms of our partial-order relation framework, this means if an execution represented by an event graph of the program satisfies the following: if 1) the depth of ISR i is greater than or equal to the depth of ISR j, and 2) event e of ISR i happens before event e' in ISR j, then for all events e'' in ISR i, we have e'' happens before e'. We say that this execution satisfies the *nested-execution property*.

Since our encoding is an over-approximation of the set of all possible program executions, if a safety property holds in our partial-order encoding, i.e. the property holds in every execution that satisfies the encoding, it also holds in every execution of the original program. On the other hand, if a safety property is violated in our partial-order encoding, it may not be violated in the original program. We exhibit one such example that violates the nested-execution property in Fig. 9.

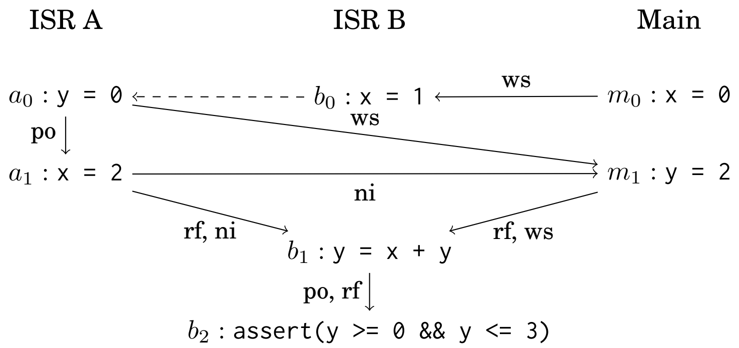


Fig. 9: Counterexample

In this example, the execution of the main program is preempted by ISR B, which is then preempted by ISR A. Since a_0 happens before m_1 represented by a ws edge, according to the definition of ni, we have an ni edge from a_1 to m_1. Similarly, since b_1 gets its x value from a_1, there is an ni edge from a_1 to b_1. However, there is no ws, rf, or fr edge from ISR B to the main program, so there is no ni edge to ensure the execution of ISR B finishes before the control can return to the main program. Note that the precise encoding would have an ni constraint indicating b_2 happens before m_1, since b_0 happens before m_1, even though they access different memory locations. Then there is a cycle $m_1 \to b_1 \to b_2 \to m_1$, excluding this counterexample.

For the non-nested case where the main program has only one ISR, a safety property violated in the partial-order encoding will also be violated in the original program. More precisely, we have the following theorem:

Theorem. For an event graph of a program with one ISR that satisfies the partial-order constraints *program order* (po), *write-serialisation* (ws), *read-from* (rf), *from-read* (fr), and *nested-isr* (ni), there exists a program execution that satisfies the nested-execution property.

Proof. Let a and m be the first event in ISR A and the main program, respectively, such that (a, m) belongs to one of the ws, rf, and fr relations. By the definition of ni, for all events that happen after a in ISR A will happen before m. For events a' and m' that happen before a and m in ISR A and the main program, respectively, they either access different memory locations or read the same memory location. Thus, their ordering does not matter, we can have m' happens before a_0 and a' happens before m as shown in Fig. 10. Then the execution satisfies the nested-execution property.[2] ■

Our experimental results in Sec. 6 show that the partial-order encoding obtains the same verification results as the precise encoding for all our benchmarks, while the runtime is faster on all but one of the benchmarks.

6. EXPERIMENTS

We compare our new partial-order encoding, described in the preceding section, with two conventional source-to-source transformations. These translate interrupt-driven code into sequential code and into multi-threaded code, respectively.

[2] For the correctness proof of relations write-serialisation, read-from, and read-from, please refer to the extended version of the Alglave et al. [2013] at https://arxiv.org/abs/1301.1629.

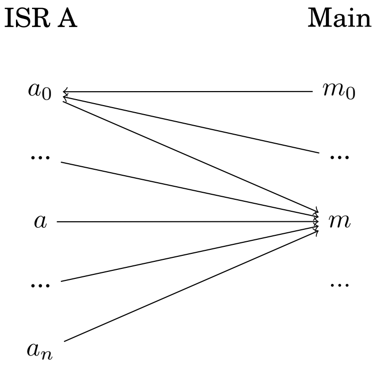


Fig. 10: ISR A preempts the main program right before event m

6.1. Verification Tools

To evaluate the performance of our partial-order encoding for nested interrupts, we implemented our algorithms in a prototype tool called *i-CBMC*, which is an extension of CBMC [Clarke et al. 2004]. Moreover, we apply a range of existing verifiers to sequentialised programs and programs that have been instrumented with threads.

For *interrupt sequentialisation* as described in Sec. 3, we used the following tools:

— The sequential version of CBMC [Clarke et al. 2004], a bounded model checker for the C programming language;
— SatAbs [Clarke et al. 2005], BLAST [Shved et al. 2012], and CPAchecker [Dangl et al. 2015], which implement counterexample-guided abstraction refinement (CEGAR) [Clarke et al. 2000];
— UFO [Albarghouthi et al. 2012d], which implements interpolation-based verification in an abstraction refinement loop [Albarghouthi et al. 2012a; 2012b; 2012c].

For *modelling interrupts with threads* as discussed in Sec. 4, we used:

— The concurrent version of CBMC [Alglave et al. 2013];
— IMPARA [Wachter et al. 2013], which combines a new, symbolic form of partial-order reduction with the Impact algorithm [McMillan 2006];
— ESBMC [Morse et al. 2014], a context-bounded symbolic model checker for multi-threaded C programs;
— Lazy-CSeq [Inverso et al. 2015], a tool that translates a multi-threaded C program into a sequential C program with a given bound on the schedule rounds;
— The concurrent version of SatAbs [Clarke et al. 2005];
— Threader [Popeea and Rybalchenko 2013], which implements thread-modular reasoning [Gupta et al. 2011].

The tools were run using the SV-COMP17 configuration (http://sv-comp.sosy-lab.org/2017/rules.php). We make our benchmarks and detailed experimental data available for other researchers at http://www.cprover.org/interrupts/. The source code of i-CBMC can be found at https://github.com/lihaol/cbmc/tree/i-cbmc.

6.2. Benchmarks

We assess the effectiveness of each method on benchmarks derived from embedded software and Linux device drivers. For each benchmark, we have a version where interrupts are correctly disabled and enabled, and another version where they are incorrectly managed so that safety properties are violated.

	LOC	Interrupts	Scheduler Calls	**Interrupt Sequentialisation**			
				BLAST 2.7.3	CPAChecker 1.6.12	UFO SV-COMP14	CBMC ea3a21f
Logger	112	2	8	✓ 17.7 s	✓ 2.7 s	✓ **1.4 s**	TO
+ incorrect	112	2	9	! **0.5 s**	! 7.0 s	! 36.2 s	TO
Logger (extended)	172	3	22	* 0.4 s	unknown	TO	TO
+ incorrect	172	3	18	! 84.1 s	! **9.1 s**	TO	TO
Blink	2,652	2	193	✗	unknown	✓**1425.1 s**	TO
+ incorrect	2,652	2	198	✗	unknown	**?** 1420.5 s	TO
RcCore	7,035	3	347	* 2.7 s	unknown	TO	TO
+ incorrect	7,035	3	347	! **2.7 s**	unknown	TO	✗
Brake (1 Wheel)	3,938	2	6	* 0.1 s	unknown	✓ **0.8 s**	✓ 80.8 s
+ incorrect	3,938	2	7	TO	unknown	**?** 1.1 s	✗
Brake (2 Wheels)	3,938	3	10	TO	unknown	✓ **7.8 s**	TO
+ incorrect	3,938	3	7	TO	unknown	**?** 1.1 s	TO
Brake (3 Wheels)	3,938	4	14	TO	unknown	✓ **854.1 s**	TO
+ incorrect	3,938	4	9	TO	unknown	**?** 3.8 s	TO

✓= proved correct, ! = bug exposed, **?** = bug missed, * = false alarm
✗ = tool crashes, PE = parse errors, TO = timeout 1800 s

Table I: Experimental results of interrupt sequentialisation

Logger models parts of the firmware of a temperature logging device from a major industrial enterprise (112 LOC, and 172 LOC for the extended version). It has two interrupt-triggered tasks: the measurement task must not be preempted by the communication task, or a measurement is written to a wrong position in the log.

Blink is an application from TinyOS. It displays a pattern on three LEDs with the help of timers. TinyOS provides a virtualisation API to handle several alarms with one hardware timer. There are two interrupt-triggered tasks, for firing the alarm and updating the timer when it overflows. If the alarm firing can preempt the updating of the timer, the timer keeps running even though the alarm has already fired.

Brake is generated from the Simulink model of a brake-by-wire system provided by Volvo Technology AB (3938 LOC). It consists of four threads communicating with four wheel brake controllers, together with one main thread, responsible for computing the braking torque, which should not be preempted. Otherwise, the brake torque can be miscalculated. We parameterised this benchmark by the number of wheels.

RcCore is a Linux device driver for a remote control together with a model of the Linux kernel (7035 LOC). The original driver uses locks to enforce priorities—namely that querying the availability of transmission protocols may not be interrupted by a modification to the protocol settings. We introduced a bug such that the driver may crash due to a NULL pointer dereference if priorities are disregarded.

We performed all experiments on a 64-bit machine running Linux with eight Intel Xeon 3.07 GHz cores and 48 GB of main memory.

6.3. Results and Discussion

Our results indicate that the transformation to sequential code performs worst, whereas our partial-order encoding is clearly more effective than the two program transformation techniques. We discuss our experimental results of each approach in more detail below.

Interrupt Sequentialisation. Table I presents the results of our interrupt sequentialisation. Column 1 lists our five benchmarks: Logger, extended Logger, Blink, RcCore and Brake (parameterised by the number of wheels). Columns 2–4 give the number of lines of code, the number of interrupt arrivals we modelled, and the number of scheduler calls after our optimisation (Sec. 3) in each benchmark. Column 5 gives results for the interrupt sequentialisation. SatAbs crashed on all our benchmarks so we did not include them in the table. The best correct entry for each benchmark is marked in bold.

In our experiments with interrupt sequentialisation, the number of interrupt arrivals is bounded. All sequential tools we tried except CBMC could handle both versions of the simplest benchmark Logger. CBMC timed out on this benchmark. For other benchmarks, the tools we tried could obtain only partial results. For the extended Logger benchmark, both UFO and CBMC timed out on the correct and faulty versions. BLAST reported a safety violation on both versions of this benchmark, whereas CPAchecker reported unknown and safety violation on the correct and faulty versions, respectively. For the Blink benchmark, only UFO was able to prove safety on the non-faulty version in about 24 minutes; but it failed to report a safety violation on the faulty one. BLAST crashed, CPAchecker returned unknown, and CBMC timed out on both versions of this benchmark. For the RcCore benchmark, only BLAST managed to report a safety violation in about 3 seconds; but it failed to prove safety on the non-faulty version. CPAchecker returned unknown and UFO timed out on both versions. Finally, UFO and CBMC were able to prove safety of the correct versions of the Brake benchmark in 1 s and 81 s, respectively, whereas BLAST returned a counterexample. None of the tools we tried was able to report a safety violation on the faulty version. It is worth noting that for most benchmarks, CBMC was not able to finish unwinding the recursion after 30 minutes. The reason is that non-determinism is encoded directly into a SAT formula.

Modelling Interrupts with Threads. Table II presents the results of modelling interrupts with threads. SatAbs crashed and Threader reported parse errors on all our benchmarks so we did not include them in the table. The results are better compared to sequentialisation: CBMC performed the best among the tools we have tried. It managed to report the expected results for most versions of our benchmarks. It timed out on the correct version of the Brake benchmark with one and three wheels, In addition, it failed to return a safety violation on the faulty version of the Blink benchmark. We suspect this may be caused by a potential bug in the current CBMC implementation, since IMPARA managed to report a safety violation on the faulty Blink. IMPARA was also able to return the expected results of the simplest Logger benchmark. For other benchmarks, it either timed out or crashed. ESBMC was able to prove safety on the correct Logger, extended Logger, and the Blink benchmarks; but it missed the bug on their incorrect versions. It could not parse the Linux device driver RcCore. For the Brake benchmark, it returned a safety violation on all its incorrect versions, and a false alarm on the correct ones. Finally, Lazy-CSeq managed to prove safety on the correct Logger, and report a safety violation on the incorrect extended Logger. But it reported a false alarm on the correct extended Logger, and unknown for other benchmarks.

In the thread instrumentation approach, each thread requires a separate copy of its ISR function, so modelling with threads can verify only a bounded number of interrupts, whereas interrupt sequentialisation can model an unbounded number of interrupts.

			Modelling Interrupts with Threads			
	LOC	Interrupts	IMPARA 0.5.0	ESBMC 4.4.0-incr	Lazy-CSeq SV-COMP16	CBMC ea3a21f
Logger	112	2	✓ 0.5 s	✓ **0.1 s**	✓ 12.2 s	✓ 0.4 s
+ incorrect	112	2	! **0.6 s**	? 0.2 s	unknown	! 1.5 s
Logger (extended)	172	3	TO	✓ **9.5 s**	* 17.6 s	✓ 399 s
+ incorrect	172	3	TO	? 9.5 s	! **17.6 s**	! 383.2 s
Blink	2,652	2	TO	✓ **1.6 s**	unknown	✓ 4.5 s
+ incorrect	2,652	2	! **212.7 s**	? 2.4 s	unknown	? 19.4 s
RcCore	7,035	3	✗	PE	unknown	✓ **74.9 s**
+ incorrect	7,035	3	✗	PE	unknown	! **73.2 s**
Brake (1 Wheel)	3,938	2	TO	* 0.5 s	unknown	TO
+ incorrect	3,938	2	TO	! **0.6 s**	unknown	! 6.8 s
Brake (2 Wheels)	3,938	3	TO	* 0.8 s	unknown	✓ **669.3 s**
+ incorrect	3,938	3	TO	! **0.8 s**	unknown	! 16.9 s
Brake (3 Wheels)	3,938	4	TO	* 1.0 s	unknown	TO
+ incorrect	3,938	4	TO	! **1.0 s**	unknown	! 39.3 s

✓= proved correct, ! = bug exposed, **?** = bug missed, * = false alarm
✗ = tool crashes, PE = parse errors, TO = timeout 1800 s

Table II: Experimental results of modelling interrupts with threads

			Naïve	Modelling Threads	Precise	PO
	LOC	Interrupts	CBMC ea3a21f	CBMC ea3a21f	i-CBMC -PRE	i-CBMC
Logger	112	2	* 0.2 s	✓ 0.4 s	✓ 0.3 s	✓ **0.2 s**
+ incorrect	112	2	! 0.2 s	! 1.5 s	! 0.3 s	! **0.2 s**
Logger (extended)	172	3	* 13.0 s	✓ 399 s	✓ 41.8 s	✓ **28.1 s**
+ incorrect	172	3	! **13.0 s**	! 383.2 s	! 35.3 s	! 25.9 s
Blink	2,652	2	* 3.4 s	✓ 4.5 s	✓ 26.5 s	✓ **2.0 s**
+ incorrect	2,652	2	! **3.0 s**	? 19.4 s	! 30.5 s	! 3.4 s
RcCore	7,035	3	* 73.4 s	✓ **74.9 s**	✓ 76.7 s	✓ 76.6 s
+ incorrect	7,035	3	! 75.1 s	! 73.2 s	! **56.1 s**	! 79.5 s
Brake (1 Wheel)	3,938	2	* 3.9 s	TO	TO	TO
+ incorrect	3,938	2	! 3.6 s	! 6.8 s	! 4.5 s	! **3.3 s**
Brake (2 Wheels)	3,938	3	* 7.1 s	✓ **669.3 s**	TO	✓ 883.9 s
+ incorrect	3,938	3	! **6.1 s**	! 16.9 s	! 9.1 s	! 7.2 s
Brake (3 Wheels)	3,938	4	* 7.2 s	TO	TO	TO
+ incorrect	3,938	4	! **6.8 s**	! 39.3 s	! 15.5 s	! 8.1 s

✓= proved correct, ! = bug exposed, **?** = bug missed, * = false alarm
✗ = tool crashes, PE = parse errors, TO = timeout 1800 s

Table III: Experimental results of partial-order encoding

Partial-Order Encoding. Overall, *i-CBMC with partial-order encoding* is the most effective and time-efficient, outperforming other approaches on most benchmarks. It timed out in the non-faulty Brake benchmark with one and three wheels, and yields the expected results in all other experiments.

Table III lists the results of running CBMC on the benchmarks without thread instrumentation. CBMC reported false alarms on all correct versions of the benchmarks. This is expectable because without the additional constraints generated by i-CBMC, CBMC would allow interleavings that are not possible in any execution of the program, as described in Fig. 3 of Sec. 2. Table III also compares the results of i-CBMC and CBMC with threads modelling (see the scatter plot in Fig. 11a). Both i-CBMC and CBMC timed out on the correct version of the Brake benchmark with one and three wheels. However, CBMC missed the bug in the faulty version of the Blink benchmark, whereas i-CBMC was able to report a safety violation on it. In addition, the runtime of i-CBMC was a lot faster than CBMC on all but two benchmarks. For example, i-CBMC was more than ten times faster than CBMC on the Blink benchmark.

We also implement the precise encoding described at the end of Sec. 5.4 in a variant of i-CBMC (i-CBMC-PRE). As we can see in Table III, i-CBMC-PRE was able to return the expected results for most versions of our benchmarks except the correct version of the three Brake benchmarks on which it timed out. In general, its runtime is between that of i-CBMC and concurrent CBMC except that it is the slowest on Blink and the fastest on the faulty version of RcCore.

In our modelling of interrupts with threads, we made each line of code atomic, which is an under-approximation of the behaviours of the original code. Our partial-order encoding models reads and writes as different atomic events and hence captures more behaviours. Yet, the runtime of i-CBMC is significantly better than that of CBMC with threads modelling for most variants of our benchmarks. The reason is that the assume statements in the modelling with threads approach increases the size of the SAT formula that CBMC generates.

Similarly, since i-CBMC-PRE implements the precise encoding that relates memory events that access different memory addresses, it generates more constraints than the partial-order encoding in i-CBMC that only relates events with the same memory address, resulting in a slower runtime in general (see the scatter plot in Fig. 11b).

We present the number of constraints (symbolic program expressions and partial orders), variables and clauses of the SAT formulas that CBMC without thread instrumentation (naïve CBMC), i-CBMC, i-CBMC-PRE, and CBMC with thread instrumentation

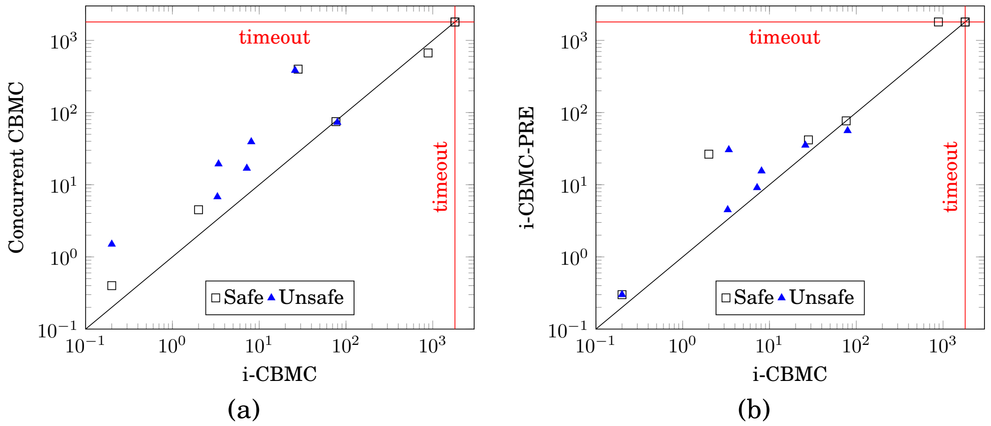


Fig. 11: Scatter plots (runtime in seconds)

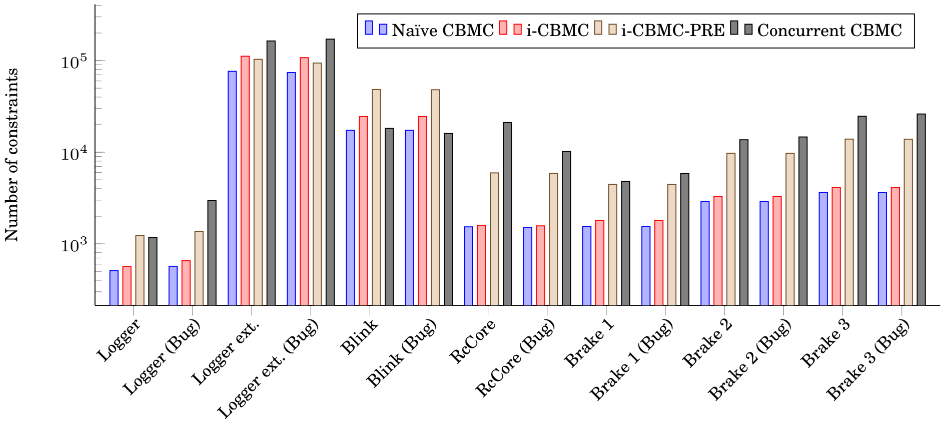


Fig. 12: Number of constraints in the encodings

(concurrent CBMC) produce in Fig. 12, 13a, and 13b, respectively. As we can see, the additional constraints generated by i-CBMC result in a slightly larger formula than those of CBMC *without* thread instrumentation. By contrast, the formula of CBMC *with* thread instrumentation are much larger than those of i-CBMC on most benchmarks. For example, on the extended Logger benchmark, the numbers of i-CBMC are substantially smaller than those of concurrent CBMC, which is reflected by the fact that the runtime of i-CBMC is an order of magnitude faster than concurrent CBMC. Similarly, i-CBMC produces smaller SAT formulas for the Logger, Blink, and the faulty version of the Brake benchmarks, so its runtime is also much lower than that of concurrent CBMC. Both tools performed similarly on the RcCore benchmark, and timed out on the correct version of the Brake benchmark with one and three wheels. The interesting case is the safe version of the Brake benchmark with two wheels. Although i-CBMC yields a similar or smaller encoding, it turns out they are more difficult to solve than those generated by concurrent CBMC (Table III).

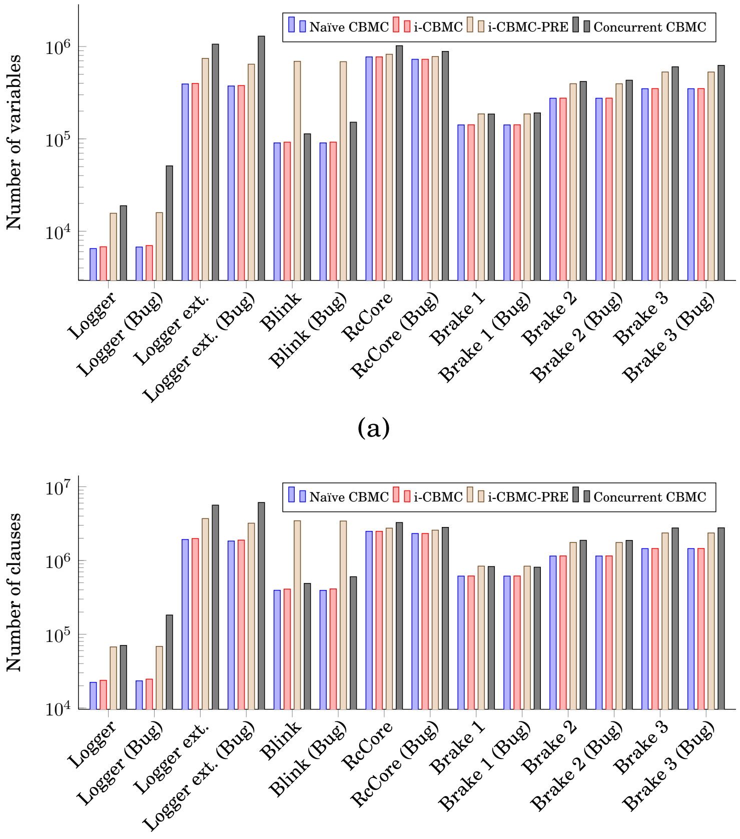
Following the same trend in runtime, the measurements for the formulas that i-CBMC-PRE generated on our benchmarks are in general between i-CBMC and concurrent CBMC. The exceptions are Blink on which i-CBMC-PRE generated the largest formula and had the slowest runtime. And it obtained the fastest runtime on the faulty version of the RcCore.

A limitation of the partial-order encoding is that it is based on bounded model checking, and thus is able to verify only a bounded number of interrupts. By contrast, the sequentialised programs can be passed to unbounded model checkers.

7. RELATED WORK

Previous research on formal verification of interrupt-driven programs uses a range of techniques, including program transformation [Kidd et al. 2010; Regehr and Cooprider 2007; Wu et al. 2013], explicit-state model checking [Schlich et al. 2009], bounded model checking [Bucur and Kwiatkowska 2011; Li et al. 2013] and predicate abstraction [Witkowski et al. 2007]. None of these methods demonstrates effective verification of programs of moderate size with nested interrupts. We briefly survey those methods most closely related to our techniques.

Wu et al. [2013] describe a translation from programs with nested interrupts into sequential code, which makes the assumption that interrupts may arrive after every



(a)

(b)

Fig. 13: Size of the SAT formulas

instruction. This approach suffers from state explosion, which in our translation is addressed by an optimisation that greatly reduces the number of calls to ISRs.

Kidd et al. [2010] introduce a program transformation that translates a multi-threaded program into a sequential one by encoding a thread scheduler. Their method covers threads with fixed priorities, and so is also applicable to interrupt-driven programs with fixed interrupt priorities. Sec. 3 discusses a modified version of their encoding, in which the number of interrupt arrivals is a parameter and multiple arrivals of the same interrupt are allowed. We also provide experimental results for such a transformation. By contrast, Kidd et al. [2010] employ the notion of a “hyperperiod”, during which every interrupt happens exactly once.

Regehr and Cooprider [2007] suggest a source-to-source transformation from interrupt-driven code to multi-threaded code that assigns a fixed priority to a thread when it is created. Threads are scheduled at runtime according to predefined priorities. This approach requires a thread-aware verification tool that can model threads with static priorities. We provide experimental results for a variant of their approach in which the thread priorities are enforced by adding further instrumentation.

Bucur and Kwiatkowska [2011] present a platform-dependent, OS-independent verification tool for programs written in C with interrupts. Their tool automatically translates C code for the MSP430 microcontroller into standard C by modelling the MCU memory map and direct memory access, and adds calls to interrupt handlers to emulate arrivals of hardware interrupts. Partial-order reduction is used to reduce the number of interrupt handler calls. The instrumented program is checked using CBMC. However, nested interrupts are not supported.

Vörtler et al. [2015] propose a verification framework for applications in the embedded operating system Contiki, and present a new modelling approach for periodically occurring interrupts that cannot reply on interrupt reduction such as partial-order reduction. In the experiments, they compare different interrupt modelling approaches on three Contiki applications using CBMC.

Li et al. [2013] propose the use of formal verification techniques in medical device software. Similar to Bucur and Kwiatkowska [2011], they translate embedded code for the MSP430 microcontroller into standard C, which can be verified by CBMC. Then interrupt handlers are called after each basic block of the translated program. Nested interrupts are not allowed.

Schlich et al. [2009] introduce an abstraction technique for microcontroller assembly code called *interrupt handler execution reduction*, which is based on partial-order reduction and reduces the number of program locations where an interrupt handler needs to be executed. Yet, they do not handle nested interrupts.

Sinha and Wang [2010; 2011] use axiomatic specifications of Sequential Consistency (SC) [Lamport 1997] to symbolically encode the executions of concurrent programs. Based on the semantics given in [Alglave et al. 2012], Alglave et al. [2013] introduce a symbolic encoding of concurrent programs that supports not only SC but also weaker memory models such as Intel's x86 and IBM's Power. By contrast, the encoding proposed in this paper is a *strengthening* of SC.

Witkowski et al. [2007] present a tool DDVerify that is also able to generate an accurate sequential and concurrent model of the relevant parts of the Linux kernel, enabling automated verification of Linux device drivers and shared-memory concurrent programs using predicate abstraction. However, when introducing additional threads, the DDVerify framework suffers from the state-space explosion problem. A concurrent model with two threads resulted in a significant increase in the number of predicates required, yielding time-outs for most of the claims.

Schwarz et al. [2014] derive a static analysis technique to detect data races in programs with multiple interrupts of static priorities. Their analysis focuses on programs that use a restricted set of synchronisation patterns with global flag variables. Their results show the technique is effective in reducing the number of race warnings.

Ouadjaout et al. [2016] present a static analysis based on abstract interpretation to verify device drivers of the TinyOS operating system. The proposed method can prove absence of interrupt-related and hardware-software interaction bugs, with several partitioning techniques to help improve precision of their analysis. They compare their tool SADA with i-CBMC on their TinyOS benchmarks. The runtime and memory consumptions of SADA were lower due to their abstraction techniques, whereas i-CBMC implements bit-precise bounded model checking. The major drawback of SADA is false alarms. i-CBMC also produced false alarms because of the imprecise modelling of the hardware and the fixed loop unwinding depth in some of the benchmarks.

Liu et al. [2016] describe a sequentialisation framework for verifying industrial Multifunction Vehicle Bus controller systems. It uses a happen-before interrupt graph to reduce false alarms. The resulting sequential program is verified by CBMC.

8. CONCLUSION

We propose a novel method to verify programs with nested interrupts, which combines symbolic execution and a partial-order encoding. Our encoding captures the interleaving semantics of nested interrupts more precisely than native threads, which allow "zigzag" behaviours between ISRs. We experimentally compare the proposed method to conventional approaches based on source-to-source transformations into sequential or multi-threaded programs, which can be handled by off-the-shelf verification tools. Our experimental results demonstrate that our partial-order encoding is the most effective method to verify software with nested interrupts.

For future work, we will implement a field-sensitive SSA encoding in i-CBMC, which has the potential to greatly reduce the number of shared variables in our partial-order encoding, and will therefore improve scalability. We used finite loop unwindings in our experiments. We plan to implement our method in the model checker IMPARA [Wachter et al. 2013], which is capable of proving unbounded safety.

Acknowledgements

We are grateful for illuminating discussions with Luke Ong (Oxford), Alan Hu (UBC), Moshe Vardi (Rice) and Sharad Malik (Princeton). We thank Ilja Zakharov (ISPRAS) and Doina Bucur (Groningen) for providing the initial version of the RcCore and Blink benchmarks, respectively, and Vincent Nimal (Microsoft) for discussions on CBMC.

REFERENCES

Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. 2012a. Craig Interpretation. In *Proceedings of the 19th International Symposium on Static Analysis (LNCS)*, Vol. 7460. Springer, 300–316.

Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. 2012b. From Under-Approximations to Over-Approximations and Back. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 7214. Springer, 157–172.

Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. 2012c. WHALE: An Interpolation-Based Algorithm for Inter-procedural Verification. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (LNCS)*, Vol. 7148. Springer, 39–55.

Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. 2012d. UFO: A Framework for Abstraction- and Interpolation-Based Software Verification. In *Proceedings of the 24th International Conference on Computer Aided Verification (LNCS)*, Vol. 7358. Springer, 672–678.

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Proceedings of the 25th International Conference on Computer Aided Verification (LNCS)*, Vol. 8044. Springer, 141–157.

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. Fences in Weak Memory Models (Extended Version). *Formal Methods in System Design* 40, 2 (2012), 170–205.

Doina Bucur and Marta Z. Kwiatkowska. 2011. On Software Verification for Sensor Nodes. *Software and Systems* 84, 10 (2011), 1693–1707.

Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. 2001. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles*. ACM, 73–88.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (LNCS)*, Vol. 1855. Springer, 154–169.

Edmund M. Clarke, Orna Grumberg, and Doron Peled. 2001. *Model checking*. MIT Press.

Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 2988. Springer, 168–176.

Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. 2005. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 3440. Springer, 570–574.

Edmund M. Clarke, Daniel Kroening, and Karen Yorav. 2003. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *Proceedings of the 40th Design Automation Conference*. ACM, 368–371.

Matthias Dangl, Stefan Löwe, and Philipp Wendler. 2015. CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic (Competition Contribution). In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 9035. Springer, 423–425.

Patrice Godefroid. 1994. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. Ph.D. Dissertation. University of Liege, Computer Science Department.

Patrice Godefroid and Pierre Wolper. 1991. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In *Proceedings of the 3rd International Workshop on Computer Aided Verification (LNCS)*, Vol. 575. Springer, 332–342.

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Predicate Abstraction and Refinement for Verifying Multi-Threaded Programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 331–344.

Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295.

Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 807–812.

Shmuel Katz and Doron Peled. 1988. An Efficient Verification Method for Parallel and Distributed Programs. In *Proceedings of the Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency (LNCS)*, Vol. 354. Springer, 489–507.

Nicholas Kidd, Suresh Jagannathan, and Jan Vitek. 2010. One Stack to Run Them All – Reducing Concurrent Analysis to Sequential Analysis under Priority Scheduling. In *Proceedings of the 17th International SPIN Workshop on Model Checking Software (LNCS)*, Vol. 6349. Springer, 245–261.

Leslie Lamport. 1997. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.* 46, 7 (1997), 779–782.

Chunxiao Li, Anand Raghunathan, and Niraj K. Jha. 2013. Improving the Trustworthiness of Medical Device Software with Formal Verification Methods. *Embedded Systems Letters* 5, 3 (2013), 50–53.

Han Liu, Yu Jiang, Huafeng Zhang, Ming Gu, and Jiaguang Sun. 2016. Taming Interrupts for Verifying Industrial Multifunction Vehicle Bus Controllers. In *Proceedings of the 21st International Symposium on Formal Methods (LNCS)*, Vol. 9995. Springer, 764–771.

Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification (LNCS)*, Vol. 4144. Springer, 123–136.

Jeremy Morse, Mikhail Ramalho, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. 2014. ESBMC 1.22 (Competition Contribution). In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 8413. Springer, 405–407.

Abdelraouf Ouadjaout, Antoine Miné, Noureddine Lasla, and Nadjib Badache. 2016. Static Analysis by Abstract Interpretation of Functional Properties of Device Drivers in TinyOS. *Software and Systems* 120 (2016), 114–132.

Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 305–318.

Doron Peled. 1994. Combining Partial Order Reductions with On-the-fly Model-Checking. In *Proceedings of the 6th International Conference on Computer Aided Verification (LNCS)*, Vol. 818. Springer, 377–390.

Corneliu Popeea and Andrey Rybalchenko. 2013. Threader: A Verifier for Multi-threaded Programs (Competition Contribution). In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 7795. Springer, 633–636.

John Regehr and Nathan Cooprider. 2007. Interrupt Verification via Thread Verification. *Electronic Notes in Theoretical Computer Science* 174, 9 (2007), 139–150.

Bastian Schlich, Thomas Noll, Jörg Brauer, and Lucas Brutschy. 2009. Reduction of Interrupt Handler Executions for Model Checking Embedded Software. In *Proceedings of the 5th International Haifa Verification Conference on Hardware and Software: Verification and Testing (LNCS)*, Vol. 6405. Springer, 5–20.

Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, and Kalmer Apinis. 2014. Precise Analysis of Value-Dependent Synchronization in Priority Scheduled Programs. In *Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation (LNCS)*, Vol. 8318. Springer, 21–38.

Pavel Shved, Mikhail U. Mandrykin, and Vadim S. Mutilin. 2012. Predicate Analysis with BLAST 2.7 (Competition Contribution). In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 7214. Springer, 525–527.

Nishant Sinha and Chao Wang. 2010. Staged Concurrent Program Analysis. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 47–56.

Nishant Sinha and Chao Wang. 2011. On Interference Abstractions. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 423–434.

Thilo Vörtler, Benny Höckner, Petra Hofstedt, and Thomas Klotz. 2015. Formal Verification of Software for the Contiki Operating System Considering Interrupts. In *Proceedings of the 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems DDECS*. IEEE, 295–298.

Björn Wachter, Daniel Kroening, and Joël Ouaknine. 2013. Verifying Multi-threaded Software with Impact. In *Proceedings of the Internation Conference on Formal Methods in Computer-Aided Design*. IEEE, 210–217.

Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. 2007. Model Checking Concurrent Linux Device Drivers. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ACM, 501–504.

Xueguang Wu, Yanjun Wen, Liqian Chen, Wei Dong, and Ji Wang. 2013. Data Race Detection for Interrupt-Driven Programs via Bounded Model Checking. In *Proceedings of the 7th International Conference on Software Security and Reliability*. IEEE, 204–210.