Modelling Task Priority in Symbolic Predictive Analysis for Embedded Software in Ada

Ranjani Krishnan[1*] and Ashutosh Gupta[2†]

[1*]Flight Computers Group, Indian Space Research Organisation, Trivandrum, 695022, Kerala, India.
[2]Department of Computer Science, Indian Institute of Technology, Bombay, Mumbai, Maharashtra, India.

*Corresponding author(s). E-mail(s): ranjani141@gmail.com;
Contributing authors: akg@iitb.ac.in;
[†]These authors contributed equally to this work.

Abstract

The concurrent, embedded software in safety-critical systems is complex, and analyzing all possible execution paths of such programs is challenging. Symbolic predictive analysis statically analyses a given concrete execution trace of a concurrent program to determine whether any feasible permutation of the given trace violates specified properties. The existing tools do not consider the priority of threads in their modeling of concurrent systems. In this work, we apply a novel static analysis technique based on symbolic predictive anal- ysis for the formal verification of concurrent programs in embedded systems with pre-defined priorities and schedules for the tasks. We consider the given trace as a total order of events coupled with a partial order for the causal model, including several alternative interleavings. Additionally, we include the constraints for the order of event execution by considering the priority of the tasks. We develop a tool chain customized for Ada code, encode the constraints for priority and use it for verifying the onboard software of an aerospace launch vehicle. We accurately model the execution environment for this concurrent software and verify it, along with other benchmarks like concurrency litmus tests and mutual exclusion algorithms. Our experimental results demonstrate the correctness of our modeling and the suitability of our tool for verification of critical Ada software.

Keywords: Formal Verification, Embedded Software, Launch Vehicle, Ada, Task priority

1 Introduction

The concurrent, embedded software in aerospace and other safety-critical systems is becoming increasingly complex. Since numerous combinations of execution paths are possible in concurrent systems, analysis and verification of such programs are challenging. Analyzing all possible execution paths of a concurrent program within a reasonable time is not a trivial task. Different techniques like static analysis, model checking, and predictive analysis are employed for this. Symbolic predictive analysis [1] is a technique that statically analyses a given concrete execution trace of a concurrent program to determine whether any feasible permutation of the given trace violates specified properties. It also provides a feasibility guarantee that all the detected errors are feasible but it is not complete. It uses the information from both the source code of the program and the concrete execution trace to consider all feasible interleavings which can be predicted from the latter.

The embedded software in the onboard computer is one of the most crucial components in safety-critical systems like automotive and aerospace systems. It has to meet real-time requirements, while ensuring minimal execution time and memory footprint. Often, the software is legacy code, written in low-level assembly language or languages like Ada, which are traditionally adopted for high-reliability systems. Bounded model checking provides a method for fast, efficient verification of embedded software.

In this paper, we apply symbolic predictive analysis for verification of concurrent programs in Ada, with a specific priority pre-defined for the tasks. As in [1], we consider the given trace as a total order of events coupled with a partial order for the causal model, including several alternative interleavings. Additionally, we include the constraints for order of event execution by considering the priority of the tasks. For an embedded software with threads running at different periodicities and priorities, the additional constraints for priority are added. For each thread, we impose the constraint that all events in the thread can occur only before the start event or after the final event in all threads of higher priority. We apply the verification method for analysis of a specific launch vehicle onboard software in Ada, with tasks running at two different periodicities and priorities.

To our knowledge, the current tools do not consider the priority of threads in their modeling and most of them support only C/C++ programs. However, in the embedded setting, the priority of the threads is also important for correctness.

The analysis algorithm based on the above technique was developed in C++. A typical safety critical, embedded, aerospace software written in Ada language was chosen as the primary benchmark. This case study was fed as input to the tool, with specifications in a separate file. Verification was done both with and without the priority of the threads considered. The order of event execution in the two cases is different. It was seen that the technique analysed the software correctly when the proper precedence among the threads was included in the encoding. In summary, our contributions are:

- We propose the application of symbolic predictive analysis for verification of concurrent programs with a pre-defined priority for the tasks

- We encode the constraints for priority and use it for verifying the onboard software of a launch vehicle. We accurately model the execution environment for this concurrent software.
- We propose a verification framework for analysis of Ada programs

The rest of this paper is organized as follows: a motivating example is given in Section II and related work is explained in Section III. The preliminaries are described in Section IV. Section V explains the basic algorithm of our implementation followed by our experiments in Section VI. Section VII concludes the paper.

2 Motivating example

Listing 1 Ada program with 2 threads

```
1   procedure thr1proc is
2   begin
3           thr2.var12 := 16#2222#;
4           thr2.var12C := 16#DDDD#;
5   end thr1proc;

6   procedure thr2proc is
7   begin
8           if (var12+ var12C /= 16#FFFF#) then
9                   ErrorFlag :=  16#0002#;
10          end if;
11  end thr2proc;
```

Consider the example shown in Listing 1 which is a simple, multithreaded Ada program representative of the onboard software in a safety critical application. *T1* and *T2* are two concurrent threads with *T1* having higher priority than *T2*. *var12* and *var12C* are shared variables and *ErrorFlag* is a global output variable.

The shared variables *var12* and *var12C* are updated in *T1*. These variables are inputs to *T2* which first checks their validity through a complement check and then executes some action. In safety-critical software, the integrity of input variables is protected in this manner by including its logical negation and then doing a complement check before usage. If this check fails, a bit in the error flag is set. One of the properties to be verified is specified as an assertion that the error flag is always 0.

In the existing tools for modelling and verification of concurrent software, this assertion will always be violated. This is because the assignment of the variable and its complement is not an atomic operation. Hence there exists the possibility that *T2* can interrupt *T1* in the middle of its execution such that *var12* is updated, but its complement is not. The variable will fail the complement check in *T2* and the error flag will be set. In the actual execution model, since *T2*, having lower priority, cannot interrupt the higher priority thread *T1*, such an error scenario does not occur. This false alarm is raised by the tools as they do not consider the priority among the threads. Also, some of the execution sequences generated by interleaving the two threads are not possible in the actual case.

In contrast, we model the priority also by including additional constraints in the encoding. The periodicity and priority of the threads are provided in a specification file. For instance, if *T1* runs at N times the frequency of *T2* and has a higher priority (as in a rate monotonic scheduling scheme [2]), we add the constraint that each event in *T2* can occur either before the start event or after the final event in the N iterations of *T1*. Considering that *T1* has a lower periodicity, N iterations of *T1* will be completed in the same time as a single iteration of *T2*. Essentially, any event in *T2* can occur only after an iteration of *T1* has finished execution and never in between. Thus, our new approach will correctly show that the assertion is not violated as *T2* cannot interrupt the higher priority thread *T1*. In addition, only those interleavings which adhere to the periodicity and priority relations between the threads would be included in our analysis.

3 Related work

Bounded model checking Among the current techniques for bounded model checking of concurrent, embedded software, DARTAGNAN [3] encodes tasks as Satisfiability Modulo Theory (SMT) queries and uses weak memory models. The extension of the model checker CBMC to support incremental bounded model checking (BMC) and its successful integration with the industrial embedded software verification tool BTC EmbeddedTester is explained in [4]. Reference [5] presents a method for verification of safety properties for embedded assembly programs by combining SMT-based bounded model checking and reduction of interrupt handler executions. In [6], the authors develop an SMT-based BMC for verification of embedded software written in ANSI-C. The authors introduce a tool named Memory Interval Constraint Solving (MEMICS) in [7] which is based on bounded model checking. Existing tools based on bounded model checking include CBMC, SMACK [8], SeaHorn [9], LLBMC [10], EsBMC, Frama-C and others, which are suited for verification of C programs. But these tools do not support an Ada frontend. In addition, some of these, such as SeaHorn and LLBMC, do not handle concurrent systems.

Aerospace software verification The authors describe a cloud-based verification and validation framework for a defined subset of SysML language in [11]. The approach uses the Gamma framework for UPPAAL and Theta model checkers and is applied to a case study from the aerospace field. The use of a toolchain to specify and generate a formal model of robotic software components is explained in [12]. This is then translated to FIACRE models for model checking and demonstrated using the case study of a UAV controller. In [13], the authors propose a framework and toolchain using GitHub for development of safety critical software following certification standards and apply it for the embedded software in an aeronautical system.

The work in [14] uses the methodology of continuous verification in Agile development. GitHub workflows are used to continuously integrate and verify the design of a satellite flight software on target hardware. The authors propose the first separation logic for modularly verifying the termination of programs using busy waiting for synchronization in multicore systems in [15]. A study of completeness thresholds for bounded memory safety proofs for arrays is also presented. In [16], a formal reusable

model of a real time operating system (RTOS) that can be customized for various applications is presented. The final model can be formally verified using the UPPAAL model checker and the approach is illustrated with a robotic case study. All these approaches require translation of Ada code to the model suitable for the verification tool. Our tool accepts the Ada code directly and can verify custom properties specified by the user.

Ada code verification The work in [17] explains an approach for model checking the Ravenscar subset of Ada using timed automata. There are tools like Polyspace [18] from Mathworks for verification of Ada programs. But this tool does not model the priority of the tasks directly, as in a rate monotonic scheme and can verify only specific errors like overflow, out of bound indices, unused variables, division by zero and unreachable code. Properties which are specific to a software system, as in the case of the onboard software in a rocket, cannot be verified using the existing tools for Ada.

Tools with priority The SPIN model checker [19] supports a construct for setting and using priorities for processes. In a prior work [20], we had modelled the basic scheduler and the communication protocol in the onboard embedded software of a rocket. Then, all the actual computations in the different tasks were abstracted away and only the basic data passing between different threads was included for simplicity. But the capability of SPIN is restricted to finite state machines or boolean/integer code. The SPIN model contains a finite number of processes and message channels, there is a preset bound on the number of messages that can be stored in each channel, and each variable has a finite range of possible values during execution. Therefore, the class of code that can be handled by the model checker is limited. It does not scale for software systems with a large state space and floating point operations. Since SPIN is based on explicit model checking, it suffers from state space explosion, that is, the state space grows exponentially with the addition of each variable. This was seen practically in our software also. Thus, it requires design abstractions in the model [19] to enable verification of larger systems. The data types that are supported by SPIN include bit, bool, byte, short, int and unsigned, and floating point arithmetic is excluded.

In [21], the authors describe a method of extending timed automata with priorities and implement it in the UPPAAL model checker [22]. Methods to perform subtractions on difference bound matrices (DBM), which are used to represent constraints on clock variables in model checkers are also presented. The work in [23] consists of modelling real time embedded systems with priority using prioritized timed automata (PTA) by proposing efficient algorithms for DBM subtraction and merging. These are implemented in State Graph Manipulators (SGM) model checker [24] and applied to two case studies. These two approaches use timed automata and a corresponding model checker to verify protocols and control systems. The authors present a method to identify implied scenarios in multi agent systems based on Kronecker Algebra in [25]. In addition, two new Kronecker operations are introduced and a method to include priorities in Kronecker Algebra is presented. This approach requires modelling the system in terms of state machines and does not support direct analysis of code. Our approach uses bounded model checking with priority implemented in a tool to process Ada source code and was applied to an actual aerospace software.

For the launch vehicle case, we require a tool which supports Ada and models the precedence relation among the tasks. Towards this, we develop a tool chain that can be used for static analysis of Ada programs with tasks following the rate monotonic priority. It supports varied data types, including floating point data and can be used for Ada code with multiple threads in other, similar systems in the aerospace or nuclear domain.

4 Preliminaries

In this section, we present a syntax of concurrent programs, the semantics of programs and the representation of priority of threads in a program.

We use the standard notation of set theory and logic. Let us highlight some notation to avoid any confusion.

- Let _ denote any value.
- For a set X, let $\mathcal{P}(X)$ denote the powerset of X.
- For a set A and an element e, we write $A \cup e$ to denote $A \cup \{e\}$. Similarly, $A - e$ is defined.
- For a vector of variables V, let V' be the vector of variables obtained by placing prime after each variable in V.
- For a vector V, let V_i denote the i^{th} entry in V and V_{-k} denote k^{th} entry from the end of V. We assume that all concerned formulas belong to a theory that has decision procedures for the needed operations, e.g., linear integer arithmetic.
- Let Σ be the set of formulas in the theory.
- Let $vars(\phi)$ be the set of free variables in ϕ.
- Let $\Sigma(V)$ and $\mathcal{T}(V)$ be the set of formulas and terms such that if $\phi \in \Sigma(V)$, $vars(\phi) \subseteq V$ and if $\tau \in \mathcal{T}(V)$, $vars(\tau) \subseteq V$, respectively.
- A *valuation* v of V is a map from V to values in the domain of the variables.
- For a formula ϕ, let $v \models \phi$ if $\phi(v)$ is satisfiable.
- In the algorithms, we use the shorthand $\phi :\wedge= \phi'$ to denote $\phi := \phi \wedge \phi'$. We define $:\vee=$ and $:\cup=$ similarly.
- For a variable to variable map σ, let $\sigma(\phi)$ be a formula where each variable $x \in vars(\phi)$ is replaced by $\sigma(x)$.
- Let $\oplus$ be boolean exclusive or.

Concurrent programs

We consider loop-free concurrent programs that have a fixed number of threads with a pre-defined priority and a distinct periodicity for each thread, global variables, and local variables. Since the primary case study is a safety critical software, all the loops in the code are ensured to be bounded. The loops in the code are unrolled as part of pre-processing before being fed to the tool. The targeted case study typically consists of 2-3 threads. But the tool is general enough to handle more number of threads. It is possible to apply the tool for verification of systems with single or multiple threads having equal or different priorities and periodicities.

Definition 1. *A program P is a tuple $(G, [L^1, .., L^n], pre, post, [T^1, .., T^n], [p^1, .., p^n], [r^1, .., r^n])$, where*

- *G is a vector of global variables*
- *L^t is a vector of local variables for thread t*
- *formulas $pre, post \in \Sigma(G, L^1, .., L^n)$ are respectively the precondition and post-condition of the program*
- *T^t is a thread, p^t and r^t are the period and priority respectively for thread t.*

A thread $T^t = (\Theta^t, \theta^t_s, \theta^t_f, \Delta^t)$ is a labeled directed acyclic graph, where

- *Θ^t is the set of program locations*
- *θ^t_s is the start location, θ^t_f is the final location*
- *Δ^t is a set of edges*

Edge $\delta = (\theta, stmt, \theta') \in \Delta^t$ consists of locations $\theta, \theta' \in \Theta^t$, and a statement $stmt$, given by the grammar:

$$stmt ::= \texttt{assume}(c) \mid \ell := \tau \mid \ell := g \mid g := \ell \mid \texttt{skip}$$

where $g \in G$, $\ell \in L^t$, $c \in \Sigma(L^t)$, $\tau \in \mathcal{T}(L^t)$.

- The first two kinds of statements only involve local variables: $\texttt{assume}(c)$ may block the execution of a thread and $\ell := \tau$ updates the value of a local variable.
- The next two kinds of statements can either read from or write to a global variable.
- `skip` does nothing.

$\delta_1, .., \delta_k$ is a *path* of thread t if for each $1 < j \leqslant k$, $\delta_{j-1} = (\_, \_, \theta) \in \Delta^t$ and $\delta_j = (\theta, \_, \_) \in \Delta^t$. For $\delta, \delta' \in \Delta^t$, we write $\delta \leqslant \delta'$ if there is a path from δ to δ'. On a branching location θ, we assume that there are two outgoing edges, the edges are labeled with assume statements that are negation of each other, and only one incoming edge at θ. For the presentation ease, the definition does not include compare-and-swap like instructions. However, our implementation does support them.

The statements of thread1 and thread2 in the example, in terms of the above grammar are shown in listing 2.

Listing 2 Program statements for threads 1 and 2

```
1        var12 := 16#2222#;
2        var12C := 16#DDDD#;

3        assume (var12 + var12C /= 16#FFFF#);
4        ErrorFlag := 16#0002#;
```

Memory accesses

Now, we define the semantics of memory accesses.

Definition 2. *An* event e *is either* $(t, \delta, \mathtt{R}, g, v)$ *or* $(t, \delta, \mathtt{W}, g, v)$ *, where* t *is a thread,* $\delta \in \Delta^t$ *is the source edge,* R*(read) and* W*(write) are event types, and* g *is a global variable accessed by edge* δ *with value* v*.*

- Let $e.t$, $e.\delta$, $e.type$, $e.g$, and $e.v$ be the first, second, third, fourth, and fifth components of e respectively.
- For events e and e', let $e \leqslant e' \triangleq e.t = e'.t \wedge e.\delta \leqslant e'.\delta$ and $e < e' \triangleq e \leqslant e' \wedge e \neq e'$.
- Let $\mathtt{prev}(e) \triangleq \{e' | e' < e \wedge \neg\exists e''.(e' < e'' < e)\}$.
- Let $e <_{glb} e' \triangleq e < e' \wedge e.g = e'.g$.
- Let E be a set of events. Let $E|_g \triangleq \{e \in E | e.g = g\}$.
- Let $E|_{\mathtt{W}} \triangleq \{e \in E | e.type = \mathtt{W}\}$, and $E|_{\mathtt{W}g} \triangleq E|_{\mathtt{W}} \cap E|_g$. $E|_{\mathtt{R}}$ and $E|_{\mathtt{R}g}$ are similarly defined.

A set of events generated by a program is subject to certain conditions, which are formalized as an event graph that is defined as follows.

Definition 3. *An* event graph *is a directed graph* $(E, \mathtt{rf}, \mathtt{ws}, \mathtt{fr})$*, where* E *is a set of events, and* rf *(reads-from),* ws *(write-serialization), and* fr *(from-read) are sets of edges satisfying the following conditions:*

1. *well formed: every read* reads from *exactly one write.*

$$\forall e_r \in E|_{\mathtt{R}g}.\ \exists! e_w \in E|_{\mathtt{W}g}.\ e_r.v = e_w.v \wedge (e_w, e_r) \in \mathtt{rf}$$

2. *write serialization: writes to the same variable are totally ordered by* ws*.*

$$\forall e_1, e_2, e_3 \in E|_{\mathtt{W}x}.((e_1, e_2) \in \mathtt{ws} \oplus (e_2, e_1) \in \mathtt{ws}) \wedge ((e_1, e_2) \in \mathtt{ws} \wedge (e_2, e_3) \in \mathtt{ws} \Rightarrow (e_1, e_3) \in \mathtt{ws})$$

3. *from-read condition: if a read reads from a write, then the following writes in serialization order after the write should happen after the read.*

$$\forall e_r, e_w, e'_w \in E.\ (e_w, e_r) \in \mathtt{rf} \wedge (e_w, e_{w'}) \in \mathtt{ws} \Rightarrow (e_r, e_{w'}) \in \mathtt{fr}$$

The above conditions of the event graphs are not dependent on the program and are viewed as the axioms of memory. The memory model and the program require certain ordering among events. We will define an additional condition that must be satisfied by the event graphs. Before that let us define a few relations.

Pre/post orderings: *pre* and *post* introduce write and read events, respectively. We will say that the events are from the 0^{th} thread. The events from the program threads should happen after the *pre* events and before the *post* events.

$$\mathtt{fd}(E) \triangleq \{(e_1, e_2) \in E|_{\mathtt{W}} \times E \mid e_2.t \neq e_1.t = 0\} \cup \{(e_1, e_2) \in E \times E|_{\mathtt{R}} \mid e_1.t \neq e_2.t = 0\}$$

Preserved program order: We consider a memory model based on sequential consistency (SC) and define its *preserved program order* $\texttt{ppo}_{\texttt{SC}}$.
$\texttt{ppo}_{\texttt{SC}}(P, E) \triangleq \ < \cup \texttt{fd}(E)$
SC (sequential consistency) fully enforces the program order. Now we are ready to present the acyclic condition that depends on P.

4. *Acyclic condition:* The set of events graphs $ordered_{\texttt{MM}}(P)$ contains $(E, \texttt{rf}, \texttt{ws}, \texttt{fr})$ iff $acyclic(\texttt{ppo}_{\texttt{MM}}(P, E) \cup rfe \cup \texttt{fr} \cup \texttt{ws})$, where $rfe \triangleq \{(e_w, e_r) \in \texttt{rf} | e_w.t \neq e_r.t\}$ is the set of external read-from edges. *acyclic* predicate says that the passed relation has no cycles.

Program semantics

The semantics of programs is parametrized with the SC memory model. For a thread t, a *thread state* s is a pair of $\theta \in \Theta^t$ and valuation of L^t. We write $s.\theta$ and $s.v$ for the respective components of s.

Definition 4. *An* execution *of a program P is a tuple $(\rho, \gamma, \gamma', (E, \texttt{rf}, \texttt{ws}, \texttt{fr}))$, where $\rho = (\rho^1, .., \rho^n)$ such that ρ^t is a vector of thread states of thread t, γ is the initial valuation of globals, γ' is the final valuation of globals, and $(E, \texttt{rf}, \texttt{ws}, \texttt{fr})$ is an event graph that satisfies the following.*

- *There is a path $\delta_1, .., \delta_{|\rho^t|-1}$ of thread t such that $(\rho^t_j.v, \rho^t_{j+1}.v) \models \delta_j.stmt$.*
- *E is the smallest set such that for each $g \in G$ the following holds.*

$$(0, 0, \texttt{W}, g, \gamma(g)) \in E, (0, 0, \texttt{R}, g, \gamma'(g)) \in E,$$
$$(t, \delta_j, \texttt{R}, g, \rho^t_{j+1}(\ell)) \in E \quad \text{iff} \quad \delta_j.stmt = (\ell := g),$$
$$and (t, \delta_j, \texttt{W}, g, \rho^t_j(\ell)) \in E \quad \text{iff} \quad \delta_j.stmt = (g := \ell).$$

- $\rho^1_1.\theta = \theta^1_s$, .., $\rho^n_1.\theta = \theta^n_s$, *and* $(\gamma, \rho^1_1.v, .., \rho^n_1.v) \models pre$.
- $\rho^1_{\text{-}1}.\theta = \theta^1_f$, .., *and* $\rho^n_{\text{-}1}.\theta = \theta^n_f$
- $(E, \texttt{rf}, \texttt{ws}, \texttt{fr}) \in ordered(P)$.

Due to the assumptions on programs, all executions reach to the end of all threads. An execution $(\rho, \gamma, \gamma', (E, \texttt{rf}, \texttt{ws}, \texttt{fr}))$ is *good* if $(\gamma', \rho^1_{\text{-}1}.v, .., \rho^n_{\text{-}1}.v) \models post$ otherwise it is *bad*.

5 Algorithm

We move on to our encoding of the program, which proceeds in two steps. First (Alg. 1–2), we generate constraints that encode the executions of the input program. Second (Alg. 2), we include the constraints for task priority and represent them as happens-before formulas.

5.1 SSA encoding

We construct a formula that is a static single assignment (SSA) encoding of the input program. This is similar to the CSSA form in [1]. An access to a global variable is translated into a fresh symbol that replaces it in the SSA encoding to denote the read/written value. The SSA form of thread1 and thread2 are shown in listing 3. Line 1 shows the initial state, followed by the assignments to the global variables in lines 2-3. The `assume` statement in line 4 uses two variables from selection functions `sel1` and `sel2` for choosing the writes to `thr2__var12` and `thr2__var12c`.

Listing 3 SSA form for threads 1 and 2

```
1  {thr2__errorflag_0 := 0,  thr2__var12_0 := 0,  thr2__var12c_0 := 0}
2  thr2__var12_1 := 16#2222#;
3  thr2__var12c_1 := 16#DDDD#;

4  assume (sel1 + sel2 /= 16#FFFF#);
5  thr2__errorflag_1 := 16#0002#;
```

We also need to generate memory events corresponding to the access. Since we do not have the concrete values during the SSA construction, we need a variant of event that holds the fresh symbol and the condition in which the event occurs.

Definition 5. *A* symbolic event e *is either* $(t, \delta, \mathtt{R}, g, z, cd)$, $(t, \delta, \mathtt{W}, g, z, cd)$ *or* $(t, \delta, \mathtt{D}, cd)$ *where* t *is a thread,* $\delta \in \Delta^t$ *is the source edge,* `R`*(read),* `W`*(write) and* `D`*(dummy) are event types, the event occurs if the formula* cd *holds true, and* g *is a global variable accessed by edge* δ *with symbolic value* z*.*

For a symbolic event e, let $e.z$ and $e.cd$ be the fifth and sixth component of e, respectively. The rest of the notation for the symbolic events is same as for the events. The symbolic events created for thread1 and thread2 in our example are shown in listing 4.

Listing 4 Events for threads 1 and 2

```
S1                      : Start event for thr1
W@thr2__var12_1         : Write event
W@thr2__var12c_1        : Write event
E1                      : End event for thr1

S2                      : Start event for thr2
R@sel1                  : Read event
R@sel2                  : Read event
E2                      : End event for thr2
```

Algorithm 1 $\textsc{Ssa}(P=(G,[L^1,..,L^n],pre,post,[T^1,..,T^n]))$

Ensure: SSA encoding ϕ_{ssa}, post encoding ϕ_{post}, symbolic events E

1: $\phi_{ssa} := \mathbf{tt}$; $\sigma : \Theta \cup \Delta \to Var \to \Sigma := \lambda x.\bot$; $Cs : \Theta \cup \Delta \to \Sigma := \lambda x.\mathbf{ff}$
2: **for** $\ell \in G \cup L^1 \cup .. \cup L^n$ **do** $\sigma_s(\ell) := fresh()$;
3: $E := \{(0,0,\mathtt{W},g,\sigma'(g),\mathbf{tt}) | g \in G\}$; $\phi_{ssa} := \sigma_s(pre)$; $\phi_{pre} := \sigma_s(pre)$;
4: **for** $t \in 1..n$ **do**
5: $\quad\sigma(\theta_s^t) := \sigma_s; Cs(\theta_s^t) := \phi_{pre}$
6: $\quad$**for** $\delta = (\theta, stmt, \theta') \in \Delta^t$ **do** ▷ topological order
7: $\quad\quad$**if** $\sigma(\theta) = \bot$ **then**
8: $\quad\quad\quad$**for** $\ell \in L^t$ **do**
9: $\quad\quad\quad\quad\sigma(\theta)(\ell) := fresh()$; $\phi_{ssa} :\wedge= \bigwedge_{\delta'=(\_,stmt,\theta)\in\Delta^t}(Cs(\delta') \Rightarrow \sigma(\theta)(\ell) = \sigma(\delta')(\ell))$;
10: $\quad\quad\quad z := fresh()$; $\sigma' := \sigma(\theta)$; $cd := Cs(\theta)$
11: $\quad\quad\quad$**match** *stmt* **with**
12: $\quad\quad\quad|\mathtt{assume}(c) \to cd :\wedge= \sigma'(c)$; $E :\cup= (t,i,\mathtt{D},cd)$;
13: $\quad\quad\quad|\ell := \tau \to \phi_{ssa} :\wedge= \sigma'(z = \tau); \sigma'(\ell) := z$; $E :\cup= (t,i,\mathtt{D},cd)$;
14: $\quad\quad\quad|\ell := g \to \sigma'(\ell) := z; e := (t,i,\mathtt{R},g,z,cd); E :\cup= e$;
15: $\quad\quad\quad|g := \ell \to \phi_{ssa} :\wedge= \sigma'(z = \ell); e := (t,i,\mathtt{W},g,z,cd);$
16: $\quad\quad\quad\quad E :\cup= e$;
17: $\quad\quad\quad|\mathtt{skip} \to E :\cup= (t,\delta,\mathtt{D},cd)$
18: $\quad\quad\quad\sigma(\delta) := \sigma'$; $Cs(\delta) := cd$; $Cs(\theta') :\vee= cd$
19: $\quad$**for** $\ell \in L^t$ **do** $\sigma_f(\ell) := \sigma(\theta_f)(\ell)$
20: **for** $g \in G$ **do** $\sigma_f(g) := fresh(); E :\cup= (0,0,\mathtt{R},g,\sigma_f(g))$
21: $\phi_{post} := (Cs(\theta_f^1) \wedge .. \wedge Cs(\theta_f^n) \Rightarrow \sigma_f(post))$;
22: **return** $(\phi_{ssa}, \phi_{post}, E)$

5.1.1 Algorithm for SSA encoding

In Algorithm 1, we present the function Ssa that takes a program as input and returns an SSA encoding ϕ_{ssa} of threads, an SSA formula ϕ_{post} for the post condition and a set of symbolic events E.

Let us first discuss ϕ_{ssa} computation. Besides the returned objects, the algorithm needs a few local objects, namely σ, which is a map from locations or edges to substitutions and Cs, which is a map from locations or edges to their condition of occurring in an execution. σ and Cs are declared in line 1. For a given location θ and edge δ, $\sigma(\theta)$ and $\sigma(\delta)$ are maps from the program variables to fresh SSA variables. In line 2, we initialize σ_s that is the substitution for the initial values of the program variables, where $fresh()$ returns a fresh SSA variable. In line 3, we assign the initialization events of globals to E, and the SSA encoding of pre to ϕ_{ssa} and ϕ_{pre}.

The loop in line 4 iterates over each thread t. In line 5, we initialize the substitution and condition map for the initial thread location θ_s^t. The loop in line 6 iterates over each edge $\delta = (\theta, stmt, \theta')$ in the topological order. In line 9, if $\sigma(\theta)$ is uninitialized, we map the locals in $\sigma(\theta)$ to fresh SSA variables, and equate the SSA variables with the values from the incoming edges and guard the equations with the edge conditions. In line 10, we get a fresh variable z for the value that may be produced by $stmt$. We copy $\sigma(\theta)$ to σ' and $Cs(\theta)$ to cd. In lines 12–17, we process $stmt$ as follows.

Algorithm 2 EVENTCONS(E)

Ensure: SSA encoding ϕ, symbolic events E

1: $\phi_{\texttt{wf}} := \phi_{\texttt{rf}} := \phi_{\texttt{fr}} := \phi_{\texttt{ws}} := \phi_{\texttt{ppo}} := \textbf{tt}$
2: **for** $g \in G$ **do**
3: **for** $r \in E|_{\texttt{R}g}$ **do**
4: $E' := E|_{\texttt{W}g} - \{w | r < w \in E|_{\texttt{W}g}\}$; $\phi_{\texttt{wf}} :\wedge= (r.cd \Rightarrow \bigvee_{w \in E'} b_{w,r}) \wedge \bigwedge_{w \in E'} (b_{w,r} \Rightarrow w.cd \wedge z_w{=}z_r)$
5: **for** $r \in E|_{\texttt{R}g}, w \in E|_{\texttt{W}g}$ such that $w.t \neq r.t$ **do** $\phi_{\texttt{rf}} :\wedge= (b_{w,r} \Rightarrow \text{HB}(w, r))$ ▷ read from
6: **for** $r \in E|_{\texttt{R}g}$, $w, w' \in E|_{\texttt{W}g}$ **do** ▷ from read
7: $rhs := (w' < r)\ ?\ \textbf{ff} : \text{HB}(r, w')$
8: $\phi_{\texttt{fr}} :\wedge= (b_{w,r} \wedge \text{HB}(w, w') \wedge w'.cd \Rightarrow rhs)$
9: $\phi_{\texttt{ws}} := \bigwedge_{w,w' \in E|_{\texttt{W}g}} (\text{HB}(w, w') \vee \text{HB}(w', w))$ ▷ write serialization
10: **for** $e, e' \in E$ if $e < e'$ **do** $\phi_{\texttt{ppo}} :\wedge= \text{HB}(e, e')$
11: **for** $e \in E|_{\texttt{W}}$, $e' \in E$ **if** $e.t = 0$ **do** $\phi_{\texttt{ppo}} :\wedge= \text{HB}(e, e')$
12: **for** $e \in E$, $e' \in E|_{\texttt{R}}$ **if** $e'.t = 0$ **do** $\phi_{\texttt{ppo}} :\wedge= \text{HB}(e, e')$
13: **for** $e \in E$, $e' \in E'$ such that $pr(e'.t) > pr(e.t)$
14: **if** $e' = start$ **do** $\phi_{\texttt{ppo}} :\wedge= \text{HB}(e, e')$
15: **if** $e' = finish$ $\phi_{\texttt{ppo}} :\wedge= \text{HB}(e', e)$
▷ priority
16: $\phi_E := \phi_{\texttt{wf}} \wedge \phi_{\texttt{rf}} \wedge \phi_{\texttt{fr}} \wedge \phi_{\texttt{ws}} \wedge \phi_{\texttt{ppo}}$
17: **return** ϕ_E

- `assume`(c): We append $\sigma'(c)$ to cd and create a dummy event.
- $l := \tau$: We append ϕ_{ssa} with SSA encoding of $stmt$ and assign z to $\sigma'(l)$.
- $l := g$: We assign z to $\sigma'(l)$ and add a new 'read' event.
- $g := l$: We append ϕ_{ssa} to record the write to g and create a new write event.
- `skip`: We create a dummy event.

At the end of the iteration, we assign σ' to $\sigma(\delta)$ and, similarly, cd to $Cs(\delta)$. We also disjunctively append to the condition of θ'. After processing all the edges in thread t, we record the last substitution map of the thread in σ_f. After the big loop, we get fresh variables for the 'read' events due to the post condition and produce ϕ_{post}, which states that the post condition is evaluated only if all the threads are finished.

5.2 Constraint generation

In Algorithm 2, we take the data structures built in the last algorithm as input, and produce constraints that encode the event graphs that satisfy the four conditions.

The conditions are about events orderings, which we encode using difference constraints. For event e, we introduce integer clock variables c_e. If (e, e') is an ordering relation then $c_e < c_{e'}$ encodes the ordering of the events. Let $\text{HB}(e, e') \triangleq e.cd \wedge e'.cd \Rightarrow c_e < c_{e'}$, which encodes that if e and e' occur, e happens before e'. For every pair of 'write' w and 'read' r, we use a bit $b_{w,r}$ to encode that r reads from w.

5.2.1 Event graph

In the algorithm, we construct the formulas $\phi_{\texttt{wf}}$, $\phi_{\texttt{rf}}$, $\phi_{\texttt{fr}}$, $\phi_{\texttt{ws}}$ and $\phi_{\texttt{ppo}}$. Their conjunction encodes the four conditions. The loop in line 2 iterates over each global variable g. It has three loops inside. The first loop in line 3 iterates over each 'read' $r \in E|_{\texttt{R}g}$, generates the constraints that encode that if r occurs, it reads from exactly one 'write', and appends the constraints to $\phi_{\texttt{wf}}$. Due to coherence restrictions a 'read' cannot read from a 'write' that occurs later in same thread. Therefore, E' excludes those writes in line 4.

The second loop in line 5 iterates over every 'read' and 'write'. If a 'read' reads from a 'write' from another thread then it must happen afterwards due to the acyclicity condition. The third loop in line 6 constructs $\phi_{\texttt{fr}}$ that encodes the acyclicity of $\texttt{fr}$. We also need to ensure that $\texttt{fr}$ and $\texttt{fr}\circ\texttt{rf}$ do not form cycles with $<_{glb}$ to enforce coherence. The loop iterates over a 'read' r and a pair of 'writes' w and w' and constructs the acyclicity constraint for $\texttt{fr}$ in line 8. Due to the coherence condition, we need to choose rhs accordingly. In line 7, if $w' < r$, we disallow $(r, w') \in \texttt{fr}$ by choosing rhs to be $\mathbf{ff}$, otherwise there will be a cycle with $<_{glb}$. In line 8, we make constraints that encode that if a 'read' r' occurs before r in a thread then r' cannot read from w', otherwise $\texttt{fr}\circ\texttt{rf}$ will form a cycle with $<_{glb}$. In line 9, we construct $\phi_{\texttt{wf}}$ that encodes that all pair of 'writes' to g must be ordered.

5.2.2 Preserved Program Order

In lines 10–12, we construct $\phi_{\texttt{ppo}}$ that encodes the preserved program ordering. The loop in line 10 adds constraints on the ordering of the events from a thread. In lines 11–12, we generate ordering constraints due to the pre/post condition. Finally, we include the additional constraints for priority among the threads, as per the rate monotonic principle. The higher the frequency of execution of a thread, the higher the priority, in this scheme. Thus we add the constraint that for each thread, the events can occur either before the *start* event or after the $finish$ event in every iteration of all threads of higher priority. This is shown in lines 14–15. Here $pr(e.t)$ denotes the priority of the thread t with event e. This further reduces the number of combinations of program interleavings possible in a concurrent system.

The constraints for program order and priority of thread1 and thread2 for our example are shown in listing 5. The sequential consistency is denoted by constraints under *Program order*. For priority, the constraints indicate that the read events in lower priority thread2 occur either after the end event in the first iteration or before the start event in the second iteration of the higher priority thread1. Thread2 events can also happen after the end event in the second iteration of thread1. Here the start and end events in the two iterations are denoted by `S1,S1a` and `E1,E1a` respectively, assuming thread2 executes at a periodicity twice that of thread1.

Listing 5 Constraints

```
Program order:
        HB(S1, W@thr2__var12_1)
        HB(W@thr2__var12_1, W@thr2__var12c_1)
        HB(W@thr2__var12_1c, E1)
```

```
        HB(S2, R@sel1)
        HB(R@sel1, R@sel2)
        HB(R@sel2, E2)

Priority:
        (HB(E1, R@sel1) and HB(E1, R@sel2)) or
        (HB(R@sel1, S1a) and HB(R@sel2, S1a)) or
        (HB(E1a, R@sel1) and HB(E1a, R@sel2)) or
```

6 Implementation

6.1 Frontend

A front-end/ pre-processor is required to obtain necessary details and convert the Ada program to a standard intermediate representation (IR). Dragonegg [26], a gcc plug-in which can generate LLVM IR, was chosen for this purpose. The LLVM (Low Level Virtual Machine) Project [27] is a collection of modular and reusable compiler and toolchain technologies. The LLVM core libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs. These libraries are built around a well specified code representation known as the LLVM intermediate representation (LLVM IR).

Dragonegg is a gcc plug-in that replaces GCC's optimizers and code generators with those from the LLVM project. It works with gcc-4.5 or newer, can target the x86-32/x86-64 and ARM processor families, and has been successfully used on various platforms. It fully supports Ada, C, C++ and Fortran. It has partial support for Go, Java, Obj-C and Obj-C++. It accepts Ada input programs and generates LLVM IR. The plug-in acts by substituting the middle- and backend of GCC with the LLVM ones and performs all the compilation steps automatically. The steps in the translation from Ada code with dragonegg are depicted in Figure 1.

The switches *-fplugin-arg-dragonegg-emit-ir -S* are applied to stop the compilation pipeline at the LLVM IR generation phase. Since dragonegg was not maintained, several changes were implemented to customize it for our application. It now works with LLVM-6.0 and gcc-8.2. The LLVM IR generated by dragonegg for the Ada procedures in Listing 1 is shown in Listing 6.

Listing 6 LLVM IR for Ada example

```
1 define void @thr1proc() unnamed_addr #0 align 2 {
2 entry:
3   store i16 8738, i16* @thr2__var12, align 2
4   store i16 -8739, i16* @thr2__var12c, align 2
5   ret void
6 }

7 define void @thr2proc() unnamed_addr #0 align 2 {
8 entry:
```

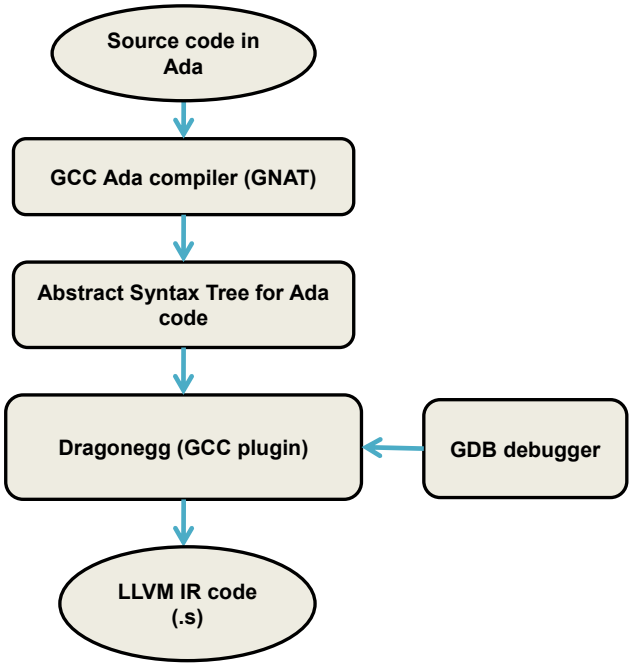

Fig. 1 Translation from Ada to LLVM IR – Dragonegg

```
  %0 = load i16, i16* @thr2__var12, align 2
  %1 = load i16, i16* @thr2__var12c, align 2
  %2 = add i16 %0, %1
  %3 = icmp ne i16 %2, -1
  br i1 %3, label %"3", label %return

"3":
  store i16 2, i16* @thr2__errorflag, align 2
  br label %return

return:
  ret void
}
```

6.2 Ada static analyser

We implemented the bounded model checking with priority encoding and developed it as a tool named LLVMBMC (LLVM Bounded Model Checker, source code available at *https://github.com/ranjani141/AdaStaticAnalyser*). It takes LLVM IR files as input along with a specification file. The overall approach for our verification with bounded

model checking, starting with the LLVM IR file is shown in Figure 2 and the details are elaborated in the following paragraphs.

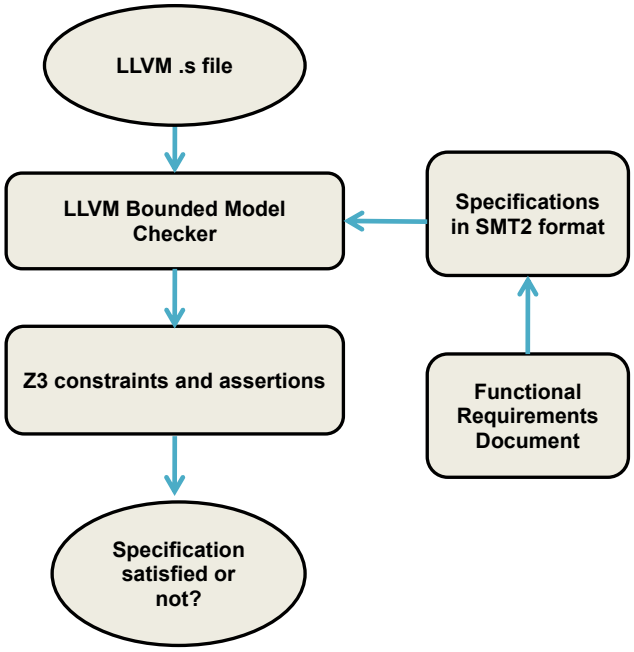

Fig. 2 Steps in LLVM BMC

Parsing of Specification file A specification file is written for the system, with details of each thread, such as its name, entry function, priority, periodicity and the global variables. The properties for each module are derived from the software requirements and written in terms of the inputs and outputs in the specification file. The standard SMT2 format [28] is adopted for writing the properties in this spec file. In addition to the LLVM IR file, this spec file is accepted as an input to the tool. In Listing 7, the format of the spec file is illustrated. There are two threads, with periodicity of 1ms and 2ms and priority 1 and 2, respectively, along with a post condition.

Listing 7 Sample Specification File

```
(declare-var @peterson__var1 i16);
(declare-var @peterson__var2 i16);

(declare-thread one peterson__thread0)
(invoke-parameters one repeated 1 priority 1)
```

```
(end-thread one)
(declare-thread two peterson__thread1)
(invoke-parameters two repeated 2 priority 2)
(end-thread two)

(pre-condition all (assert (= @peterson__var1 0)))
(pre-condition all (assert (= @peterson__var2 0)))
;
(post-condition all
(assert (bvugt @peterson__var1 @peterson__var2) ))
;
```

The software requirement in the document, the property in terms of the program variables and translated to the SMT2 format are shown below with an example from the clock synchronization module in the launch vehicle software:

Requirement1 (in Requirements document):

If the computed clock skew is outside the lower and upper limits, no correction is applied to the clock

Property1 (in terms of input and output variables):

if ((Skew < LowerLimit) or (Skew > UpperLimit)), then (CorrectFlag = 0)

Specification1 (in SMT2 format):

```
(post-condition all (assert(=>(or (bvult @sync__skew
@sync__lowerlimit) (bvugt @sync__skew @sync__upperlimit))(=
@sync__correctflag #x0000))))
```

The specification file is parsed to collect the name and type of global variables, along with the details of threads, namely its name, entry function, periodicity and priority. For each thread, the number of iterations in a cycle are computed. Also, the pre- and post- conditions are translated to Z3 expressions.

Pre-processing As part of pre-processing, the input module is fully inlined and the initial memory state is defined. Details of arrays, loops and concurrent global variables are also collected. For each thread, the start event is declared.

Translation The pre- and post-conditions are translated to the corresponding Z3 expressions, with the variable names replaced with their names in the initial and final states respectively. Each instruction is converted to Z3 constraints. Read and write events are generated for loads and stores to concurrent global variables.

Encoding in SSA Multiple copies of constraints and events are then created, based on the number of iterations for the threads. Symbolic predictive analysis [1] is implemented for considering the different possible program executions. Every new write and read to a variable is distinguished using a distinct name by encoding in Static Single Assignment(SSA) form. For each read from a global variable, a selection function is used to identify the most recent write.

Encoding of constraints for program order and priority We then included the constraints on order of events. The additional constraints for priority as per the rate monotonic scheduling scheme are also added.

Verification The tool can verify properties including correct sequence of function calls, proper event timings, conditional execution and standard errors like overflow.

The given pre-conditions are assumed to be true by the verifier and it checks whether the assertions or post conditions are always met. We also have the provision to obtain the error trace when a specification is violated. Since the initial states are known, along with the various events and branching conditions, the execution sequence for the program is generated and written to a file, when specified by the user. For instance, the error trace generated when the specification is violated for the example in Listing 1 is shown in Listing 8. The different events, along with their respective time stamps are shown. As the two reads (lines 2-3) happen before the second write (line 4) is completed, the specification is violated.

Listing 8 Error trace

```
1          (define-fun W@thr2__var12@_u2 () 3)
2          (define-fun R@thr2__var12@_u6 () 4)
3          (define-fun R@thr2__var12c@_u7 () 5)
4          (define-fun W@thr2__var12c@_u3 () 6)
```

7 Testing

The onboard software in Indian launch vehicles is developed in a safe subset of Ada83 [29]. An associated library file contains the hardware specific constructs, user-defined data structures, data type conversions/type casting functions and mathematical/trigonometric functions in the code. The software consists of two parts, the minor tasks executed at a periodicity of 20 ms and major tasks, executed at a periodicity of 500 ms. For some missions, a micro cycle periodicity of 10 ms is also demanded. The tasks are scheduled such that those with higher frequency of execution have higher priority than those of lower frequency. Thus, the minor tasks will always have higher priority than the major tasks. This rate monotonic scheduling scheme, considered optimal for concurrent, periodic embedded systems, is to be accurately represented. The tests were carried out on a PC with Intel i5 processor and 16 GB RAM running Ubuntu operating system.

The execution model where the minor and major threads are running should be modelled correctly for verification. All feasible execution traces are considered using sequences of events. The constraints for sequential consistency and program order are encoded. Additionally, the constraints for the rate monotonic scheme are included as per the priority of the threads. This is a novel technique developed by the authors for modelling a concurrent system with pre-defined priorities for tasks. For each thread, every event in the thread can occur either before the start event or after the end event of all other threads with higher priority. For instance, in the case of the launch vehicle software with two threads, when the minor thread is executing, no events in the major thread can occur. The major thread can execute only between two successive iterations of the minor thread. That is, each event in the major thread can occur either before the start event of each iteration of the minor thread or after the final event in each iteration of the minor thread. We modelled this priority for a general system with any number of threads and priorities assigned as per the rate monotonic scheme.

Table 1 Verification results for concurrency litmus tests

Program name	Threads	Memory-CBMC (MB)	Memory-LLVMBMC (MB)	Time-CBMC (s)	Time-LLVMBMC (s)
sb	2	36.784	126.712	0.20	0.09
sbr	2	36.792	126.888	0.20	0.09
lb	2	36.684	126.660	0.19	0.13
corr0	2	36.808	123.90	0.19	0.08
corr1	2	35.536	126.664	0.18	0.08
corw	2	36.552	126.708	0.19	0.09
cowr	2	36.464	124.012	0.19	0.08
iriw	4	36.260	126.676	0.19	0.12

7.1 Simulation and Verification Results

We tested the modelling using the different sets of benchmarks. Standard litmus tests for concurrency [30] constituted the first set. Traditionally, a litmus test is a small parallel program designed to exercise the memory model of a parallel, shared-memory computer. The properties to be checked are also part of the litmus test set. We rewrote the litmus programs in Ada and processed it using our frontend to generate the LLVM IR. We then fed it to LLVMBMC along with the specifications file in SMT2 format.

In the second set, standard mutual exclusion protocols like Dekker [31], Dijkstra [32], Peterson [33] and other examples were included. Here too, we programmed the algorithms in the Ada language and modified the code such that the specifications would be satisfied only if priority is respected. The third set consisted of several examples with different number of threads and priorities from the launch vehicle software. The properties were selected so as to check the correctness of the implementation with priority.

7.1.1 Concurrency litmus tests

Table **??** shows the results of our analysis with the concurrency litmus test programs. The name of the program, number of threads and the time and memory for verification are shown in the columns. The same programs, coded in C, were tested using CBMC tool also, as CBMC is representative of other tools for verification of C/C++ programs, like SMACK, EsBMC, Frama-C and LLBMC.

7.1.2 Tests with mutual exclusion algorithms

The results of our experiments with standard mutual exclusion algorithms are shown in Table 2. The programs were modified such that the values of the concurrent variables depend on the priority. As listed in the table, the properties in all the tests were satisfied only when the precedence of the threads was also considered, as in LLVMBMC. Here, the properties were violated when CBMC was used for verifying the corresponding C programs, since it does not consider the priority of the threads.

Table 2 Test Results with mutual exclusion algorithms

Program name	Threads	Result-CBMC	Result-LLVMBMC
Burns	2	Violated	Satisfied
Dekker	2	Violated	Satisfied
Dijkstra	2	Violated	Satisfied
Peterson	2	Violated	Satisfied
Szymanski	2	Violated	Satisfied

7.1.3 Aerospace software case study

Table 3 shows the results of our analysis with the flight software of a launch vehicle. The name of the program, number of threads and the result of verification of the property are shown in the columns, along with the lines of code in the program and the time for verification. Since the test cases form part of classified onboard software of a launch vehicle, we present only a portion of the results here. The test cases consist of modules from the navigation and guidance software for the rocket. Some modules from fuel tank venting algorithm, clock synchronization, input data acquisition and output data posting were also part of the case studies. A few properties from navigation and guidance modules are listed below to illustrate the functional requirements:

- *Property1 (Navigation) - The norm of the quaternions is always equal to 1.*
- *Property2 (Navigation) - When the health of both the navigation systems is not OK, the software should set the failure flag.*
- *Property1 (Guidance) - The software should set the second stage shut down flag only after receiving the third stage ignition flag from the sequencer.*
- *Property2 (Guidance) - Closed loop guidance starts execution only after reception of the initiation flag.*

In each case, we modified the specification file to remove the priority, that is, to assign equal priority to all threads and re-executed the test cases. It was seen that the properties are violated in these cases. For instance, Test6 is part of the guidance software with minor and major threads. Data is passed from minor to major thread assuming that the minor thread, having higher priority, would not be interrupted. So, no explicit atomicity is implemented in the code. If the priority of the threads is not considered, the minor thread could be interrupted by the major thread and the expected atomicity of the operations would be violated. Thus, the inputs from the minor thread to the major thread would be partially updated, leading to erroneous computations. This is because the rate monotonic implementation assumes that the scheduling of the tasks is as per their priority based on the frequency of execution. Similarly, in Test7, which is the navigation software in the rocket, the major thread uses certain variables from the minor thread for computations and monitoring. Here too, the implicit assumption is that the higher priority minor thread has already completed execution. The approach is general and can be applied for larger software systems with more number of threads also.

Smaller test cases, based on modules in the launch vehicle software were written in C and tested with CBMC. Here, the property was violated in all cases as the tool does

Table 3 Verification results for launch vehicle software

Program name	Threads	Property	Lines of Code	Verification time
Test1	2	Satisfied	1800	120s
Test2	3	Satisfied	1050	600s
Test3	2	Satisfied	320	45s
Test4	2	Satisfied	2200	600s
Test5	2	Satisfied	300	28s
Test6	2	Satisfied	15000	5900s
Test7	2	Satisfied	10000	3000s

Table 4 Verification results for launch vehicle software - CBMC and LLVMBMC

Program name	Threads	Result-CBMC	Result-LLVMBMC
Test1	2	Violated	Satisfied
Test2	3	Violated	Satisfied
Test3	2	Violated	Satisfied
Test4	2	Violated	Satisfied
Test5	2	Violated	Satisfied

not take into account the priority of the threads. The corresponding Ada programs were verified using LLVMBMC also. A comparison of the verification results, with CBMC and LLVMBMC, is shown in Table 4.

8 Conclusion

In this paper, we propose a novel static analysis technique for the verification of concurrent programs in Ada language in embedded systems with pre-defined priorities and periodicities for the tasks. Each statement in the program execution is encoded in a static single assignment form and the additional constraints for priority are also included. Any violation of the constraints specifying the properties to be verified are indicated by the analysis. The developed tool was customised and applied to a real-world case study, the onboard software in Ada language in a safety critical aerospace system.

References

[1] Wang, C., Kundu, S., Limaye, R., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. Formal aspects of computing **23**, 781–805 (2011)

[2] Warren, C.: Rate monotonic scheduling. IEEE Micro **11**(3), 34–38 (1991)

[3] Ponce-de-León, H., Haas, T., Meyer, R.: Dartagnan: SMT-based violation witness validation (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022,

Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II, pp. 418–423 (2022). Springer

[4] Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Incremental bounded model checking for embedded software. Formal Aspects of Computing **29**, 911–931 (2017)

[5] Yamane, S., Kobashi, J., Uemura, K.: Verification Method of Safety Properties of Embedded Assembly Program by Combining SMT-Based Bounded Model Checking and Reduction of Interrupt Handler Executions. Electronics **9**(7), 1060 (2020)

[6] Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE Transactions on Software Engineering **38**(4), 957–974 (2011)

[7] Traub, J.F.J.: Formal Verification of Concurrent Embedded Software. Kiel Computer Science Series, vol. 2016/1. Department of Computer Science, CAU Kiel, ??? (2016). Dissertation, Faculty of Engineering, Kiel University.

[8] Carter, M., He, S., Whitaker, J., Rakamarić, Z., Emmi, M.: SMACK software verification toolchain. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 589–592 (2016)

[9] Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: A framework for verifying C programs (competition contribution). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 447–450 (2015). Springer

[10] Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: Verified Software: Theories, Tools, Experiments: 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings 4, pp. 146–161 (2012). Springer

[11] Horváth, B., Molnár, V., Graics, B., Hajdu, Á., Ráth, I., Horváth, Á., Karban, R., Trancho, G., Micskei, Z.: Pragmatic verification and validation of industrial executable SysML models. Systems Engineering **26**(6), 693–714 (2023)

[12] Dal Zilio, S., Hladik, P.-E., Ingrand, F., Mallet, A.: A formal toolchain for offline and run-time verification of robotic systems. Robotics and Autonomous Systems **159**, 104301 (2023)

[13] Baron, C., Louis, V.: Framework and tooling proposals for Agile certification of safety-critical embedded software in avionic systems. Computers in Industry **148**, 103887 (2023)

[14] McColl, M., McColl, C., Pereira, A., Souza, P., Tuxworth, G., Hexel, R.: Continuous Formal Verification for Aerospace Applications. In: 2024 IEEE Aerospace Conference, pp. 1–19 (2024). https://doi.org/10.1109/AERO58975.2024.10521386

[15] Reinhard, T.: Semi-Automated Modular Formal Verification of Critical Software (2024)

[16] Adelt, J., Gebker, J., Herber, P.: Reusable formal models for concurrency and communication in custom real-time operating systems. International Journal on Software Tools for Technology Transfer **26**(2), 229–245 (2024)

[17] Guaspari, D., Naydich, D.: Analysis of real-time code by model checking. In: 19th DASC. 19th Digital Avionics Systems Conference. Proceedings (Cat. No. 00CH37126), vol. 1, pp. 1–51 (2000). IEEE

[18] Mathworks: Comprehensive Static Analysis Using Polyspace Products-White Paper. Mathworks, ??? (2013)

[19] Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual vol. 1003. Addison-Wesley Reading, ??? (2004)

[20] Krishnan, R., Lalithambika, V.: Modeling and Validating Launch Vehicle Onboard Software Using the SPIN Model Checker. Journal of Aerospace Information Systems **17**(12), 695–699 (2020)

[21] David, A., Håkansson, J., Larsen, K.G., Pettersson, P.: Model checking timed automata with priorities using DBM subtraction. In: Formal Modeling and Analysis of Timed Systems: 4th International Conference, FORMATS 2006, Paris, France, September 25-27, 2006. Proceedings 4, pp. 128–142 (2006). Springer

[22] Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International journal on software tools for technology transfer **1**, 134–152 (1997)

[23] Lin, S.-W., Hsiung, P.-A.: Model checking prioritized timed systems. IEEE Transactions on Computers **61**(6), 843–856 (2011)

[24] Hsiung, P.-A., Wang, F.: A state graph manipulator tool for real-time system specification and verification. In: Proceedings Fifth International Conference on Real-Time Computing Systems and Applications (Cat. No. 98EX236), pp. 181–188 (1998). IEEE

[25] Denzler, P.: Approaching Emergent Patterns with Kronecker Algebra in Industrial Agents. PhD thesis, Technische Universität Wien (2023)

[26] LLVM: Dragonegg-Using LLVM as a GCC Backend (2013)

[27] Lattner, C.: Introduction to the LLVM compiler infrastructure. In: Itanium

Conference and Expo (2006)

[28] Cok, D.R., et al.: The SMT-LIBv2 language and tools: A tutorial. Language c, 2010–2011 (2011)

[29] Smith, D.A.: ANSI Standard Ada: quick-reference sheet. ACM SIGAda Ada Letters **4**(1), 61–66 (1984)

[30] Maranget, L., Sarkar, S., Sewell, P.: A tutorial introduction to the ARM and POWER relaxed memory models. Draft available from http://www. cl. cam. ac. uk/˜ pes20/ppc-supplemental/test7. pdf (2012)

[31] Dijkstra, E.W.: Over de sequentialiteit van procesbeschrijvingen. (1962)

[32] Dijkstra, E.W.: Solution of a problem in concurrent programming control. In: Pioneers and Their Contributions to Software Engineering: Sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions, pp. 289–294 (2001). Springer

[33] Peterson, G.L.: Myths about the mutual exclusion problem. Information Processing Letters **12**, 115–116 (1981)