Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 331 (2017) 3–39

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Static Analysis of Embedded Real-Time Concurrent Software with Dynamic Priorities

Antoine Miné[1,2]

Sorbonne Universités, UPMC Univ. Paris 06, CNRS, LIP6, Paris, France

Abstract

In previous work, we developed a sound static analysis by abstract interpretation to check the absence of run-time errors in concurrent programs, focusing on embedded C programs composed of a fixed set of threads in a shared memory. The method is thread-modular: it considers each thread independently, analyzing them with respect to an abstraction of the effect of the other threads, so-called interference, which are also inferred automatically as part of analyzing the threads. The analysis thus proceeds in a series of rounds that reanalyze all threads, gathering an increasing set of interference, until stabilization. We proved that this method is sound and covers all possible thread interleavings. This analysis was integrated into the Astrée industrial-scale static analyzer, deployed in avionics and automotive industries.

In this article, we consider the more specific case of programs running under a priority-based real-time scheduler, as is often the case in embedded systems. In such programs, higher priority threads cannot be preempted by lower priority ones (except when waiting explicitly for some resource). The programmer exploits this property to reduce the reliance on locks when protecting critical sections. We show how our analysis can be refined through partitioning in order to take into account the real-time hypothesis, remove spurious interleavings, and gain precision on programs that rely on priorities. Our analysis supports in particular dynamic priorities: we handle explicit modifications of the priorities by the program, as well as implicit ones through the priority ceiling protocol.

We illustrate our construction formally on an idealized language. Following previous work, we first provide a concrete semantics in thread-modular denotational form that is complete for safety properties, and then show how to apply classic abstractions to obtain an effective static analyzer, able to detect all run-time errors, data-races, as well as deadlocks. Finally, we briefly discuss our implementation inside the Astrée analyzer and on-going experimentation, with results limited for now to small programs.

Keywords: static analysis, abstract interpretation, verification, safety, concurrency, run-time errors, data-races, deadlocks, real-time scheduling, priority ceiling protocol

1 Introduction

Program verification is an important part of software development, and has a significant cost in industry, hence the need to research more cost-effective methods. It is particularly important to ensure that critical embedded software that control planes or cars are free from errors. Testing, the main established method, is more

[1] The work described in this article is supported in part by the project ANR-11-INSE-014 (AstréeA) from the French *Agence nationale de la recherche* and in part by the ITEA 3 project 14014 (ASSUME).

[2] Email: antoine.mine@lip6.fr

http://dx.doi.org/10.1016/j.entcs.2017.02.002

1571-0661/© 2017 The Author(s). Published by Elsevier B.V.

and more often supplemented with formal methods. Unlike testing, they can provide strong mathematical guarantees about the behaviors of programs. Semantic-based static analysis by abstract interpretation [12] is particularly attractive: it infers statically properties of the dynamical behaviors of a program, reasons directly on the original, unmodified source (or binary) code, it is fully automated, and it is sound. Soundness is a key property: it states that no behavior of the program is omitted by the analysis, so that any property derived by the analysis is true on all possible executions. In the avionics field, for instance, certification authorities require static analyses to be sound in order to be considered part of the certification process [2]. To scale up, and to produce effective results on what is in general an undecidable problem, abstract interpretation performs approximations which, to maintain soundness, must be conservative and over-approximate the set of possible behaviors. This may result in false alarms: situations where the approximation considers spurious erroneous executions and fails to establish the correctness of a correct program. Nevertheless, through a manual process of specialization of abstractions targeting specific classes of programs and properties to prove, we can hope to achieve a precise and efficient enough analysis, at least on the target programs. For instance, in previous work, we participated to the design of Astrée [6], one such analyzer aiming to prove the absence of run-time errors (such as integer or float overflows, invalid pointer accesses, etc.) on embedded critical sequential C code, and specialized it for control-command avionics software. Specialization took the form of developing new, sophisticated abstract domains, mainly for numeric properties, and heuristic to choose wisely at each program point the desired cost versus expressiveness among to abstractions available. Astrée is being used in an industrial context, initially at Airbus [14] and, following its commercialization by AbsInt [20], in other embedded critical industries, such as the automotive industry.

We consider here the static analysis of *concurrent* embedded software. On the one hand, more and more software are concurrent, either to exploit the parallel execution offered by current multi-core processors, or simply for ease of programming. This trend also affects critical software. For instance, Integrated Modular Avionics [38] aims at replacing sets of processors, running sequential programs communicating through a bus, with a single processor, running a multi-threaded program in a shared memory implementing the same set of functionalities. On the other hand, concurrent program verification is challenging for traditional methods. Indeed, concurrent programs generally feature a large number of possible executions due to highly non-deterministic control flows, causing test coverage to drop dramatically. Full control and data coverage, i.e., soundness, is nevertheless very important as effective errors often occur in rare but possible situations (e.g., scheduling corner cases or specific interleavings). In such a setting, sound static analysis becomes even more attractive. We have been working on an extension of Astrée to concurrent embedded C software [27]. We report here on a recent improvement concerning the specific case of programs running under a priority-based real-time scheduler.

```
void main() {
  while (x < 10) {
    y = x;
  }
}

void ISR() {
  x++;
}
```

Fig. 1. A simple program with an interrupt.

1.1 Concurrency and Scheduling Models

In the general sense, a program is concurrent if it is composed of several execution units, that we will call *threads*,[3] each with its independent control flow. The overall execution, that is, which thread executes at each time, is orchestrated by a *scheduler*. Some schedulers allow arbitrary preemption of any thread by another, others limit thread switching to selected preemption points, and finally some schedulers use priorities to determine which threads should run, as exemplified below. There also exist several methods for threads to communicate. We consider here that all the memory is shared, which provides an implicit way for threads to communicate. This is the most general model, but puts the most strain on an analyzer, as it now must determine which variables are effectively shared, and which values flow between threads. Additionnaly, threads can synchronize and enforce mutual exclusion through the use of locks.

Interrupt-Driven Programs. At one extreme, we can see interrupts in sequential programs as a simple form of concurrency. Figure 1 presents an example of a `main` function that reads a variable `x`, modified concurrently by an interrupt `ISR`. Ignoring the effect of the interrupt would not result in a sound analysis. The interrupt can occur at any time, any number of times during the execution of `main`. However, once it preempts the function `main`, `ISR` must finish before control is returned to `main`. Interrupts can be seen as similar to function calls occurring non-deterministically, and so, interrupt-driven programs can be modeled in terms of sequential programs. Naturally, interrupts can occur during interrupts, and can be arbitrarily nested (similarly, again, to function calls). Interrupt-driven programming can be found in embedded systems running on “bare metal,” that is, without a proper operating system nor any scheduler. Lightweight operating systems, such as TinyOS [21], add a notion of non-preemptive tasks. Each task is a function, and the scheduler maintains an ordered list of tasks to run. Still, while tasks can be preempted by interrupts at any point, a task cannot interrupt another task: the task function has to return to give the control back to the scheduler (so-called cooperative scheduling). We are also in the situation of a near-sequential program.

[3] In the rest of the article, we denote execution units as *threads*, even though they might be named, depending on the system studied, “processes” (ARINC 653 [4]), “tasks” (OSEK/AUTOSAR [1], TinyOS [21]), “threads” (POSIX [17]), “interrupts,” etc.

```
void t1() {              |  void t2() {
  while (1) {            |    while (1) {
    glob = 100;          |      glob = -100;
    glob += 2;           |    }
    print(glob);         |  }
  }                      |
}                        |
```

Fig. 2. Two threads interacting through a global variable `glob`.

```
void t1() {              |  void t2() {
  while (1) {            |    while (1) {
    lock(m);             |      lock(m);
    glob = 100;          |      glob = -100;
    glob += 2;           |      unlock(m);
    print(glob);         |    }
    unlock(m);           |  }
  }                      |
}                        |
```

Fig. 3. Variant of Fig. 2 where the two threads ensure mutual exclusion through a mutual exclusion lock.

Preemptive Scheduling. At another extreme lie fully preemptive systems. One example is the common multi-threaded applications found in desktop computers. In such programs, any thread can preempt a currently running thread, at any point. The result is a program execution that is an arbitrary interleaving of thread executions. The analogy with a sequential program with function calls breaks down. Figure 2 gives an example program with two threads using a shared variable `glob`. Depending on the position at which `t2` interrupts `t1`, the `print` instruction in `t1` may print 102, -98, or -100. The program is thus highly non-deterministic.

Mutexes. It may be desirable to prevent `t2` from accessing `glob` during the instruction sequence modifying and printing `glob` in `t1`. This effect can be achieved using mutexes, i.e. mutual exclusion locks, that are objects that can be owned (locked) by at most one thread at a time in the system. This is illustrated in Fig. 3: due to protection by mutex locks, the `print` instruction will now always print 102, thus reducing the non-determinism. In effect, when the currently executing thread tries to lock a mutex already owned by another thread, the scheduler puts it into a wait state until the mutex is available again. As such critical sections are usually short and far apart, to avoid contention in the system, the program execution remains highly non-deterministic. On the one hand, it is generally desirable that all accesses to shared variables are protected by locks: unprotected concurrent accesses, also called *data-races*, are known to cause miscompilations [7], as well as unexpected behaviors by exposing the lack of consistency of the memory at the hardware level [3]. On the other hand, unchecked uses of lock operations can cause *deadlocks*, a situation where a set of threads block each other by owning each a mutex required by another one. A static analysis should be able report both deadlocks and data-races, to ensure that fixing one kind of problems does not produce another kind of problems.

```
void high() {
  while (1) {
    glob = 100;
    glob += 2;
    print(glob);
    yield();
  }
}
```

```
void low() {
  while (1) {
    glob = -100;
  }
}
```

Fig. 4. Variant of Fig. 2 where the two threads ensure mutual exclusion through priorities.

```
void low() {
  while (1) {
    setPrio(999);
    glob = 100;
    glob += 2;
    print(glob);
    setPrio(0);
  }
}
```

```
void high() {
  while (1) {
    glob = -100;
    yield();
  }
}
```

Fig. 5. Variant of Fig. 2 where the lower priority thread protects a critical section by raising temporarily its priority.

```
void low() {
  while (1) {
    lock(m);
    glob++;
    if (glob > 10) glob = 0;
    unlock(m);
  }
}
```

```
void med() {
  while (1) {
    x = glob;
    yield();
  }
}
```

```
void high() {
  while (1) {
    lock(m);
    ...
    unlock(m);
    yield();
  }
}
```

Fig. 6. Priority inversion, which can be avoided by using the priority ceiling protocol.

Real-Time Scheduling. Embedded critical systems generally employ so-called *real-time schedulers* that provide low latency and help ensuring strong timing constraints. An important feature is the use of priorities for threads, that are strictly enforced: each thread is given a numeric priority, and only the thread of highest priority not blocked (e.g., not waiting to lock a mutex) can run. Figure 4 gives an example program using priorities to ensure mutual exclusion. The thread of highest priority must explicitly relinquish the control to give lower priority threads the opportunity to run. We model this using the `yield` instruction: it corresponds to a non-deterministic wait and can also model a timer (as we abstract away the physical time, the wait becomes non-deterministic) or waiting for an external event out of our control (that can thus happen at any time). In the example, we are guaranteed that the lower priority thread cannot preempt the higher priority one while it is accessing `glob`. However, the higher priority thread can preempt the lower priority one at any point of its execution (which explains why the lower priority thread does not feature a `yield` instruction). More generally, in a system, we cannot assume that a non-running thread is waiting at a `lock` or `yield` instruction (except the thread of highest priority) as it may have been preempted non-deterministically

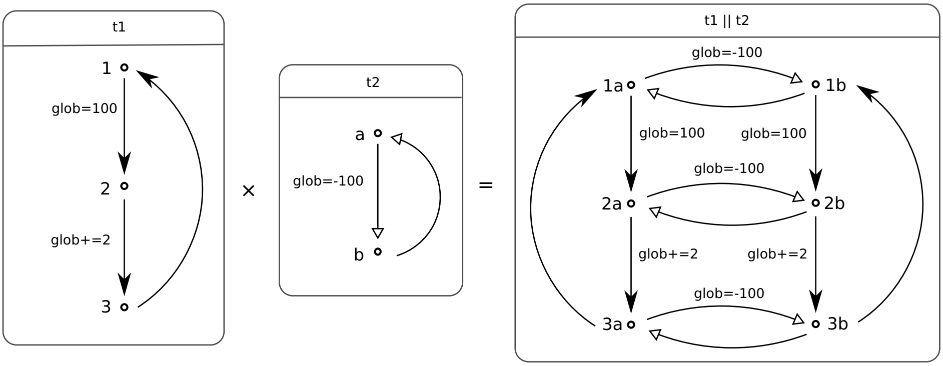

Fig. 7. Product of control-flow graphs for the analysis of the program of Fig. 2.

by a yielding higher priority thread. Despite the strict adherence to priorities, this real-time, fully-preemptive scheduling generates a large space of possible executions.

Priorities can also be dynamic. Figure 5 illustrates a common pattern where a low-priority thread temporarily elevates its priority to avoid being interrupted during a critical section. Finally, a mutex lock can be associated to a raise in priority. The goal is to avoid the case where a low-priority thread locks a mutex, then is preempted by a medium-priority thread while, at the same time, a high priority thread that wishes to lock the mutex is blocked until the medium-priority thread lets the low-priority thread run again and release the mutex. This undesirable scenario, called *priority inversion*, is illustrated in Fig. 6. It can be avoided by raising the priority of thread `low` while it holds mutex `m`, so that it cannot be preempted by `med`. The so-called *priority ceiling protocol*,[4] implemented in operating systems such as OSEK/AUTOSAR [1] or POSIX [17], automates this by associating to each mutex a minimum priority: every thread is elevated to this priority upon locking the mutex, and resumes its former priority when unlocking the mutex. When using this technique in Fig. 6, there is no data-race as thread `med` cannot observe `glob` while either `low` or `high` are modifying it.

To sum up, priorities can be employed in place of, or in addition to mutexes to enforce mutual exclusion and remove data-races. Thus, it is important to take priorities into account to achieve a sufficiently precise static analysis.

1.2 Analysis of Concurrent Software, Related Work

A variety of methods have been developed to analyze concurrent systems, and in particular multi-threaded software in a shared memory. Some methods are not sound, or not scalable, or only adapted to a specific scheduling model.

Sequentialization Methods. Given the large number of techniques and tools available to analyze sequential programs, one attractive solution is to reduce the problem to a sequential analysis. Qadeer and Wu [33] pioneered such transformations in their

[4] The technique is sometimes called *immediate priority ceiling protocol*, to distinguish it from a variant, where a low-priority thread inherits the priority of any higher-priority thread it blocks. The former is more common in safety critical software, hence our choice to include it here. Nevertheless, our framework can be easily extended to handle both variants.

```
void main() {             void schedule() {
  schedule();               int oldPrio = prio;
  while (x < 10) {          for (int i = 0; i < N; i++) {
    schedule();               if (i > prio && nondet()) {
    y = x;                      prio = i;
    schedule()                  ISR[i].entry();
  }                           }
  schedule(); }             }
                            prio = oldPrio;
void ISR() {              }
  schedule();
  x++;
  schedule();
}
```

Fig. 8. Sequentialization of program Fig. 1, on [39].

KISS tool. Their idea is to weave calls to a synthesized scheduler function throughout the program, and feed the resulting program to the SLAM model-checker. The method is complete, as an error trace on the sequential program corresponds to a trace on the concurrent one, but it is not sound and may miss errors.

In the special case of interrupt-driven programs, however, the sequentialization transformation itself can be made sound and complete. This technique has been combined with an abstract interpreter to construct a sound (but incomplete) static analysis of interrupt-driven programs [39], or of sequential programs (a USB driver) interacting with complex environments (a USB controller) [29]. Figure 8 illustrates the transformation from [39], applied to the program from Fig. 1. For large programs, if handled naively by inlining, the additional calls may threaten the scalability of the analysis. To combat this, one natural solution is to employ modular inter-procedural analysis techniques. For instance, [31] analyzes TinyOS drivers using a context-insensitive abstraction of interrupt handlers.

Graph-Based Methods. A more classic and general method consists in reasoning on the control-flow graph of the threads. On the one hand, the analysis of arbitrary control-flow graphs can be achieved by model-checking or abstract interpretation [8]. On the other hand, a control-flow graph modeling all interleavings of threads can be constructed as a product of the graphs of the individual threads. This is illustrated in Fig. 7, for the example of Fig. 2. Even on such a simple example, a significant drawback becomes apparent: the resulting graph can be very large. The analysis seems only practical for small programs; it is implemented for instance in the ConcurInterproc academic tool [18]. Partial order reduction methods, introduced by Godefroid [16], attempt to address the problem by soundly removing redundant computations during model checking, but may not be sufficient for large programs. Context bounded model checking [32] has been advocated as a more practical solution; however, this method does not remain sound as it considers only small prefixes of executions. Both regular (SPIN-based) and bounded model-checking have been applied to the analysis of OSEK software, taking priorities into account [40].

Even when considering sequential programs, graph-based methods may not be suitable for high-precision whole-program analysis of large software, because of the difficulty to fit in the memory an invariant for each program point. The Astrée analyzer [6], which is whole-program and employs relational, flow-sensitive, context-sensitive, partially path-sensitive abstractions, opts for an iteration by induction on the program syntax, which requires far less memory (intuitively, in the depth of nested loops and conditionals after inlining, instead of in the program length).

Thread-Modular Methods. Prior work [24] and the work of Carré and Hymans [10] proposed thread-modular static analyses, inspired from the seminal work of Jones on rely–guarantee proof methods [19]. In these analyses, each thread is analyzed in isolation, first ignoring the effect of other threads. During this analysis, the set of interference, i.e., possible values stored in global variables, is gathered. The threads are then reanalyzed, but injecting this time the possible interference gathered from the other threads. Generally, this uncovers new possible behaviors, and so new interference. Hence, the thread analyses are iterated until the increasing set of interference stabilizes, at which point we are guaranteed to cover at least all possible thread interleavings. On the example of Fig. 2, we get, in the first analysis round, $[\texttt{glob} \mapsto \{100, 102\}]$ for `t1` and $[\texttt{glob} \mapsto \{-100\}]$ for `t2`. In the second analysis round, the assignment `glob += 2` from `t1` can now also store -98 into `glob`. In the third round, interference are stable, and we indeed get that `print(glob)` can print 102, -98, or -100, which is sound.

Integration within the Astrée analyzer and experimentation on large (2 Mloc) embedded avionics software [24] demonstrated the scalability of the method. In particular, few (around 6) rounds of reanalysis are necessary to stabilize interference, i.e., the sound analysis of a concurrent software is less than one order of magnitude more costly than that of a sequential program of similar size. Moreover, the concurrent analysis is based on a simple modification of a sequential analysis (for each thread), and so, can benefit from existing abstractions and implementations available for these (in our case, the Astrée analyzer). Additionnaly, mutual exclusion can be handled in [24] through partitioning of interference with respect to the mutex protecting them. On the example of Fig. 3, the analysis would gather that only the interference $[\texttt{glob} \mapsto \{102\}]$ from `t1` is visible to other threads that access `glob` while holding mutex `m` (such as `t2`), but the analysis would also keep $[\texttt{glob} \mapsto \{100, 102\}]$ and apply it to threads accessing `glob` while not owning `m`.

These works are limited to non-relational and flow-insensitive handling of interference. This was addressed in [26], which first reformulated these methods as an abstraction of a complete (but uncomputable) concrete semantics already in thread-modular form, and then showed how possibly relational and flow-sensitive abstractions could be derived. This approach was pioneered by Cousot and Cousot for non-modular methods [13], and extended in [26] to thread-modular methods. Deriving the static analysis as an abstraction of a complete concrete semantics is very attractive from a theoretical point of view, as it means that, for any correct program, an effective static analysis able to prove it correct can be constructed. The idea of adapting thread-modular techniques to program analysis has also been applied to model-checking [15], although it is often limited to non-relational abstractions of interference [22].

Note that these methods handle priorities and real-time scheduling imprecisely (if at all). They would not be able to infer and exploit mutual exclusion in the examples of Figs. 4–6, and would instead generate spurious interference. The Goblint analyzer [37] is another sound static analyzer of concurrent C software. While it focuses mainly on data-races and uses simple numeric abstractions, it is the only prior sound analyzer we know of that supports the priority ceiling protocol [35].

1.3 Contribution

In this article, we propose a small improvement on the thread-modular static analysis implemented in Astrée to check the absence of run-time errors and data-races [24,26]. Our goal is to improve the precision by taking into account the priorities of threads, including dynamic priorities, assuming they are run by a real-time scheduler. For instance, the analysis of the examples from Figs. 4–5 using priorities will provide, with our new analysis, the same level of precision than when using mutexes to ensure mutual exclusion (Fig. 3). Similarly to our previous work [24], the improvement is implemented through a judicious use of interference partitioning. However, following [26], we justify the correctness of the approach by expressing it as an abstraction of a concrete, thread-modular semantics. Moreover, we extend [26] from a state-based semantics to a trace-based semantics in order to develop the history-sensitive abstractions necessary to justify our method. Our main targets are software running under OSEK/AUTOSAR operating systems [1]. Thus, we also support the priority ceiling protocol used in such software (Fig. 6). Our final contribution is a simple sound deadlock analysis that exploits the precise results of the analysis in terms of reachability to detect all possible deadlock cycles.

Limitations. As in [24], we are currently limited to non-relational interference abstractions (although each thread analysis is fully relational). Combination with relational interference abstractions from [26] remains future work. Additionnaly, we largely ignore the effects of weak memory models, although this is somewhat offset by the ability of the analysis to find all data-races, and so, warn the programmer about possible hazards related to memory inconsistency [3]. Finally, we focus solely on mono-core applications. The reason is that current real-time systems and applications, including all our target applications, are largely mono-core and exploit the fact that only one thread runs at a time. The analysis must also exploit it. Previous work that ignored priorities [24,26] was sound for multi-core applications.

Although we have implemented our method within the Astrée analyzer, experimentation is still on-going. We can only report for now on the analysis of small demonstration example OSEK software of a few hundred lines, and present the small precision improvement we bring compared to the results to the former, priority-unaware version of Astrée.

$$\begin{array}{rcll}
\mathit{prog} & ::= & \mathit{thread}_1 \,||\, \mathit{thread}_2 \,||\, \cdots \,||\, \mathit{thread}_n & \textit{(program)} \\
\mathit{thread} & ::= & {}^{[\ell]}\,\mathit{stat}\,{}^{[\ell']} & \textit{(thread)} \\
{}^{[\ell]}\,\mathit{stat}\,{}^{[\ell']} & ::= & {}^{[\ell]}\mathcal{V} \leftarrow \mathit{expr}\,{}^{[\ell']} & \textit{(assignment)} \\
 & | & {}^{[\ell]}\,\mathit{stat}\,;{}^{[\ell_1]}\,\mathit{stat}\,{}^{[\ell']} & \textit{(sequence)} \\
 & | & {}^{[\ell]}\textbf{if}\ \ \mathit{expr}\ \ \textbf{then}\ {}^{[\ell_1]}\ \ \mathit{stat}\ \ \textbf{endif}^{[\ell']} & \textit{(conditional)} \\
 & | & {}^{[\ell]}\textbf{while}\ {}^{[\ell_1]}\ \ \mathit{expr}\ \ \textbf{do}\ {}^{[\ell_2]}\ \ \mathit{stat}\ \ \textbf{done}^{[\ell']} & \textit{(loop)} \\
 & | & {}^{[\ell]}\textbf{lock}(\mathcal{M})^{[\ell']} & \textit{(mutex lock)} \\
 & | & {}^{[\ell]}\textbf{unlock}(\mathcal{M})^{[\ell']} & \textit{(mutex unlock)} \\
 & | & {}^{[\ell]}\textbf{yield}^{[\ell']} & \textit{(non-deterministic wait)} \\
 & | & {}^{[\ell]}\textbf{setpriority}(\mathbb{N})^{[\ell']} & \textit{(change priority)}
\end{array}$$

Fig. 9. Language syntax.

1.4 Overview

The remaining of the article is organized as follows: we presents our abstract language and its concrete semantics; first, as a classic monolithic fixpoint interleaving all the threads in Sec. 2, then, in a thread-modular form as a fixpoint of thread semantics in Sec. 3; Sec. 4 then presents an abstraction of this semantics into an effective static analyzer parameterized by an arbitrary domain for intra-thread analysis and a partitioned, non-relational domain for inter-thread interference analysis; Sec. 5 presents our deadlock analysis; Sec. 6 discusses the implementation within the Astrée analyzer and early experimental validation; Sec. 7 concludes.

2 Concrete Semantics

This section presents our language and its concrete semantics, that is, the most precise mathematical definition of program behaviors. We will progressively add features until we achieve a concrete semantics that is thread-modular, takes thread priorities and real-time scheduling into account, and is complete for safety properties (such as invariants, data-race freedom, and deadlock freedom). The final concrete semantics will be uncomputable, but amenable to abstractions, as shown in Sec. 4.

2.1 Language

For the sake of presentation, and in order to offer a full formal treatment, we consider a simple, abstract language, with a minimum number of constructions, focusing notably on those relevant to concurrency and real-time scheduling. Nevertheless, the method has been applied to a real language, C, as reported in Sec. 6.

Syntax. The grammar of the language is presented in Fig. 9. A program prog is composed of a finite set of threads thread_t numbered from $t = 1$ to $t = n$. We will write as $\mathcal{T} \overset{\text{def}}{=} \{1, 2, \ldots, n\}$ the set of thread numbers. A thread is a single statement *stat*. Statements in the sequential fragment include classic assignments, sequences, if-then conditionals, and loops. We denote as $\mathcal{V}$ the (finite, fixed) set of variables. We use expressions *expr*, but do not specify what they are: their precise syntax is irrelevant to our analysis; they contain at least boolean values $\mathtt{tt}$ and $\mathtt{ff}$, but could also contain integer arithmetic, floats, pointers, etc. The concurrency-specific statements include locking (**lock**) and unlocking (**unlock**) a mutex, non-deterministic wait (**yield**), and changing the priority of the current thread to some integer value (**setpriority**). We denote as $\mathcal{M}$ the (finite, fixed) set of mutexes. Finally, all the statements in our program are decorated with labels ℓ. Every statement has a label before it, a label after it, and possibly labels in-between; these will facilitate the definition of the transition system semantics in the following sections. We denote as $\mathcal{L}$ the (finite) set of labels in the program.

Limitations. The main limitations of our language are as follows. Firstly, the sets $\mathcal{V}$ of variables, $\mathcal{M}$ of mutexes, and $\mathcal{T}$ of threads are fixed, i.e., there is no dynamic allocation of resources. Secondly, there are not procedures. These limitations are actually guided by our application domain and also apply to our C analyzer (Sec. 6): as we are analyzing embedded software, the threads, mutexes and variables are statically allocated (or allocated through a sequential initialization phase that we analyze beforehand). Moreover, the C analyzer performs an interprocedural analysis sensitive to the full call stack, through semantic inlining of all functions calls; this prevents us from supporting unbounded recursive calls, but these are forbidden in embedded software. As a consequence, the set of local variables is unambiguously determined at every point of the analysis by the call stack currently analyzed.

2.2 Interleaving Semantics

We present a concrete semantics for our language. At first, we consider that an execution of the program is an arbitrary interleaving of thread executions, up to mutual exclusion enforced by locks. We thus ignore for now the effect of priorities and real-time scheduling, which will be accounted for in the next section.

Transition System. At the most concrete level, the semantics of the program is described as a discrete labelled transition system $(\Sigma, \mathcal{A}, \tau)$, composed of: a set Σ of states, a set $\mathcal{A}$ of actions, and a transition relation $\tau \subseteq \Sigma \times \mathcal{A} \times \Sigma$ modeling the effect of individual program statements as discrete transitions from one state to another state following some action. This model is quite standard [11] as it allows deriving a variety of semantics expressing important program properties, such as reachability, safety, and liveness, and then abstract them into static analyzers for such properties.

States. Our program states are composed of several parts, collecting information about the memory, the control location, and the scheduler state. A memory state $\rho \in \mathcal{E} \overset{\text{def}}{=} \mathcal{V} \to \mathbb{V}$ assigns a value in $\mathbb{V}$ to each variable in $\mathcal{V}$. A control state $c \in \mathcal{C} \overset{\text{def}}{=} \mathcal{T} \to \mathcal{L}$ assigns a control location in $\mathcal{L}$ to each thread in $\mathcal{T}$. In addition,

$$\underline{\tau[\mathit{prog}] \subseteq \Sigma \times \mathcal{A} \times \Sigma}$$

$$\tau[\mathit{prog}] \overset{\text{def}}{=} \cup_{t \in \mathcal{T}}\ \tau_t[^{[\ell_t]}\, \mathit{stat}_t\, ^{[\ell'_t]}]$$

where $\mathit{prog} = \mathit{thread}_1 \ || \cdots || \ \mathit{thread}_n$ and $\forall t \in \mathcal{T} :\ \mathit{thread}_t = {}^{[\ell_t]}\, \mathit{stat}_t\, ^{[\ell'_t]}$

$$\tau_t[^{[\ell_1]} V \leftarrow e^{[\ell_2]}] \overset{\text{def}}{=}$$
$$\{ (c, \rho, o, \pi) \xrightarrow{t:V \leftarrow e} (c[t \mapsto \ell_2], \rho[V \mapsto v], o, \pi) \mid c(t) = \ell_1 \wedge v \in \mathbb{E}[\![e]\!]\rho \}$$

$$\tau_t[^{[\ell_1]} s;^{[\ell_2]} s'^{[\ell_3]}] \overset{\text{def}}{=} \tau_t[^{[\ell_1]} s^{[\ell_2]}] \cup \tau_t[^{[\ell_2]} s'^{[\ell_3]}]$$

$$\tau_t[^{[\ell_1]} e?^{[\ell_2]}] \overset{\text{def}}{=} \{ (c, \rho, o, \pi) \xrightarrow{t:e?} (c[t \mapsto \ell_2], \rho, o, \pi) \mid c(t) = \ell_1 \wedge \mathsf{tt} \in \mathbb{E}[\![e]\!]\rho \}$$

$$\tau_t[^{[\ell_1]} \textbf{if}\ e\ \textbf{then}\ ^{[\ell_2]}\ s\ \textbf{endif}^{[\ell_3]}] \overset{\text{def}}{=} \tau_t[^{[\ell_1]} e?^{[\ell_2]}] \cup \tau_t[^{[\ell_2]} s^{[\ell_3]}] \cup \tau_t[^{[\ell_1]} \neg e?^{[\ell_3]}]$$

$$\tau_t[^{[\ell_1]} \textbf{while}\ ^{[\ell_2]}\ e\ \textbf{do}\ ^{[\ell_3]}\ s\ \textbf{done}^{[\ell_4]}] \overset{\text{def}}{=}$$
$$\tau_t[^{[\ell_1]} \mathsf{tt}?^{[\ell_2]}] \cup \tau_t[^{[\ell_2]} e?^{[\ell_3]}] \cup \tau_t[^{[\ell_3]} s^{[\ell_2]}] \cup \tau_t[^{[\ell_2]} \neg e?^{[\ell_4]}]$$

$$\tau_t[^{[\ell_1]} \textbf{lock}(m)^{[\ell_2]}] \overset{\text{def}}{=}$$
$$\{ (c, \rho, o, \pi) \xrightarrow{t:\textbf{lock}(m)} (c[t \mapsto \ell_2], \rho, o[m \mapsto t], \pi) \mid c(t) = \ell_1 \wedge o(m) = \bot \}$$

$$\tau_t[^{[\ell_1]} \textbf{unlock}(m)^{[\ell_2]}] \overset{\text{def}}{=}$$
$$\{ (c, \rho, o, \pi) \xrightarrow{t:\textbf{unlock}(m)} (c[t \mapsto \ell_2], \rho, o[m \mapsto \bot], \pi) \mid c(t) = \ell_1 \wedge o(m) = t \}$$

$$\tau_t[^{[\ell_1]} \textbf{yield}^{[\ell_2]}] \overset{\text{def}}{=} \{ (c, \rho, o, \pi) \xrightarrow{t:\textbf{yield}} (c[t \mapsto \ell_2], \rho, o, \pi) \mid c(t) = \ell_1 \}$$

$$\tau_t[^{[\ell_1]} \textbf{setpriority}(p)^{[\ell_2]}] \overset{\text{def}}{=}$$
$$\{ (c, \rho, o, \pi) \xrightarrow{t:\textbf{setpriority}(p)} (c[t \mapsto \ell_2], \rho, o, \pi[t \mapsto p]) \mid c(t) = \ell_1 \}$$

Fig. 10. Transition system for a concurrent program.

we must track mutex locking. As a mutex can only be locked by a single thread at a time, we store this information as an ownership map $o \in \mathcal{O} \overset{\text{def}}{=} \mathcal{M} \to \mathcal{T}_\bot$, where $\mathcal{T}_\bot$ is $\mathcal{T} \cup \{\bot\}$, indicating which thread $o(m) \in \mathcal{T}$ owns (i.e., has locked) a mutex m, while $o(m) = \bot$ indicates that m is not locked by any thread. Finally, as thread priorities can be changed dynamically, we keep a map $\pi \in \Pi \overset{\text{def}}{=} \mathcal{T} \to \mathbb{N}$ denoting the current priority of each thread. To sum up, the program states live in:

$$\Sigma \overset{\text{def}}{=} \mathcal{C} \times \mathcal{E} \times \mathcal{O} \times \Pi\ . \qquad (1)$$

Transitions. We use actions, in $\mathcal{A}$, on transitions to remember which thread generates each transition as well as the nature of the instruction executed, i.e.:

$$\begin{aligned} &\mathcal{A} \stackrel{\text{def}}{=} \mathcal{T} \times \mathit{atomic}\text{, where} \\ &\mathit{atomic} \stackrel{\text{def}}{=} \{\, V \leftarrow e,\ e?,\ \textbf{yield},\ \textbf{lock}(m),\ \textbf{unlock}(m),\ \textbf{setpriority}(p)\,\} \end{aligned} \tag{2}$$

where $(t, a) \in \mathcal{A}$ denotes that t performs some atomic action a (assignment, guard, locking, etc.). The transitions live in $\Sigma \times \mathcal{A} \times \Sigma$. The transition system can be derived statically from the program, by induction on the syntax, as shown in Fig. 10. We denote as $\tau_t[^{[\ell]}\ \mathit{stat}\ ^{[\ell']}]$ the set of transitions generated by a statement stat from thread t between control locations ℓ and ℓ'. The transitions $\tau[\mathit{prog}]$ generated by the whole program $\mathit{prog} \stackrel{\text{def}}{=} \mathit{thread}_1 \,||\, \cdots \,||\, \mathit{thread}_n$ is the union of the transitions generated by the statement $^{[\ell_i]}\ \mathit{stat}_i\ ^{[\ell'_i]}$ of each thread thread_i. We use the notation $\sigma \xrightarrow{t:i} \sigma'$ as a shortcut for the fact that the transition $(\sigma, (t, i), \sigma')$ exists in $\tau[\mathit{prog}]$.

All the transitions update the control part $c(t)$ of the thread t they belong to, while leaving the control part of the other threads intact. The semantics of classic sequential constructs is standard: the assignment updates the memory state, while the semantics of conditionals, sequences, and loops is by induction. We introduced a synthetic *guard* statement $e?$ which is useful to simplify the definition of conditionals and loops (note that $\mathtt{tt}?$ denotes an unconditional jump). As we did not specify the syntax of expressions, we do not make any hypothesis on the set $\mathbb{V}$ of program values either, and also leave this set unspecified (except for the presence of boolean values $\mathtt{tt}$ and $\mathtt{ff}$). We assume the existence of a function $\mathbb{E}[\![\, \mathit{expr} \,]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{V})$ able to return the set of possible values an expression can evaluate to in a given memory state $\rho \in \mathcal{E}$. For the sake of generality, the semantics of expressions returns a set: we can thus easily model non-deterministic expressions (that return several possible values, all of which must be considered by the analysis) and errors (i.e., returning no value at all, blocking program execution).

The transitions generated from concurrent statements are straightforward: in addition to updating $c(t)$, they update the mutex ownership map o (**lock**, **unlock**), or the priority map π (**setpriority**). For now, **yield** is a “no-op” in the transition system. It will be given a special significance for real-time schedulers in Sec. 2.3, as it allows other threads to run even if they have a lower priority.

Initial State. We denote as $E \stackrel{\text{def}}{=} \{(c_0, \rho_0, o_0, \pi_0)\}$ the set of initial states, reduced here to a single state where c_0 maps each thread control location to its initial control point, ρ_0 maps every variable to 0, o_0 maps every mutex to $\perp$, and π_0 maps threads to their initial priority.

Interleaving Semantics. A transition system provides a very static view of a program semantics. Information about the dynamic semantics, i.e., the possible executions of the program, can be derived classically [11] using fixpoints. A possible execution is modeled as a so-called *trace*, i.e., a sequence of states interspersed with actions. A trace is noted as $\sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \overset{a_1}{\rightarrowtail} \cdots \sigma_n$, with $\forall i : \sigma_i \in \Sigma$, $a_i \in \mathcal{A}$, while we denote as $\Sigma\mathcal{A}^*$ the set of all traces. We are in particular interested in observable program traces, i.e., traces starting in an initial state and where consecutive states are related by the transition relation. Note that, to avoid any confusion, we use a different kind of arrows for traces $\overset{a}{\rightarrowtail}$ and for the transition relation $\xrightarrow{a}$. The trace semantics $\mathcal{F}$, which collects all program traces, can be classically expressed as a fixpoint [11]:

$$\mathcal{F} \overset{\text{def}}{=} \text{lfp}\, \lambda T.\, E \cup \{\, \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_{n+1} \mid \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \in T \wedge \sigma_n \xrightarrow{a_n} \sigma_{n+1} \,\} \ .$$

It corresponds to modeling program executions as arbitrary interleavings of thread executions. Note that this semantics ignores, for now, thread priorities but respects the mutual exclusion property enforced by mutexes, by definition of the set of transitions generated by a **lock** instruction.

Most analyses are not based on a trace semantics, but rather on a state semantics, which is simpler and can nevertheless express important correction properties, such as the absence of run-time errors. The set $\mathcal{R}$ of reachable states can be expressed, similarly to $\mathcal{F}$, as a fixpoint:

$$\mathcal{R} \overset{\text{def}}{=} \text{lfp}\, \lambda S.\, E \cup \{\, \sigma' \mid \exists \sigma \in S,\, a \in \mathcal{A} : \sigma \xrightarrow{a} \sigma' \,\} \ .$$

The reachability semantics $\mathcal{R}$ can be seen as an abstraction, through $\alpha_{\mathcal{R}}$, of the trace semantics $\mathcal{F}$, which collects the last reachable state of each trace:

$$\mathcal{R} = \alpha_{\mathcal{R}}(\mathcal{F}),\ \text{where}\ \alpha_{\mathcal{R}}(T) \overset{\text{def}}{=} \{\, \sigma \mid \exists \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \in T : \sigma = \sigma_n \,\} \ . \qquad (3)$$

Even when we are ultimately interested in state properties, it is worthwhile to derive our static analysis from a trace semantics, instead of a state semantics, as it facilitates the design of precise and efficient computable abstractions by exposing information about the history of computations. For instance, trace partitioning [23] is a popular such abstraction: it uses (parsimoniously) trace information in order to turn a flow-sensitive analysis into a path-sensitive one.

Granularity and Memory Models. Our semantics considers that an assignment $V \leftarrow e$ evaluates the expression e and updates V in a single, atomic step, i.e., thread preemption cannot occur in the middle of this operation. This may not be realistic: in a real language, two threads executing concurrently $V \leftarrow V + 1$ may actually increase V by one instead of two, assuming preemption occurs after the first thread evaluates $V + 1$ and before it updates V. One solution would be to further split expression evaluation, using temporaries, to match the granularity of the language (e.g., $\mathit{tmp} \leftarrow V + 1; V \leftarrow \mathit{tmp}$). Reynolds advocates instead the use of a "grainless" semantics [34], where such thread interactions are considered as errors. Our static analyzer will also be consistent with this semantics as it reports them as data-races.

This problem is also tied to the issue of weak memory models [5] where, due to compiler and hardware optimization interacting with thread preemption, the program exhibits more behaviors than the interleavings of threads. While modeling directly popular weak memory models in a static analyzer is possible [36], we do not discuss it here and only focus on sequentially consistent memories. Note, however, that current high-level models tend towards the "data-race freedom" guarantee, i.e., if there are no data-races in any sequentially consistent execution, then there are no other possible executions than the sequentially consistent ones. As our analyzer will report all data-races, it becomes possible to check *a posteriori* whether all behaviors have been taken into account by the analysis or not.

2.3 Real-Time Concurrent Semantics

Compared to the interleaving semantics of Sec. 2.2, a real-time scheduler obeys thread priorities strictly, and uses them to restrict the set of allowed transitions.

Priorities. To model the (immediate) priority ceiling protocol, a program assigns a priority ceiling to each mutex, denoted here as $\mathit{mprio} : \mathcal{M} \to \mathbb{N}$, indicating the value that the priority of a thread will be raised to upon locking the mutex. This is a static information, provided as part of the program but not the program state. Thread priorities are, however, dynamic. We define the *actual priority* $\mathit{tprio}(t)(\sigma)$ of a thread t in a specific state $\sigma \overset{\text{def}}{=} (c, \rho, o, \pi)$, taking into account the current priority map π as well as the static priority mprio of the mutexes it currently owns:

$$\mathit{tprio}(t)(c, \rho, o, \pi) \overset{\text{def}}{=} \max\left(\{\, \pi(t) \,\} \cup \{\, \mathit{mprio}(m) \mid m \in \mathcal{M},\, o(m) = t \,\}\right) \ . \tag{4}$$

Scheduling. In a real-time mono-core scheduler, only the runnable thread of highest priority can run. By runnable, we mean that it is neither waiting at a **lock** statement for a lock to become available, nor finished, nor waiting at a **yield** statement for some external event. In order to develop thread-modular analyses, it is useful to define the scheduler by stating, for a given thread t in a given program state σ, whether it can run or not, i.e., whether it is not necessarily blocked by a higher priority thread. We thus define the "enabled" predicate $\mathit{enbl}(t, \sigma)$ as:

$$\begin{aligned} \mathit{enbl}(t, \sigma) \ \overset{\text{def}}{\iff} \ & \forall t' \neq t \in \mathcal{T} : \ \mathit{tprio}(t')(\sigma) \leq \mathit{tprio}(t)(\sigma) \\ & \qquad \vee\ \exists \sigma' : \sigma \xrightarrow{t':\mathbf{yield}} \sigma' \ \vee\ \nexists i, \sigma' : \sigma \xrightarrow{t':i} \sigma' \end{aligned} \tag{5}$$

This rule explicitly prevents a yielding or blocked thread from blocking lower priority threads. In case there are yielding threads, there may be several simultaneously enabled threads (e.g., the yielding thread, and a lower priority thread), leading to a non-deterministic behavior. If several threads are waiting for a mutex to be unlocked, then, upon unlocking, our semantics states naturally that the thread of highest priority waiting for the mutex will run immediately and acquire it, blocking lower priority threads. The trace semantics $\mathcal{F}$ then becomes:

$$\begin{aligned} \mathcal{F} \overset{\text{def}}{=} \ \mathrm{lfp}\, \lambda T.\, E \cup \{\, & \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \overset{t:i}{\rightarrowtail} \sigma_{n+1} \mid \\ & \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \in T \wedge \mathit{enbl}(t, \sigma_n) \wedge \sigma_n \xrightarrow{t:i} \sigma_{n+1} \,\} \end{aligned} \tag{6}$$

and the reachable states are:

$$\mathcal{R} \overset{\text{def}}{=} \ \mathrm{lfp}\, \lambda S.\, E \cup \{\, \sigma' \mid \exists \sigma \in S,\, i \in \mathit{atomic} : \sigma \xrightarrow{t:i} \sigma' \wedge \mathit{enbl}(t, \sigma) \,\} \ . \tag{7}$$

We also have $\mathcal{R} = \alpha_{\mathcal{R}}(\mathcal{F})$, using the same reachability abstraction $\alpha_{\mathcal{R}}$ as for the interleaving semantics (3).

Limitations. In our semantics, if two threads have the exact same priority, they can preempt each other arbitrarily (none blocks the other). In practice, however, in order for a system to behave in a more deterministic way, real-time programs often opt instead for a scheduler where a thread can only be preempted by a thread with strictly greater priority. Modeling this would require tracking additionally, in the program state, the thread which performed the last step, and ensuring it keeps running in case of priority tie. This is not conceptually difficult, but we avoid presenting this complication here as it would obscure the semantics. Our semantics allows more behaviors, and so, is nevertheless sound with respect this alternate, more strict kind of schedulers.

The interleaving semantics of Sec. 2.2 actually models multi-core systems but, here, when taking priorities into account, we assumed a mono-core system, where at most one thread runs at a time. There exist several ways to extend mono-core real-time schedulers to the n-core case, such as scheduling the n highest priority threads, or pinning thread to cores and scheduling on each core the highest priority thread in the set associated to the core. We believe that our framework could handle these cases painlessly through an adaptation of the *enbl* function (5), but we do not discuss this further and leave the handling of multi-core as future work.

3 Thread-Modular Concrete Semantics

The concrete semantics we proposed in the previous section is still monolithic: it is expressed as a fixpoint interleaving actions from all the threads in an arbitrary order. We now propose a thread-modular version, which is based on previous work for the interleaving semantics [25] implemented inside Astrée [6], but adapted here to a priority-aware real-time trace-based semantics.

3.1 Nested Fixpoint Semantics

Our modular semantics is based on the *rely-guarantee* principle [19]: each thread is analyzed separately, assuming some information about the effect of the environment, i.e., the other threads. Given a thread $t \in \mathcal{T}$, we give an expression of the program execution traces, denoted now as $\mathcal{F}_M(t, I)$ (standing for "modular semantics"), that only explores the transition system generated by $thread_t$. It also takes as parameter a set $I \in \mathcal{P}(\mathcal{I})$ of transitions from the environment, so-called "interference," where $\mathcal{I} \overset{\text{def}}{=} \Sigma \times \mathcal{A} \times \Sigma$ here. From the point of view of thread t, an execution step consists in either applying some interference from the environment or, when reaching a state enabled for t, a step from thread t, i.e.:

$$\begin{aligned}\mathcal{F}_M(t,I) \overset{\text{def}}{=} \text{lfp}\,\lambda T.\,E \cup \{\,\sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \overset{t':i}{\rightarrowtail} \sigma_{n+1} \mid \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \in T \wedge \\ ((t = t' \wedge \mathit{enbl}(t,\sigma_n) \wedge \sigma_n \xrightarrow{t:i} \sigma_{n+1}) \vee \\ (t \neq t' \wedge \langle \sigma_n, (t',i), \sigma_{n+1} \rangle \in I))\,\}\end{aligned} \tag{8}$$

A natural way to recover exactly the program semantics $\mathcal{F}$ (6) from $\mathcal{F}_M$ is to provide, as interference I, the transitions effectively appearing in $\mathcal{F}$, i.e., $\alpha_{\rightarrowtail}(\mathcal{F})$, where the abstraction $\alpha_{\rightarrowtail}(T)$ gathers the transitions in the set of traces T:

$$\alpha_{\rightarrowtail}(T) \overset{\text{def}}{=} \{\,\langle \sigma_k, a_k, \sigma_{k+1} \rangle \mid \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \in T \wedge k < n\,\}\ . \tag{9}$$

Note that $\alpha_{\rightarrowtail}(\mathcal{F})$ is a subset of the transition relation $\tau[\mathit{prog}]$ of the full program, but can be much smaller as it takes real-time scheduling into account (enbl) and discards transitions that are unreachable from the initial environment.

Intuitively, $\forall t \in \mathcal{T} : \mathcal{F} = \mathcal{F}_M(t, \alpha_{\rightarrowtail}(\mathcal{F}))$, i.e., we have expressed $\mathcal{F}$ as a solution of a system of fixpoint equations. This is formalized more precisely in the following theorem:

Theorem 3.1 $\mathcal{F} = \text{lfp}\,\lambda T.\,\cup_{t\in\mathcal{T}}\ \mathcal{F}_M(t, \alpha_{\rightarrowtail}(T))$.

Proof. In Appendix A.1. □

As $\mathcal{F}_M$ is itself defined as a fixpoint, we have expressed $\mathcal{F}$ as a nested fixpoint. Using Kleene's theorem [11], we can provide an alternate expression for $\mathcal{F}$, as the limit of a sequence of iteration $\cup_{i\in\mathbb{N}} T_i$, where:

$$\begin{cases} T_0 & \overset{\text{def}}{=} \emptyset \\ T_{i+1} & \overset{\text{def}}{=} \cup_{t\in\mathcal{T}}\,\mathcal{F}_M(t, \alpha_{\rightarrowtail}(T_i))\ . \end{cases} \tag{10}$$

As $\mathcal{F}_M(t,I)$ only uses the part of the transition relation $\rightarrow$ corresponding to thread t, it is similar to a sequential analysis of t, up to the application of interference from I (this aspect will become more obvious when switching to a denotational-style semantics of threads, in Sec. 3.2). We have expressed that $\mathcal{F}$ can be computed by iterating sequential analyses of individual threads, i.e., our expression is indeed thread-modular. This fixpoint formulation also expresses that, unlike classic rely-guarantee [19], the environment is inferred and not provided by the user.

Comparison with Previous Work. The nested fixpoint formulation is similar to [25] with two main differences. Firstly, [25] models the interleaving semantics, while we model a priority-aware real-time semantics, thanks to the use of the *enbl* function. Secondly, we express a trace semantics, while [25] presents a state semantics. As stated before, traces allow history-sensitive abstractions. We could, for instance, apply trace partitioning [23] on the analysis of each individual thread to achieve

$$\underline{\mathbb{S}\llbracket\, {}^{[\ell_1]}\, \mathit{stat}\, {}^{[\ell_2]} \,\rrbracket : \mathcal{T} \to \mathcal{D} \to \mathcal{D}}$$

$$\mathbb{S}\llbracket\, {}^{[\ell_1]} s ;^{[\ell_2]} s'^{[\ell_3]} \,\rrbracket(t) \stackrel{\text{def}}{=} \mathbb{S}\llbracket\, {}^{[\ell_2]} s'^{[\ell_3]} \,\rrbracket(t) \circ \mathbb{S}\llbracket\, {}^{[\ell_1]} s^{[\ell_2]} \,\rrbracket(t)$$

$$\mathbb{S}\llbracket\, {}^{[\ell_1]}\textbf{if } e \textbf{ then } {}^{[\ell_2]}\, s \textbf{ endif}^{[\ell_3]} \,\rrbracket(t)(D) \stackrel{\text{def}}{=}$$
$$\qquad \mathbb{S}\llbracket\, {}^{[\ell_2]} s^{[\ell_3]} \,\rrbracket(t)(\mathbb{S}\llbracket\, {}^{[\ell_1]} e?^{[\ell_2]} \,\rrbracket(t)(D)) \,\dot\cup\, \mathbb{S}\llbracket\, {}^{[\ell_1]} \neg e?^{[\ell_3]} \,\rrbracket(t)(D)$$

$$\mathbb{S}\llbracket\, {}^{[\ell_1]}\textbf{while } {}^{[\ell_2]}\, e \textbf{ do } {}^{[\ell_3]}\, s \textbf{ done}^{[\ell_4]} \,\rrbracket(t)(D) \stackrel{\text{def}}{=}$$
$$\qquad \mathbb{S}\llbracket\, {}^{[\ell_2]} \neg e?^{[\ell_4]} \,\rrbracket(t)(\mathrm{lfp}\, \lambda(D'.\, D \,\dot\cup\, \mathbb{S}\llbracket\, {}^{[\ell_3]} s^{[\ell_2]} \,\rrbracket(t)(\mathbb{S}\llbracket\, {}^{[\ell_2]} e?^{[\ell_3]} \,\rrbracket(t)(D'))$$

$$\mathbb{S}\llbracket\, {}^{[\ell_1]} s^{[\ell_2]} \,\rrbracket(t)(T, I) \stackrel{\text{def}}{=}$$
$$\begin{aligned}
&\text{let } T' = \mathit{apply}(T, t, I) \text{ in} \\
&\text{let } T'' = \{\, \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \overset{t:s}{\rightarrowtail} \sigma_{n+1} \mid \qquad\qquad \text{in} \\
&\qquad\qquad \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \in T' \wedge \mathit{enbl}(t, \sigma_n) \wedge \sigma_n \xrightarrow{t:s} \sigma_{n+1} \,\} \\
&(T'',\, I \cup T'')
\end{aligned}$$

when $s \in \mathit{atomic}$ (2)

$$\mathit{apply}(T, t, I) \stackrel{\text{def}}{=}$$
$$\begin{aligned}
&\mathrm{lfp}\, \lambda X.\, T \cup \{\, \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \overset{t':s}{\rightarrowtail} \sigma_{n+1} \mid \\
&\qquad\qquad \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \in X \wedge t' \neq t \wedge \langle \sigma_n, (t', s), \sigma_{n+1} \rangle \in \alpha_{\rightarrowtail}(I) \,\}
\end{aligned}$$

Fig. 11. Concrete thread-modular denotational semantics.

path-sensitivity. Although our implementation (Sec. 6) does employ trace partitioning, we will present a more classic flow-sensitive but path-insensitive thread analysis in Sec. 4. In this article, our motivation for keeping traces is rather to allow history-sensitive abstractions of interference. For instance, we will be able to distinguish, within critical sections, the last update of a variable from the previous updates, as only the last one can be actually seen by other threads.

3.2 Denotational Form

Equations (8)–(10) present a very generic formulation of a thread-modular semantics independent from the programming language, based on a transition relation τ and a scheduler enabling functions *enbl*. We now specialize the semantics to our language, using the transition relation defined in Fig. 10. The resulting semantics is presented in Fig. 11. We use a denotational form, well-suited to a definition by induction on the syntax: it mimics the way the transition system was defined (Fig. 10). This form is also well-suited to derive, by abstraction, static analyzers defined by induction on the program syntax (such as Astrée [6]) as we will see in the next section.

Our semantics is a function $\mathbb{S}\llbracket\, {}^{[\ell_1]}\, \mathit{stat}\, {}^{[\ell_2]} \,\rrbracket(t)(T, I)$ that, given a statement *stat* in a thread t and a set T of traces ending at control point ℓ_1, returns a set of traces ending at control point ℓ_2. Moreover, the function takes as argument an interference set I and returns this set enriched with the interference generated specifically by the statement. To facilitate further abstractions, an interference is also modeled as a full trace, and not simply a transition: this will allow the use of history-sensitive abstractions of interference. The domain of $\mathbb{S}[\![\cdot]\!](t)$, for each thread t, is thus $\mathcal{D} \to \mathcal{D}$, where $\mathcal{D} \overset{\text{def}}{=} \mathcal{P}(\Sigma\mathcal{A}^*) \times \mathcal{P}(\Sigma\mathcal{A}^*)$. We denote as $\dot{\cup}$ the natural join in $\mathcal{D}$, which joins sets pairwise. For compound constructs, such as the sequences, loops, and tests, the semantics is given by induction on the syntax, in a standard way. For all other statements: assignments, guards, **lock**, **unlock**, **yield**, **setpriority**, which are atomic (2), we reuse the transition relation → from Fig. 10 to simplify the presentation. The semantics of an atomic statement starts by applying an arbitrary number of transitions from the interference I (*apply*), then it selects the traces ending in a state where the thread is not blocked (*enbl*), and finally it perform a transition according to the transition relation associated with the statement. This formulation is very close to a classic denotational semantics of sequential programs, up to the gathering and application of interference. In the concrete, the interference simply appears as an accumulating version of the trace semantics, i.e., we add traces to I, while we replace traces from T with longer traces T'' performing an additional step of thread t. Given a thread $\mathit{thread}_t = {}^{[\ell_t]}\, \mathit{stat}_t\, {}^{[\ell'_t]}$ and an interference I, its denotational semantics is:

$$\mathbb{T}(t, I) \overset{\text{def}}{=} \mathbb{S}[\![\, {}^{[\ell_t]}\, \mathit{stat}_t\, {}^{[\ell'_t]}]\!]\,(t)(E, I) \enspace .$$

The semantics of the whole program is then the limit of the following iteration which computes, at each step, and for each thread t, a set of traces $T_n(t)$ and a set of interference $I_n(t)$ from the denotational semantics of t:

$$\begin{array}{l} \forall t \in \mathcal{T} :\ (T_0(t), I_0(t)) \overset{\text{def}}{=} (\emptyset, \emptyset) \\ \forall t \in \mathcal{T} :\ (T_{n+1}(t), I_{n+1}(t)) \overset{\text{def}}{=} \mathbb{T}\,(t, \cup_{t' \in \mathcal{T}}\, I_n(t')) \enspace . \end{array} \tag{11}$$

Although it has a different expression, it computes a similar trace semantics as the iteration from (10) and, thus, as $\mathcal{F}$ (6). More precisely, $\cup_{n \in \mathbb{N}}\, T_n(t)$ gives the set of program traces, but restricted to the traces where thread t reaches its final control state with the last transition of the trace:

Theorem 3.2 $\cup_{n\in\mathbb{N}}\, T_n(t) = \{\, \sigma_0 \overset{t_0:a_0}{\rightarrowtail} \cdots \overset{t_{n-1}:a_{n-1}}{\rightarrowtail} \sigma_n \in \mathcal{F} \mid t_{n-1} = t \wedge c_n(t) = \ell'_t \,\}$ *where* $\sigma_n = (c_n, \rho_n, o_n, \pi_n)$ *and thread* t *is* ${}^{[\ell_t]}\, \mathit{stat}_t\, {}^{[\ell'_t]}$*.*

Proof. We only provide a simple proof sketch. A first step would be to note that, for every thread t and every iteration of (11), $\mathbb{T}(t, I)$ returns the traces of $\mathcal{F}_M(t, \alpha_{\rightarrowtail}(I))$ restricted to traces that end with the termination of thread t. This is done by induction on the syntax of threads, and by observing that the inductive definition of $\mathbb{S}[\![\cdot]\!]$ (Fig. 11) has the same structure as the inductive definition of the transition system $\tau[\mathit{prog}]$ (Fig. 10). Then, Thm. 3.1 concludes. □

This is our final concrete semantics, as it is both thread-modular and defined by induction on the syntax of threads. It is complete for safety properties, as it takes all program traces into account, but remains uncomputable.

4 Abstract Semantics

We now show how we can abstract the thread-modular concrete denotational semantics from the last section into a static analyzer. The concrete semantics manipulates information about state and about interference, and we will be able to choose different abstractions for each kind of information. This allows us, in particular, to perform intra-thread analyses using precise, flow-sensitive, relational abstractions, focusing on the thread's local view of the program state at the current control location, as we would do for a sequential program analysis, while abstracting inter-thread interference more aggressively, as we must maintain information summarizing the whole execution of the program. In our method, interference are abstracted in a non-relational and mostly flow-insensitive way, only keeping some relations between variable values and the scheduler state (such as which mutexes are locked) and maintaining a small amount of history (such as which interference occurs last in a critical section).

In this section, we first discuss state and interference abstractions, and then provide the abstract transfer functions of the analysis, parameterized by value abstract domains. The key difference with a sequential analysis is the process to extract interference from an analysis and inject it into transfer functions.

4.1 State Abstraction

Recall that a program state $(c, \rho, o, \pi) \in \Sigma$ contains a control part $c \in \mathcal{C} \stackrel{\text{def}}{=} \mathcal{T} \to \mathcal{L}$, a memory part $\rho \in \mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{V}$, and scheduler specific information about mutex ownership $o \in \mathcal{O} \stackrel{\text{def}}{=} \mathcal{M} \to \mathcal{T}_\perp$ and thread priority $\pi \in \Pi \stackrel{\text{def}}{=} \mathcal{T} \to \mathbb{N}$. When designing a state abstraction to use to analyze one thread, our goal is threefold: firstly, leverage existing domains (such as numeric domains, pointer abstractions, etc.) for the memory state; secondly, keep precise information about control and scheduling for the analyzed thread; finally, abstract away information about other threads that we cannot hope to infer by looking only at the current thread. This is achieved by partitioning memory states with respect to an abstraction of the control and scheduler information, and using some existing value abstract domain to abstract sets of memory states in each partition. More formally, the partitioning domain, $\mathcal{P}_S$, and the corresponding abstraction, β_S, are defined as:

$$\begin{aligned} &\mathcal{P}_S \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{M}) \times \mathbb{N} \\ &\beta_S : \mathcal{T} \to \Sigma \to \mathcal{P}_S \\ &\beta_S(t)(c, \rho, o, \pi) \stackrel{\text{def}}{=} (\{\, m \in \mathcal{M} \mid o(m) = t \,\}, \pi(t)) \ . \end{aligned} \tag{12}$$

The abstraction remembers the set of locks the current thread owns, in $\mathcal{P}(\mathcal{M})$, as well as its current priority, in $\mathbb{N}$. We remember just enough information to handle precisely **lock**, **unlock**, and **setpriority** instructions. The actual priority $\mathit{tprio}(t)(\sigma)$ of thread t in state σ is completely specified given $\beta_S(t)(\sigma)$, hence, we silently overload tprio (4) to a function in $\mathcal{P}_S \to \mathbb{N}$.

Let us denote as $\mathcal{E}^\sharp$ a value abstract domain (e.g., intervals, polyhedra, etc.), with concretization: $\gamma_\mathcal{E} : \mathcal{E}^\sharp \to \mathcal{P}(\mathcal{E})$ and featuring the classic abstract transfer functions $\mathbb{S}^\sharp_\mathcal{E}[\![\cdot]\!]$ for assignments and guards, a join $\cup^\sharp_\mathcal{E}$, and widening. The partitioned state domain $\Sigma^\sharp$ and concretization γ_S are then defined as:

$$\begin{aligned}
&\Sigma^\sharp \overset{\text{def}}{=} \mathcal{P}_S \to \mathcal{E}^\sharp \\
&\gamma_S : \Sigma^\sharp \to \mathcal{P}(\Sigma) \\
&\gamma_S(S^\sharp) \overset{\text{def}}{=} \{\, (c, \rho, o, \pi) \mid \rho \in \gamma_\mathcal{E}(S^\sharp(\beta_S(c, \rho, o, \pi))) \,\} \ .
\end{aligned} \tag{13}$$

Note that our domain does not retain any control information. One benefit of employing a denotational-style semantics (e.g., Fig. 11) is that the control location $c(t)$ of the current thread is implicit. We deliberately abstract away the control location of the other threads to avoid the state explosion problem related to large control spaces in concurrent programs. Moreover, we do not keep information about the history of computation, keeping only the current program state, i.e., we abstract traces into states using $\alpha_\mathcal{R}$ (3).

4.2 Interference Abstraction

The abstract interference should collect all the effects of the current thread that can be seen by other threads. As a first approximation, we can simply collect every value written into every variable, and inject them into the other threads, whatever the program point and scheduler state. However, more precision can be obtained by removing spurious interference that cannot occur due to priority and mutual exclusion. This is achieved by partitioning interference abstractions, and applying to each abstract state partition only interference from compatible partitions. This leads to an abstraction of the form $\mathcal{P}_I \to \mathcal{P}(\mathbb{V})$, for some partitioning domain $\mathcal{P}_I$. To achieve a computable analysis, interference are further abstracted as $\mathcal{I}^\sharp \overset{\text{def}}{=} \mathcal{P}_I \to \mathbb{V}^\sharp$, where $\mathbb{V}^\sharp$ is an abstract domain representing sets of values, with concretization $\gamma_\mathbb{V} : \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V})$ (e.g., intervals for numeric variables). In our case, we define $\mathcal{P}_I$ as:

$$\mathcal{P}_I \overset{\text{def}}{=} \mathcal{T} \times \mathcal{P}_S \times \mathcal{V} \times \{\, \mathit{weak}, \mathit{yield}, \mathit{lock}(m) \mid m \in \mathcal{M} \,\} \ . \tag{14}$$

Informally, we distinguish a variable write according to the thread that performs it, the abstract scheduler state (in $\mathcal{P}_S$) the thread is in, the modified variable, and some partition information in $\{\, \mathit{weak}, \mathit{yield}, \mathit{lock}(m) \mid m \in \mathcal{M} \,\}$ that helps distinguishing the point in other threads where the interference is visible. We will illustrate the meaning of each partition in the following. An abstract interference $I^\sharp \in \mathcal{I}^\sharp$ corresponds to a set of concrete interference, i.e., a set of traces, through a concretization function γ_I, defined in Fig. 12. Each partition in $\mathcal{P}_I$ is associated to an abstract value in $\mathbb{V}^\sharp$ indicating the set of allowed written values for the traces

$$\underline{\gamma_I : \mathcal{I}^\sharp \to \mathcal{P}(\Sigma\mathcal{A}^*)}$$

$$\gamma_I(I^\sharp) \stackrel{\text{def}}{=} \cap \; \{\, \delta_I(p, \gamma_{\mathbb{V}}(I^\sharp(p))) \mid p \in \mathcal{P}_I \,\}$$

$$\underline{\delta_I : \mathcal{P}_I \times \mathcal{P}(\mathbb{V}) \to \mathcal{P}(\Sigma\mathcal{A}^*)}$$

$$\begin{aligned}
&\delta_I((t,p,V,\mathit{weak}),R) \stackrel{\text{def}}{=} \\
&\quad \{\, \sigma_0 \overset{t_0:a_0}{\rightarrowtail} \cdots \sigma_n \mid \forall i < n : \\
&\qquad (\beta_S(t)(\sigma_i) = p \wedge t_i = t \wedge a_i = (V \leftarrow e)) \Rightarrow \rho_{i+1}(V) \in R \,\} \\
&\delta_I((t,p,V,\mathit{yield}),R) \stackrel{\text{def}}{=} \\
&\quad \{\, \sigma_0 \overset{t_0:a_0}{\rightarrowtail} \cdots \sigma_n \mid \forall i < n : \\
&\qquad (\beta_S(t)(\sigma_i) = p \wedge t_i = t \wedge a_i = (V \leftarrow e) \wedge \\
&\qquad \exists j > i : \, t_j = t \wedge a_j \in \mathit{release} \;\wedge \\
&\qquad \forall k \in [i, j-1] : \, t_k \neq t \vee (a_k \notin \mathit{release} \wedge a_k \neq (V \leftarrow e'))) \Rightarrow \rho_{i+1}(V) \in R \,\} \\
&\delta_I((t,p,V,\mathit{lock}(m)),R) \stackrel{\text{def}}{=} \\
&\quad \{\, \sigma_0 \overset{t_0:a_0}{\rightarrowtail} \cdots \sigma_n \mid \forall i < n : \\
&\qquad (\beta_S(t)(\sigma_i) = p \wedge t_i = t \wedge a_i = (V \leftarrow e) \wedge \\
&\qquad \exists j > i : \, t_j = t, \, a_j = \mathbf{unlock}(m) \wedge \\
&\qquad \forall k \in [i, j-1] : \, t_k \neq t \vee (a_k \neq \mathbf{unlock}(m) \wedge a_k \neq (V \leftarrow e'))) \Rightarrow \rho_{i+1}(V) \in R \,\}
\end{aligned}$$

where $\sigma_{i+1} = (c_{i+1}, \rho_{i+1}, o_{i+1}, \pi_{i+1})$

$$\mathit{release} \stackrel{\text{def}}{=} \{\, \mathbf{yield}, \, \mathbf{lock}(m), \, \mathbf{unlock}(m), \, \mathbf{setpriority}(p) \mid m \in \mathcal{M}, \, p \in \mathbb{N} \,\}$$

Fig. 12. Concretization for the interference abstraction.

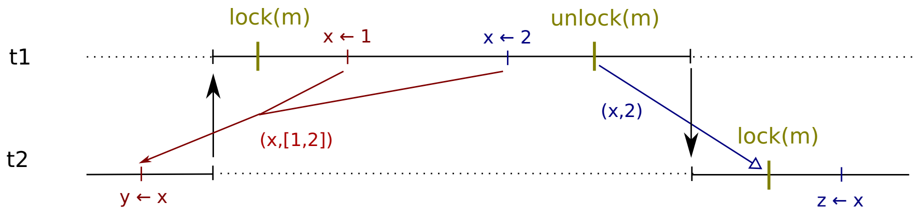

Fig. 13. Interference propagation based on locks.

in the partition. Hence, γ_I is defined as the conjunction over all partitions of the auxiliary function δ_I, that gives the semantics of a single partition. This function is also defined formally in Fig. 12 and informally below.

Locks. In order to handle mutual exclusion enforced by locks, we reuse the partitioning technique from [24], but recast it as a history-sensitive abstract partitioning of our trace-based concrete semantics (Sec. 3.2).

Consider the example in Fig. 13, where thread $t1$ stores 1 and then 2 into x while holding mutex m. In the example, the control flows freely between $t1$ and $t2$ (as shown by the vertical arrows) and is only restricted by the fact that $t1$ and $t2$ cannot both hold m at the same time. When thread $t2$ performs the assignment $z \leftarrow x$ while holding m, only the second value, 2 is visible, as the value 1 has been necessarily overwritten with 2 by $t1$ before releasing the lock. More generally, only the last write into each variable while holding a lock is visible to other threads protecting the access to the variable with the same lock. However, when $t2$ executes $y \leftarrow x$ while not holding m, it can read all the values written by $t1$ even when $t1$ holds the mutex: 1 or 2.[5]

We distinguish these two kinds of interference in $\mathcal{P}_I$ by storing them into two different partitions, namely *weak* and *lock*(m):

(i) an abstract value $[1, 2]$ associated with $(t1, p, V, \textit{weak}) \in \mathcal{P}_I$ corresponds to traces where thread $t1$ can only store 1 or 2 into variable $V \in \mathcal{V}$ while in state $p \in \mathcal{P}_S$;

(ii) an abstract value $\{2\}$ associated with $(t1, p, V, \textit{lock}(m)) \in \mathcal{P}_I$ corresponds to traces where thread $t1$ can only store 2 into variable $V \in \mathcal{V}$ while in state $p \in \mathcal{P}_I$ and holding $m \in \mathcal{M}$, when the trace contains an **unlock**(m) instruction and $t1$ performs no further modification of V before the next **unlock**(m) instruction (although another thread might modify V).

These partitions are formalized through the function δ_I in Fig. 12. In first approximation, any interference in the *weak* partition from t should be visible at any point of another thread t' where t and t' have not both locked the same mutex. Additionally, any interference on V in the *lock*(m) partition from t should be imported at the first reads from V following a **lock**(m) instruction in t' before t' overrides V. For convenience, in the second case, it is easier to consider a flow of information from the **unlock**(m) instruction in t to the **lock**(m) instruction in t'. The interference generated by the *weak* and *lock*(m) partitions are materialized, respectively, by filled red and hollow blue arrows in Fig. 13.

The flow-sensitivity is local to each mutex-protected critical section: it is limited to determining which interference occurs last inside each critical section, but it does not keep any information on the respective order of the critical sections within the program traces (e.g., in Fig. 13, the abstraction does not specify whether the assignment $x \leftarrow 2$ in $t1$ occurs before or after the assignment $z \leftarrow x$ in $t2$).

Priorities and Rescheduling. We now refine the effect of the *weak* partition by taking priorities into account. Consider first the simpler case where the priorities are fixed. In this case, a thread can only run when all higher priority threads are executing a **yield**. Figure 14 shows, from the point of view of the current thread, the effect of threads of higher and lower priority. In these examples, the control of

[5] For convenience, Fig. 13 shows the writes into x by $t1$ after the instruction $y \leftarrow x$ by $t2$, but the writes could occur earlier in a slightly different interleaving. Our abstraction is mostly flow-insensitive and will not distinguish the two cases, hence, we state that $t2$ can read the values 1 and 2.

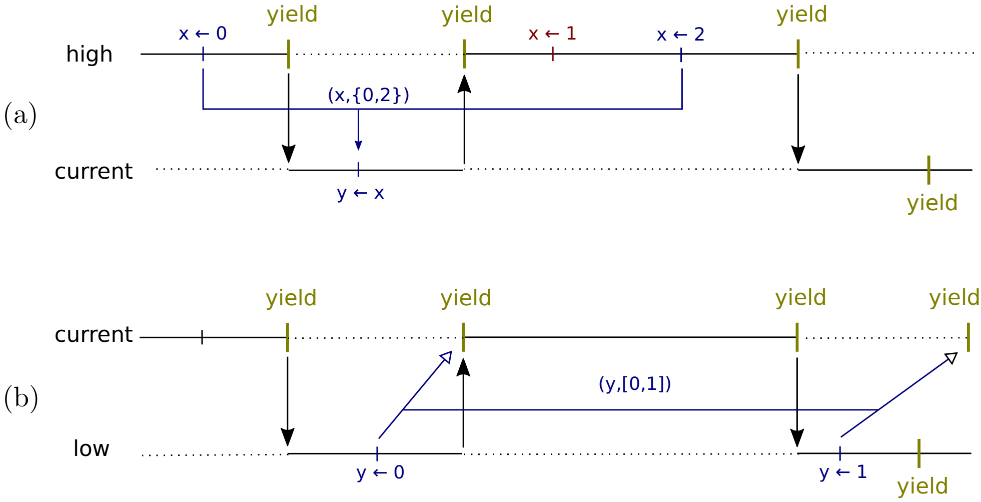

Fig. 14. Interference propagation based on priorities.

the program switches from higher priority threads to lower priority ones at **yield** instructions, and back to the higher priority threads at a non-deterministic time.

As shown in Fig. 14.(a), the current thread *current* can only see the values $\{0,2\}$ written into x by a higher priority thread *high*: these are the last values written before the thread issues a **yield** instruction; other values, such as 1, that are overwritten by another write performed before a **yield** are invisible, as *current* cannot preempt *high* between two **yield**. However, the effect of writing $\{0,2\}$ can be seen at any point in *current*, as *high* can resume its execution and perform this effect at any point.

On the other hand, as shown in Fig. 14.(b), the current thread *current* can interrupt a lower priority thread *low* at any point, and so, observe any value written into y, but this effect can only occur when *current* performs a **yield** operation, not in-between **yield**.

Although this example focuses on the **yield** instruction, the effect would be the same for any instruction that may cause a thread to give control to a lower priority thread. This includes **yield**, but also **lock**(m), **unlock**(m), and **setpriority**(p). We call these instructions *release points.* We thus refine the partitioning as follows:

(i) the *weak* partition keeps gathering all the writes to all the variables at any point in the trace, but they now only affect higher-priority threads when they reach a release point (e.g., $[0,1]$ for y in Fig. 14.(b));

(ii) a new *yield* partition gathers only the last write to each variable before the thread reaches a release point; this interference can affect a lower priority thread at any point of its execution (e.g., $\{0,2\}$ in Fig. 14.(a)).

The precise concretization of the *yield* partitions is shown formally in Fig. 12. Note that considering spurious release points is sound, as it can only enlarge the interference sets associated to the *yield* partitions.

Dynamic Priorities. The priority of a thread t can change as a result of executing a **setpriority**(p) instruction or as a result of a **lock**(m) or **unlock**(m) instruction, through the priority ceiling protocol. The semantics we described above is fully compatible with dynamic priorities, thanks to two principles:

- We partition both the state and the interference with respect to the scheduler abstraction $\mathcal{P}_S$. Hence, each abstract interference and each abstract state can be associated with a precise priority in $\mathbb{N}$, and we can determine which interference influences which state with a simple numeric comparison.
- A change of priority is seen as a release point. This gives the analysis the opportunity to import the interference from the *weak* partition of all the threads that now have a lower priority than the new priority of the current thread.

4.3 Static Analysis

Now that we are given an abstract domain for states $\Sigma^\sharp$ and for interference $\mathcal{I}^\sharp$, we are ready to state our computable abstract semantics in denotational form: $\mathbb{S}^\sharp[\![\ \mathit{stat}\ ]\!]$. It is described in Figs. 15–16 and explained below.

Abstract Domain. Naturally, the semantics operates in an abstract domain $\mathcal{D}^\sharp$ that has a component in $\Sigma^\sharp$ and a component in $\mathcal{I}^\sharp$. However, we need to compute abstract interference in a flow-sensitive way for the *yield* partitions, as we must only take into account the last assignment before each release point. Hence, we keep two abstract interference copies in $\mathcal{D}^\sharp$:

- one copy $J^\sharp \in \mathcal{I}^\sharp$ contains the last update performed on each variable;
- another copy $I^\sharp \in \mathcal{I}^\sharp$ collects the join over all release points of the last update before that release point.

Any new interference overwrites the current interference in $J^\sharp$ with a strong update while, at release points, the current value of $J^\sharp$ is accumulated, using a join, into $I^\sharp$ while $J^\sharp$ is reset to $\bot_{\mathbb{V}^\sharp}$. Hence, $J^\sharp$ is only used locally and, at the end of the analysis, the interference information we seek can be found in $I^\sharp$. The situation is similar for *lock*(m) partitions, which are also handled in a flow-sensitive way. The *weak* partitions, however, are flow-insensitive, so, any new interference is directly accumulated, with a join, into $I^\sharp$. We thus use the following abstract domain:

$$\mathcal{D}^\sharp \overset{\text{def}}{=} \Sigma^\sharp \times (\mathcal{I}^\sharp \times \mathcal{I}^\sharp)\ .$$

Composed Instructions. The abstract semantics of loops, sequences, and tests in Fig. 15 is classic, by induction on the syntax. The only significant difference with the concrete semantics is the use of a widening $\triangledown$ [12] to approximate in finite time the fixpoint defining the semantics of loops. In the formula, $\cup^\sharp$ and $\triangledown$ denote the element-wise versions of the join and widening available on abstract memory states $\mathcal{E}^\sharp$ and abstract values $\mathbb{V}^\sharp$ (i.e., they are applied independently on each partition).

Assignments and Guards. The abstract semantics of assignments and guards, in Fig. 15, is more involved. For the abstract memory component $S^\sharp \in \Sigma^\sharp$, in both

$$\underline{\mathbb{S}^\sharp[\![\,^{[\ell_1]}\,\mathit{stat}\,^{[\ell_2]}\,]\!] : \mathcal{T} \to \mathcal{D}^\sharp \to \mathcal{D}^\sharp}$$

$$\mathbb{S}^\sharp[\![\,^{[\ell_1]}s;^{[\ell_2]}s'^{[\ell_3]}\,]\!](t) \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![\,^{[\ell_2]}s'^{[\ell_3]}\,]\!]\,(t) \circ \mathbb{S}^\sharp[\![\,^{[\ell_1]}s^{[\ell_2]}\,]\!](t)$$

$$\mathbb{S}^\sharp[\![\,^{[\ell_1]}\textbf{if } e \textbf{ then } ^{[\ell_2]}\ s\ \textbf{endif}^{[\ell_3]}\,]\!](t)(D^\sharp) \stackrel{\text{def}}{=}$$

$$\quad \mathbb{S}^\sharp[\![\,^{[\ell_2]}s^{[\ell_3]}\,]\!](t)(\mathbb{S}^\sharp[\![\,^{[\ell_1]}e?^{[\ell_2]}\,]\!](t)(D^\sharp)) \cup^\sharp \mathbb{S}^\sharp[\![\,^{[\ell_1]}\neg e?^{[\ell_3]}\,]\!](t)(D^\sharp)$$

$$\mathbb{S}^\sharp[\![\,^{[\ell_1]}\textbf{while } ^{[\ell_2]}\ e\ \textbf{do } ^{[\ell_3]}\ s\ \textbf{done}^{[\ell_4]}\,]\!](t)(D^\sharp) \stackrel{\text{def}}{=}$$

$$\quad \text{let } F^\sharp(D^{\sharp\prime}) = D^{\sharp\prime} \mathbin{\triangledown} (D^\sharp \cup^\sharp \mathbb{S}^\sharp[\![\,^{[\ell_3]}s^{[\ell_2]}\,]\!](t)(\mathbb{S}^\sharp[\![\,^{[\ell_2]}e?^{[\ell_3]}\,]\!](t)(D^{\sharp\prime}))) \text{ in}$$

$$\quad \mathbb{S}^\sharp[\![\,^{[\ell_2]}\neg e?^{[\ell_4]}\,]\!](t)(\lim F^\sharp)$$

$$\mathbb{S}^\sharp[\![\,^{[\ell_2]}e?^{[\ell_3]}\,]\!](t)(S^\sharp, (I^\sharp, J^\sharp)) \stackrel{\text{def}}{=}$$

$$\quad (\lambda p \in \mathcal{P}_S.\ \mathbb{S}^\sharp_{\mathcal{E}}[\![\,\mathit{import}(I^\sharp, t, p, e)?\,]\!]\ (S^\sharp(p))),$$

$$\quad (I^\sharp, J^\sharp)$$

$$\mathbb{S}^\sharp[\![\,^{[\ell_2]}V \leftarrow e^{[\ell_3]}\,]\!](t)(S^\sharp, (I^\sharp, J^\sharp)) \stackrel{\text{def}}{=}$$

$$\quad (\lambda p \in \mathcal{P}_S.\ \mathbb{S}^\sharp_{\mathcal{E}}[\![\,V \leftarrow \mathit{import}(I^\sharp, t, p, e)\,]\!]\ (S^\sharp(p))),$$

$$\quad (\lambda(t', p, V', q) \in \mathcal{P}_I.\ \begin{cases} I^\sharp(t, p, V, q) \cup^\sharp_{\mathbb{V}} \mathbb{E}^\sharp_{\mathcal{E}}[\![\,e\,]\!](S^\sharp(p)) & \text{if } t = t' \wedge V = V' \wedge \\ & S^\sharp(p) \neq \bot^\sharp_{\mathcal{E}} \wedge q = \mathit{weak} \\ I^\sharp(t', p, V', q) & \text{otherwise,} \end{cases}$$

$$\quad \lambda(t', p, V', q) \in \mathcal{P}_I.\ \begin{cases} \mathbb{E}^\sharp_{\mathcal{E}}[\![\,e\,]\!](S^\sharp(p)) & \text{if } t = t' \wedge V = V' \wedge S^\sharp(p) \neq \bot^\sharp_{\mathcal{E}} \wedge \\ & (q = \mathit{yield} \vee (q = \mathit{lock}(m) \wedge m \in O)) \\ & \text{where } p = (O, \pi) \\ J^\sharp(t', p, V', q) & \text{otherwise)} \end{cases}$$

$$\mathit{import}(I^\sharp, t, p, e) \stackrel{\text{def}}{=}$$

$$\quad \text{let } i^\sharp = \lambda V \in \mathcal{V}.\ \cup^\sharp_{\mathbb{V}} \{\, I^\sharp(t', p', V, \mathit{yield}) \mid \mathit{itf}(t, t', p, p') \,\} \text{ in}$$

$$\quad e[\forall V \in \mathcal{V} :\ V \mapsto V \cup i^\sharp(V)]$$

$$\mathit{itf}(t, t', p, p') \stackrel{\text{def}}{\iff} (t \neq t') \wedge (O \cap O' = \emptyset) \wedge (\mathit{tprio}(t)(p) \leq \mathit{tprio}(t')(p'))$$

$$\quad \text{where } (O, \pi) = p,\ (O', \pi') = p'$$

Fig. 15. Abstract thread-modular denotational semantics.

cases, we reduce the transfer function $\mathbb{S}^\sharp[\![\,\cdot\,]\!]$ to the corresponding assignment or guard $\mathbb{S}^\sharp_{\mathcal{E}}[\![\,\cdot\,]\!]$ applied in the memory abstract domain $\mathcal{E}^\sharp$ point-wise on each partition in $\mathcal{P}_S$. However, in order to take into account the interference from the *yield* partitions, which can occur at any point in the trace, we modify the expression using the auxiliary function *import*. This function physically modifies the expression e in the assignment or guard by changing every occurrence of every variable V with a non-deterministic choice between V and the abstract interference gathered from

$$
\begin{aligned}
&\mathbb{S}^\sharp[\![\, {}^{[\ell_1]}\mathbf{yield}^{[\ell_2]} \,]\!](t)(S^\sharp, (I^\sharp, J^\sharp)) \stackrel{\text{def}}{=}\\
&\qquad \mathit{apply}(S^\sharp, t, I^\sharp, \{\mathit{yield}\}),\\
&\qquad \mathit{shift}(I^\sharp, J^\sharp, t, \{\mathit{yield}\})\\
&\mathbb{S}^\sharp[\![\, {}^{[\ell_1]}\mathbf{lock}(m)^{[\ell_2]} \,]\!](t)(S^\sharp, (I^\sharp, J^\sharp)) \stackrel{\text{def}}{=}\\
&\qquad \text{let } S^{\sharp\prime} = \mathit{apply}(S^\sharp, t, I^\sharp, \{\mathit{yield}, \mathit{lock}(m)\}) \text{ in}\\
&\qquad \mathit{map}(S^{\sharp\prime}, \lambda(M, \pi) \in \mathcal{P}_S.\, (M \cup \{m\}, \pi)),\\
&\qquad \mathit{shift}(I^\sharp, J^\sharp, t, \{\mathit{yield}\})\\
&\mathbb{S}^\sharp[\![\, {}^{[\ell_1]}\mathbf{unlock}(m)^{[\ell_2]} \,]\!](t)(S^\sharp, (I^\sharp, J^\sharp)) \stackrel{\text{def}}{=}\\
&\qquad \text{let } S^{\sharp\prime} = \mathit{map}(S^{\sharp\prime}, \lambda(M, \pi) \in \mathcal{P}_S.\, (M \setminus \{m\}, \pi)) \text{ in}\\
&\qquad \mathit{apply}(S^{\sharp\prime}, t, I^\sharp, \{\mathit{yield}\}),\\
&\qquad \mathit{shift}(I^\sharp, J^\sharp, t, \{\mathit{yield}, \mathit{lock}(m)\})\\
&\mathbb{S}^\sharp[\![\, {}^{[\ell_1]}\mathbf{setpriority}(p)^{[\ell_2]} \,]\!](t)(S^\sharp, (I^\sharp, J^\sharp)) \stackrel{\text{def}}{=}\\
&\qquad \text{let } S^{\sharp\prime} = \mathit{map}(S^{\sharp\prime}, \lambda(M, \pi) \in \mathcal{P}_S.\, (M, p)) \text{ in}\\
&\qquad \mathit{apply}(S^\sharp, t, I^\sharp, \{\mathit{yield}\}),\\
&\qquad \mathit{shift}(I^\sharp, J^\sharp, t, \{\mathit{yield}\})\\
&\mathit{map}(S^\sharp, f) \stackrel{\text{def}}{=} \lambda q.\, \cup^\sharp_{\mathcal{E}}\ \{\, S^\sharp(q') \mid f(q') = q \,\}\\
&\mathit{shift}(I^\sharp, J^\sharp, t, Q) \stackrel{\text{def}}{=} I^{\sharp\prime}, J^{\sharp\prime}\\
&\qquad \text{where } \forall t', p, V, q :\ I^{\sharp\prime}(t', p, V, q), J^{\sharp\prime}(t', p, V, q) =\\
&\qquad \begin{cases} I^\sharp(t', p, V, q) \cup^\sharp_{\mathcal{E}} J^\sharp(t', p, V, q),\ \bot_{\mathbb{V}^\sharp} & \text{if } t = t' \wedge q \in Q\\ I^\sharp(t', p, V, q),\ J^\sharp(t', p, V, q) & \text{otherwise}\end{cases}\\
&\mathit{apply}(S^\sharp, t, I^\sharp, Q) \stackrel{\text{def}}{=}\\
&\qquad \lambda p \in \mathcal{P}_S.\, \cup^\sharp_{\mathcal{E}}\ \{\, S^\sharp(p), \mathbb{S}^\sharp_{\mathcal{E}}[\![\, V \leftarrow I^\sharp(t', p', V, q) \,]\!] S^\sharp(p) \mid V \in \mathcal{V} \wedge q \in Q \wedge t \neq t'\\
&\qquad\qquad\qquad \wedge\, O \cap O' = \emptyset \wedge (q = \mathit{yield} \Rightarrow \mathit{tprio}(t)(p) \geq \mathit{tprio}(t')(p')) \,\}\\
&\qquad \text{where } (O, \pi) = p,\ (O', \pi') = p'
\end{aligned}
$$

Fig. 16. Abstract thread-modular denotational semantics (cont.).

the *yield* partitions from $I^\sharp$ for V. As discussed in Sec. 4.2 and illustrated in Fig. 14.(a), we select from the *yield* partitions only the interference generated by threads of greater priority that do not share a common mutex with the current state. This condition is expressed as the *itf* predicate in Fig. 15.

The assignment also updates the interference abstraction. We assume the existence of an abstract evaluation function $\mathbb{E}^\sharp_{\mathcal{E}}[\![\ expr\ ]\!] : \mathcal{E}^\sharp \to \mathbb{V}^\sharp$ able to compute a sound abstract representation in $\mathbb{V}^\sharp$ of the set of values an expression can evaluate to in an abstract memory state in $\mathcal{E}^\sharp$ (such a function is generally available in abstract domain implementations). Any write is accumulated directly into the *weak* partitions of $I^\sharp$ with a weak update. It is also stored with a strong update into the *yield* partitions of $J^\sharp$ and all the *lock*(m) partitions where m is currently locked. Note that we only consider the partitions that are live in the current state, i.e., $p \in \mathcal{P}_S$ such that $S^\sharp(p) \neq \bot^\sharp_{\mathcal{E}}$.

Concurrency Instructions. Figure 16 presents the semantics of the concurrency-specific instructions. This semantics also employs a number of auxiliary functions.

Firstly, as these instructions correspond to release points, we must apply to the abstract state the interference from the *weak* partitions of threads with lower priority, as discussed in Sec. 4.2 and illustrated Fig. 14.(b). This is performed by the *apply* function in Fig. 16, by converting the interference to assignment instructions in the memory abstract domain $\mathcal{E}^\sharp$. Similarly to the *import* function, the *apply* function takes care to only apply *weak* interference from threads of lower priority, and which do not hold a common mutex with the current thread. Similarly, the *apply* function is used to import interference from a *lock*(m) partition when encountering a **lock**(m) instruction, but this time disregarding the priorities.

Secondly, as a result of the instruction, our abstraction of the scheduler, i.e., our current priority and the set of mutexes owned, can change. As a consequence, the set of partitions in $S^\sharp$ might also change: some partitions may become empty (e.g., after a **lock**(m), there no partition $(O,p) \in \mathcal{P}_S$ where $m \notin O$) and some partitions may be merged (e.g., after a **setpriority**(p), the partition (O,p) contains the join of all previous partitions of the form (O,p')). This partition update is performed by the *map* function in Fig. 16, where the argument $f : \mathcal{P}_S \to \mathcal{P}_S$ indicates how partitions change.

Finally, as stated before, some interference that have been accumulated in the $J^\sharp$ map in a flow-sensitive way must be shifted into the corresponding $I^\sharp$ partition. This is performed by the *shift* function in Fig. 16.

Program Analysis. Figures 15–16 present an abstract semantics for each thread that takes interference as argument and output interference. Following the concrete denotational semantics (11), in order to achieve a sound analysis of the concurrent program, we must iterate the analysis of the threads until the interference stabilize. We perform such an iteration and use the widening over $\mathbb{V}^\sharp$ point-wise to stabilize interference in finite time:

$$\begin{aligned}
&I^\sharp_0 \stackrel{\text{def}}{=} \bot^\sharp_{\mathcal{I}} \\
&I^\sharp_{n+1} \stackrel{\text{def}}{=} \\
&\qquad \text{let } \forall t \in \mathcal{T} :\ S^\sharp(t), (I^\sharp(t), J^\sharp(t)) = \mathbb{S}^\sharp[\![\ ^{[\ell_t]}\ stat_t\ ^{[\ell'_t]}\ ]\!]\ (t)(E^\sharp, (I^\sharp_n, \bot^\sharp_{\mathcal{I}})) \text{ in} \\
&\qquad I^\sharp_n \mathbin{\triangledown} (\cup^\sharp_{\mathcal{I}} \{\, I^\sharp(t) \mid t \in \mathcal{T}\,\})
\end{aligned} \tag{15}$$

Each new analysis restarts at an abstraction $E^\sharp$ of the initial concrete states E. It also starts with the interference set $I_n^\sharp$ found at the last iteration, and outputs as $I_{n+1}^\sharp$ the join of the new interference $I^\sharp : \mathcal{T} \to \mathcal{I}^\sharp$ found by the analysis of each thread. As $J^\sharp$ is only used internally in the analysis, it is initialized to $\bot_{\mathcal{I}}^\sharp$ and its output value is ignored.

Soundness. The analysis is sound in that, after stabilization, $\gamma_I(I_n^\sharp)$ over approximates the interference computed by the concrete fixpoint (11). Moreover, for any thread $t \in \mathcal{T}$, $\gamma_S(S^\sharp(t))$, as computed in the last, stable iteration of (15), over-approximates the set of memory states t can be in when reaching its last program point ℓ'_t. More importantly, we know that, when computing $\mathbb{S}^\sharp[\![\ ^{[\ell_t]}\ stat_t\ ^{[\ell'_t]}\ ]\!](t)$ with a stable interference, we explore all the possible behaviors of the thread in the program. Returning to the introductory examples from Figs. 3–5, when parameterized with a simple interval abstraction for $\mathcal{E}^\sharp$ and $\mathbb{V}^\sharp$, our analysis is able to find the precise values of the variable `glob` as printed by the program. If the semantics is further instrumented to detect run-time errors (e.g., checking arithmetic overflows, division by zero, etc.), then the analysis will soundly report all possible run-time errors. Additionnaly, for more complex programs, more precision can be achieved by using a relational domain for $\mathcal{E}^\sharp$ (such as octagons), as done in Astrée (Sec. 6).

Data-Race Detection. A data-race can be defined as the possibility for two threads to access the same variable, one access at least being a write, and the accesses are not protected by a common mutex. As the set of writes from all threads is computed and the writes are associated with an information in $\mathcal{P}_S$ indicating the set of locked mutexes when the write occurs, it is straightforward to instrument the semantics to check, at every assignment or guard, whether another thread can write into the same variable. We can thus easily report all data-races. Moreover, as the interference analysis discards interference that are impossible due to real-time scheduling, our analysis is more precise and reports less spurious data-races than an analysis that would ignore priorities. For instance, we do not report spurious data-races in the introductory examples of Figs. 4–5, nor Fig. 6 (with or without using the priority ceiling protocol).

5 Deadlock Detection

Data-races in concurrent programs can be avoided by protecting shared data accesses with mutexes. However, introducing lock instructions has the potential to also introduce deadlocks, highly undesirable situations where two or more threads mutually block each other forever, as each one is waiting for a mutex owned by another one and none can advance. Because of the severity of deadlocks, the data-races reported by a static analyzer may remain uncorrected for fear of introducing deadlocks. The solution is to use the analyzer to soundly report both data-races and deadlocks. As the presence of deadlocks is a reachability property, it can be inferred, conservatively, using the information already gathered by our analysis.

Example. Figure 17 gives a simple example program with two threads `t1` and `t2`,

t1	t2
$1 :$ **lock**(a)	**lock**(a)
$2 :$ $\qquad$ **lock**(c)	**lock**(b)
$3 :$ $\qquad$ **unlock**(c)	**unlock**(a)
$4 :$ $\qquad$ **lock**(b)	$\qquad$ **lock**(a)
$5 :$ $\qquad$ **unlock**(b)	$\qquad$ **unlock**(a)
$6 :$ **unlock**(a)	**unlock**(b)

Fig. 17. Example of deadlocking program.

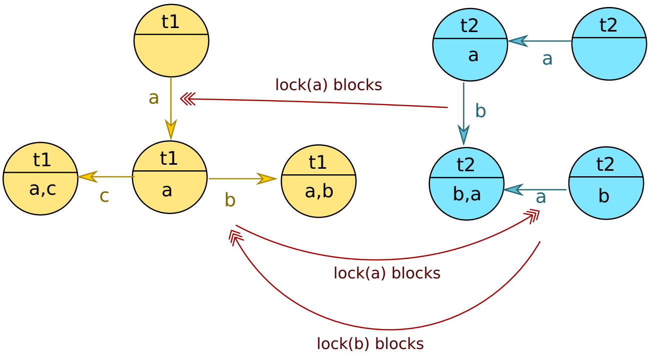

Fig. 18. Blocking graph for the program of Fig. 17 showing a deadlock cycle.

and three mutexes a, b, and c. This program can deadlock when t1, having acquired a, waits at point 4 for b, while t2, having acquired b (but not a) at point 4 is waiting for a. Neither thread can continue their execution, and the system is blocked.

Deadlock Analysis. The first step of our deadlock detection consists in collecting, during the analysis, for each thread $t \in \mathcal{T}$ and each mutex $m \in \mathcal{M}$, the **lock**(m) instructions, while remembering their context in the form of the set $O \subseteq \mathcal{M}$ of mutexes that t already owns when executing the instruction. More precisely, we construct the *configuration graph*, a labelled directed graph $(N, A) \in \mathcal{N} \times \mathcal{A}$ with nodes N in $\mathcal{N} \stackrel{\text{def}}{=} \mathcal{T} \times \mathcal{P}(M)$, and arcs A in $\mathcal{A} \stackrel{\text{def}}{=} \mathcal{N} \times \mathcal{M} \times \mathcal{N}$. Arcs are labelled with a mutex. A node $(t, O) \in \mathcal{N}$ represents a reachable configuration, and arcs have the form $((t, O), m, (t, O \cup \{m\})) \in \mathcal{A}$ to indicate that thread t attempts to lock m in configuration O, and changes to configuration $O \cup \{m\}$ upon success. After the program analysis terminates, when all the reachable configurations and lock instructions have been collected into (N, A), a second step creates a *blocking graph* $(A, C) \in \mathcal{A} \times \mathcal{C}$ that materializes that a thread can prevent another thread to lock a mutex. The nodes of the blocking graph are the collected lock instructions A from the first step, while arcs are in $\mathcal{C} \stackrel{\text{def}}{=} \mathcal{A} \times \mathcal{A}$. An arc in $(l, l') \in C \subseteq \mathcal{C}$ indicates that the lock instruction $l = ((t, O), m, (t, O \cup \{m\}))$ is executed in a configuration that prevents $l' = ((t', O'), m', (t', O' \cup \{m'\}))$ from succeeding. More precisely:

$$C \stackrel{\text{def}}{=} \{\, (((t, O), m, (t, O \cup \{m\})), ((t', O'), m', (t', O' \cup \{m'\}))) \in \mathcal{C} \mid m' \in O \,\} \ .$$

A deadlock corresponds to a *cycle* in the blocking graph, i.e., a sequence of threads that are waiting trying to lock a mutex, and each thread is prevented from effectively locking it by the next thread in the sequence. Not all cycles correspond to deadlocks. We can in particular rule out cycles where the configurations involved are not compatible, i.e., cannot possibly refer to the same program state. More precisely, a cycle $l_1 \stackrel{\text{def}}{=} ((t_1, O_1), m_1, (t_1, O_1 \cup \{m_1\}))$, ..., $l_n \stackrel{\text{def}}{=} ((t_n, O_n), m_n, (t_n, O_n \cup \{m_n\}))$ is *acceptable* only if it satisfies:

$$\forall i \neq j \in [1, n] : t_i \neq t_j \wedge O_i \cap O_j = \emptyset$$

i.e., the threads attempting the locks are distinct and no mutex is already locked by two threads.

Theorem 5.1 *The deadlock analysis is sound: every deadlock corresponds to an acceptable cycle found by the analysis.*

Proof. In Appendix A.2. □

Example Revisited. Figure 18 shows the configuration and blocking graph for the example of Fig. 17, using circles as configurations, filled arrows labelled with a mutex between configurations to denote lock instructions, and triple-lined arrows between lock instructions to denote blocks. The deadlock is visible as the cycle between the lock $((t1, \{a\}), b, (t1, \{a, b\}))$ and the lock $((t2, \{b\}), a, (t2, \{a, b\}))$. Although our example is limited to two threads, the analysis can naturally detect deadlock cycles involving more threads (unlike, for instance, [30]).

Our blocking graph is similar to the lock graph of [9] except that our graph is constructed from the output of a static analysis, and not extracted from an execution trace, hence, it reports all deadlocks.

Limitations. Our deadlock analysis is conservative and can report spurious deadlocks. This is expected as the reachability analysis it builds upon in conservative. However, an important limitation is that thread priorities are not fully exploited in the analysis to discard certain sequences of locks that would cause a deadlock but cannot actually occur. In particular, the priority ceiling protocol, which is commonly employed to avoid deadlocks, is not handled precisely in our analysis. Consider again the example of Fig. 17, but assuming now that the priority ceiling protocol is used to raise threads to a high priority upon locking either a or b. Thus, whichever thread locks a first at line 1 is guaranteed to finish its execution at line 6 without the other thread preempting it and causing a deadlock. The configuration graph of Fig. 18 cannot express that it is not possible to reach both configurations $(t1, \{a\})$ and $(t2, \{b\})$ simultaneously due to the real-time scheduling, and thus, it will report a spurious deadlock.

		no priorities			priorities		
name	lines	time	memory	alarms	time	memory	alarms
FreeOSEK-COM	210	1s	12MB	1	1s	12MB	0
HiTechnic	65	0.44s	11MB	0	0.42s	11MB	0
NXT GT	190	1.23s	17MB	12	1.33s	17MB	11
NXTway-GS	341	4.21s	20MB	8	10.47s	20MB	7
NXT Cesar	4429	7mn	380MB	182	14mn	500MB	182

Fig. 19. Blocking graph for the program of Fig. 17 showing a deadlock cycle.

6 Preliminary Experiments

The methods presented in this article have been implemented in the Astrée/AstréeA analyzer.

Sequential Astrée. Astrée [6] is a static analyzer to check for run-time errors in embedded critical sequential C software. It handles a faithful, low-level C semantics, including machine integers with wrap-around, floating-point arithmetic with rounding, pointers and pointer arithmetic, structures, arrays, union types, goto statements, etc. However, as it targets embedded software, it does not support recursion and has limited support for dynamic memory allocation. The alarms it reports include behaviors undefined according to the C semantics such as arithmetic and memory overflow, invalid operations, assertion violations, etc. One particular aspect of Astrée is its design by specialization: starting from a simple, interval-based analyzer, and a target application domain, avionics control-command safety-critical synchronous C software, Astrée was enriched with new abstract domains, combined together through a reduced product, until the analysis reported no false alarm on programs from the application domain. Astrée was then shown to be usable in avionics industrial context [14] and subsequently commercialized by AbsInt [20].

Concurrent Astrée. Subsequently, the AstréeA project aimed at extending Astrée to analyze concurrent embedded C software. It first focused on ARINC 653 applications [4], and was later extended to a fragment of Real-Time POSIX [17] and OSEK/AUTOSAR [1]. As Astrée, AstréeA has undergone a specialization process in an academic context (not quite achieving the zero false alarm mark, but raising the selectivity[6] up to 99.94%), and has been evaluated in an industrial context during research projects [27]. It was then integrated into the commercial version of Astrée [28]. Astrée for concurrent programs is based on the principles described in this article. It performs a thread-modular analysis, where threads are reanalyzed until an abstraction of their interference stabilizes, and each thread analysis is performed using a fully flow- and context-sensitive abstract interpreted by induction on the syntax over a reduced product of abstract state domains. Previous versions Astrée did not handle priorities precisely and did not report deadlocks. We report here on the improvements brought by the novel partitioning-based improvement to priority handling described in Sec. 4.

[6] The selectivity denotes the percentage of program lines without any alarm.

Experiments. In our experiments, we focused on OSEK/AUTOSAR [1] applications. Astrée is able to analyze large sequential and concurrent industrial programs of several million lines [6,27]. However, in this preliminary evaluation, we were only able to evaluate open-source demonstration programs, which are rather small. These include demonstration programs for FreeOSEK, and programs for Lego Mindstorm NXT robots under the nxtOSEK system. Figure 19 provides, for each example, its size (in lines), the analysis time, and the number of alarms (taking into account run-time errors, data-races and deadlocks), with and without our new priority-aware analysis. The experiments show a small improvement in precision. More precisely, some data-races were removed using the priority-aware analysis, and the programs are proved free of deadlock. Additionnaly, our prototype implementation has been tested through a series of 62 unit tests similar to Figures 2–6, which demonstrated the ability of our analysis to remove spurious data-races by exploiting priorities. These experiments are very preliminary. In future work, we will consider larger applications, such as the industrial AUTOSAR applications currently evaluated with the priority-unaware version of Astrée in [27].

7 Conclusion

In this article, we have reviewed the thread-modular abstract interpretation based static analysis of concurrent embedded C software performed by Astrée, and we showed how to modify it to take into account thread priorities and the specific characteristics of real-time schedulers. The method employs partitioning techniques in order to classify more finely the set of effects a thread can have on other threads depending on their respective priority and set of owned mutexes, so as to remove spurious interference. The resulting analysis targets software running on systems such as ARINC [4], OSEK/AUTOSAR [1], or real-time POSIX [17], where the thread priority can be modified dynamically. It also supports the popular priority ceiling protocol. We have shown, by experimentation with Astrée on some simple examples and OSEK demonstration programs, that this technique allows removing alarms with respect to a semantics that disregards priorities and allows all interleavings of threads.

Future Works. There are many avenues for future work. Firstly, experimental evaluation should be conducted on large, real-life applications, such as the ARINC and POSIX avionics applications from [27] and the AUTOSAR automotive applications from [28], to determine whether the increase in precision is significant and whether the analysis retains its scalability. Otherwise, new abstractions should be devised to take priority only partially into account and achieve a cost versus precision tradeoff. It would also be interesting to compare our analysis with those based on sequentialization [39] on interrupt-driven programs: our priority-aware analysis may bridge the gap between efficient but coarse priority-unaware thread-modular methods and precise, but less efficient, sequentialization-based techniques.

Secondly, in this work, we only considered non-relational abstractions for interference. In previous works [26], we considered relational abstractions for interference (such as, for instance, lock invariants), but restricted to the case of arbitrary preemption. While these two works are based on the same flavor of thread-modular concrete semantics [25], effectively merging the abstractions to achieve interference abstraction that is both priority-aware and relational remains future work.

Thirdly, we should improve the deadlock analysis in order to take priorities into account. One goal would be to prove that the correct use of the priority ceiling protocol can indeed prevent deadlocks. Additionnaly, the deadlock analysis could be extended to an analysis of priority inversions, undesirable situations where a lower priority thread hoarding a mutex prevents a higher priority thread to execute. This situation can also be addressed through the priority ceiling protocol.

A fourth avenue of future work is the development of history-sensitive abstractions for thread interference. Currently, we are able, through locks or priorities, to discover that two parts of two threads cannot interact, but we have no information about the ordering of threads. It would be useful to be able to infer that one section of a thread necessarily executes before one section of another thread. Applications include a precise analysis of any initialization process, where the initializing thread is guaranteed to execute before any thread that use the initialized data.

A fifth avenue would be to extend and generalize the real-time scheduling model. One interesting direction would be the addition of new synchronization primitives beside locks, such as events and barriers. Another direction would be the support for various flavors of multi-core real-time schedulers.

A last avenue for future work would be to study more closely the validity of our abstract semantics in the presence of weak memory models.

References

[1] AUTOSAR (AUTomotive Open System ARchitecture). http://www.autosar.org.

[2] DO-178C: Software considerations in airborne systems and equipment certification, 2011.

[3] S. V. Adve and M. D. Hill. Weak ordering – A new definition. In *Proc. of the 17th ACM SIGARCH Symp. on Comp. Arch. (ISCA'90)*, volume 18, pages 2–14. ACM, June 1990.

[4] Aeronautical Radio Inc. ARINC 653. http://www.arinc.com.

[5] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL'10*, pages 7–18. ACM, Jan. 2010.

[6] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385 in AIAA, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), Apr. 2010.

[7] H.-J. Boehm. How to miscompile programs with "benign" data races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 3–3. USENIX Association, 2011.

[8] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications (FMPA'93)*, volume 735 of *LNCS*, pages 128–141. Springer, June 1993.

[9] Y. Cai and W.K. Chan. Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Trans. Softw. Eng.*, (40):266–281, 2014.

[10] J.-L. Carré and C. Hymans. From single-thread to multithreaded: An efficient static analysis algorithm. Technical Report arXiv:0910.5833v1, EADS, Oct. 2009.

[11] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, Jan. 1977.

[13] P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, NY, USA, 1984.

[14] D. Delmas and J. Souyris. Astrée: from research to industry. In *SAS'07*, volume 4634 of *LNCS*, pages 437–451. Springer, Aug. 2007.

[15] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN'03*, volume 2648 of *LNCS*, pages 213–224. Springer, 2003.

[16] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem.* PhD thesis, University of Liege, Computer Science Department, 1994.

[17] IEEE Computer Society and The Open Group. Portable operating system interface (POSIX) – Application program interface (API) amendment 2: Threads extension (C language). Technical report, ANSI/IEEE Std. 1003.1c-1995, 1995.

[18] B. Jeannet. Relational interprocedural verification of concurrent programs. *Software & Systems Modeling*, 12(2):285–306, 2013.

[19] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference.* PhD thesis, Oxford University, Jun. 1981.

[20] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée: Proving the absence of runtime errors. In *Proc. of Embedded Real Time Software and Systems (ERTS2 2010)*, page 9, May 2010.

[21] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*, pages 115–148. Springer Berlin Heidelberg, 2005.

[22] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is Cartesian abstract interpretation. In *ICTAC'06*, volume 4281 of *LNCS*, pages 183–197, 2006.

[23] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In *ESOP'05*, volume 3444 of *LNCS*, pages 5–20. Springer, 2005.

[24] A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *Proc. of the 20th European Symp. on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 398–418. Springer, Mar. 2011.

[25] A. Miné. Static analysis by abstract interpretation of sequential and multi-thread programs. In *MOVEP'12*, pages 35–48, Dec. 2012.

[26] A. Miné. Relational thread-modular static value analysis by abstract interpretation. In *Proc. of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'14)*, volume 8318 of *LNCS*, pages 39–58. Springer, 2014.

[27] A. Miné and D. Delmas. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In *Proc. of the 15th International Conference on Embedded Software (EMSOFT'15)*, pages 65–74. IEEE CS Press, Oct. 2015.

[28] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking static analysis to the next level: Proving the absence of run-time errors and data races with Astrée. In *Proc. of Embedded Real Time Software and Systems (ERTS2 2016)*, pages 570–579, Jan 2016.

[29] D. Monniaux. Verification of device drivers and intelligent controllers: a case study. In *Proc. of the 7th ACM & IEEE International conference on Embedded software (EMSOFT'07)*, pages 30–36. ACM, Oct. 2007.

[30] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 386–396. IEEE Computer Society, 2009.

[31] A. Ouadjaout, A. Miné, N. Lasla, and N. Badache. Static analysis by abstract interpretation of functional properties of device drivers in TinyOS. *Journal of Systems and Software (JSS)*, 120:114–132, 2016.

[32] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proc. of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[33] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Proc. of the ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'04)*, pages 14–24. ACM, June 2004.

[34] J. C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In *Proc. of the Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *LNCS*, pages 35–48. Springer, Dec. 2004.

[35] M. D. Schwarz, H. Seidl, V. Vojdani, P. Lammich, and M. Müller-Olm. Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. *SIGPLAN Not.*, 46(1):93–104, January 2011.

[36] T. Suzanne and A. Miné. From array domains to abstract interpretation under store-buffer-based memory models. In *Proc. of the 23st International Static Analysis Symposium (SAS'16)*, volume 9837 of *LNCS*, pages 469–488. Springer, Sep. 2016.

[37] V. Vojdani and V. Vene. Goblint: Path-sensitive data race analysis. *Annales Univ. Sci. Budapest, Sect. Comp.*, 30, 2009.

[38] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *DASC'07*, volume 2.A.1, pages 1–10. IEEE, Oct. 2007.

[39] W. Wu, L. Chen, A. Miné, D. Dong, and J. Wang. Numerical static analysis of interrupt-driven programs via sequentialization. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(70):26, Aug. 2016.

[40] H. Zhang, T. Aoki, and Y. Chiba. A Spin-based approach for checking OSEK/VDX applications. In *Formal Techniques for Safety-Critical Systems: Third International Workshop, FTSCS 2014*, pages 239–255. Springer, 2015.

A Proof of Theorems

A.1 Proof of Thm. 3.1

Proof. Recall that $\mathcal{F}$ is defined in (6) as $\mathcal{F} \overset{\text{def}}{=} \mathrm{lfp}\, F$ where

$$F(T) \overset{\text{def}}{=} E \cup \{\, \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \overset{t:i}{\rightarrowtail} \sigma_{n+1} \mid \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \in T \wedge enbl(t, \sigma_n) \wedge \sigma_n \xrightarrow{t:i} \sigma_{n+1} \,\}$$

We wish to prove that $\mathcal{F} = \mathrm{lfp}\, G$ where:

$$G(T) \overset{\text{def}}{=} \cup_{t \in \mathcal{T}}\ \mathcal{F}_M(t, \alpha_{\rightarrowtail}(T))$$

and $\mathcal{F}_M(t, I)$ is defined in (8) as $\mathrm{lfp}\, H(t, I)$ where

$$\begin{aligned} H(t,I)(T) \overset{\text{def}}{=} E \cup \{\, \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \overset{t':i}{\rightarrowtail} \sigma_{n+1} \mid \sigma_0 \overset{a_0}{\rightarrowtail} \sigma_1 \cdots \sigma_n \in T \wedge \\ ((t = t' \wedge enbl(t, \sigma_n) \wedge \sigma_n \xrightarrow{t:i} \sigma_{n+1}) \vee \\ (t \neq t' \wedge \langle \sigma_n, (t', i), \sigma_{n+1} \rangle \in I)) \,\} \end{aligned}$$

We first note that $\alpha_{\rightarrowtail}(\mathcal{F})$ is a subset of the transition relation $\sigma \xrightarrow{t:i} \sigma'$. As a consequence, $\forall t, T : H(t, \alpha_{\rightarrowtail}(\mathcal{F}))(T) \subseteq F(T)$. As both functions $H(t, \alpha_{\rightarrowtail}(\mathcal{F}))$ and F a monotonic in complete powerset lattices, they have a least fixpoint. As

$H(t, \alpha_{\rightarrowtail}(\mathcal{F}))$ is smaller than F, any fixpoint of F is a post-fixpoint of $H(t, \alpha_{\rightarrowtail}(\mathcal{F}))$, and we have $\forall t : \text{lfp}\, H(t, \alpha_{\rightarrowtail}(\mathcal{F})) \subseteq \text{lfp}\, F = \mathcal{F}$. Hence, $\forall t : \mathcal{F}_M(t, \alpha_{\rightarrowtail}(\mathcal{F})) \subseteq \mathcal{F}$, and $G(\mathcal{F}) \subseteq \mathcal{F}$, i.e., $\mathcal{F}$ is a post-fixpoint of G. As G is also monotonic, it has a least fixpoint, which satisfies $\text{lfp}\, G \subseteq \mathcal{F}$.

To prove the converse inclusion, consider a trace $\mathfrak{t} \overset{\text{def}}{=} \sigma_0 \overset{t_0:i_0}{\rightarrowtail} \cdots \sigma_{n-1} \in \mathcal{F}$. We prove by recurrence on n that $\mathfrak{t} \in G^n(\emptyset)$. Indeed, if $n = 1$, then $\mathfrak{t} = \sigma_0 \in E$, and $E \subseteq G^1(\emptyset)$. Assume that the property is true at rank n and consider $\mathfrak{t} \overset{\text{def}}{=} \sigma_0 \overset{t_0:i_0}{\rightarrowtail} \cdots \sigma_n \in \mathcal{F}$. By induction hypothesis, $\sigma_0 \overset{t_0:i_0}{\rightarrowtail} \cdots \sigma_{n-1} \in G^n(\emptyset)$. We can see that $\mathfrak{t} \in \mathcal{F}_M(t_n, \alpha_{\rightarrowtail}(G^n(\emptyset))$ as $\mathfrak{t}$ contains only transitions that are either in $\alpha_{\rightarrowtail}(G^n(\emptyset))$, or generated by thread t_n. Hence, $\mathfrak{t} \in G^{n+1}(\emptyset)$. By Kleene's theorem, $\text{lfp}\ G = \cup_{n \in \mathbb{N}} G^n(\emptyset)$, so that $\mathfrak{t} \in \text{lfp}\ G$, which proves that $\mathcal{F} \subseteq \text{lfp}\ G$. □

A.2 Proof of Thm. 5.1

Proof. Consider a concrete reachable state $\sigma = (c, \rho, o, \pi) \in \Sigma$ that corresponds to a deadlock. We prove that we can exhibit an acceptable cycle in the blocking graph computed by the analysis.

By the definition of deadlocks, there exists a set of threads $T = \{t_1, \ldots, t_n\}$ such that every thread $t_i \in T$ is blocked on a lock instruction **lock**(m_i) while m_i is owned by another thread $o(m_i) \in T$. As T is finite, the graph $(T, \{\, (o(m_i), t_i) \mid t_i \in T\,\})$ contains a cycle. Without loss of generality, we can restrict ourselves to the case where the graph is a simple cycle: $t_1 = o(m_2)$, $\ldots$, $t_{n-1} = o(m_n)$, $t_n = o(m_1)$. Finally, let us note $O_i \overset{\text{def}}{=} \{\, m \mid o(m) = t_i \,\}$. Then, as σ is reachable, and the reachability analysis enumerating configurations is sound, for every i, the configuration (t_i, O_i) is reachable: it can be seen as an abstraction of σ. The configuration graph thus contains the arcs $l_i \overset{\text{def}}{=} ((t_i, O_i), m_i, (t_i, O_i \cup \{m_i\}))$. As $t_i = o(m_{i+1})$, we get that $m_{i+1} \in O_i$, so that $(l_i, l_{i+1}) \in C$. Thus, $l_1, \ldots, l_n$ forms a cycle in the blocking graph. As all the t_i are distinct, and the O_i (being the images of the t_i by o^{-1}) are pairwise disjoint, the cycle is acceptable. □