Appears in the proceedings of the *IEEE 17th International Conference on Application of Concurrency to System Design* (ACSD), 2017.

Lock Optimization for Hoare Monitors in Real-Time Systems

Georges-Axel Jaloyan
École normale supérieure
Computer Science Department
45 rue d'Ulm, F-75230 Paris CEDEX 05, France
Email: georges-axel.jaloyan@ens.fr

Lee Pike
Galois Inc.
421 SW 6th Avenue, Suite 300 Portland
Oregon 97204, United States
Email: leepike@galois.com

***Abstract*—Hoare monitors are a safe concurrency abstraction built around a monitor with shared state and methods that operate on the shared state. While well-known, they have been little used as a concurrency framework in real-time systems. We describe a Hoare monitor framework called *Tower* developed for real-time systems programming that targets multiple real-time operating systems. Hoare monitors use coarse-grained locking across all of the methods in a monitor. In a real-time setting, this coarse-grained locking can be too restrictive, but it is difficult and tedious for a programmer to reason about which methods may safely execute in parallel. Therefore, we present an automated compiler optimization for refining locks in Hoare monitors using partially-weighted MAXSAT. We formalize Tower semantics using Petri nets and show that safe concurrency is preserved under the optimization. Finally, we present a number of empirical benchmarks for our optimization as well as a case-study of a real-time autopilot built and optimized with our approach.**

I. Introduction

Concurrency in embedded real-time systems is often necessary to handle interrupts and deadline constraints, but it can be notoriously difficult to implement correctly. One famous example is the Mars Pathfinder concurrency bug [16].

Monitors are a programming abstraction invented in the 1970s by C. A. R. Hoare and Brinch Hasen for safe concurrency [13], [4]. A *Hoare monitor* is a construct that guarantees thread-safe accesses to shared resources. A monitor contains a set of *methods* that share resources. Only one method can execute at a time. Hoare monitors make safe concurrency easier, since by construction, if the implementation is correct, deadlocks are not possible.

But that safety comes at price: a method takes a lock that is held while a thread is executing the method, blocking all other methods in the monitor from being executed. Indeed, if the lock is global, then no other thread can execute until the lock is released.

While Hoare monitors have been implemented in a variety of programming languages, they have rarely been used in the context of embedded real-time systems and real-time operating systems; we revisit the use of Hoare monitors in this context. In particular, we have used a Hoare-monitor based programming paradigm called *Tower* to design and implement an autopilot for small unmanned air vehicles. We introduce Hoare monitors and describe our implementation of them for real-time systems in Section II. Tower has backends that target FreeRTOS [2] and eChronos (a formally verified RTOS) [7], POSIX, and the formally verified seL4 microkernel [18] with recent real-time support [20].

We propose three properties that should hold of a Hoare monitor implementation: (1) absence of dataflow cycles between methods, (2) absence of race conditions, and (3) deadlock freedom. In Section III, we develop a Petri net formalization of Tower and show the properties hold.

One benefit of Hoare monitors is that they provide a convenient programmer abstraction of the system in which the programming model is a dataflow model between methods. Methods that are conceptually related (e.g., for a device driver), belong to the same monitor, much in the same way that conceptually related functions are placed in the same module. By construction, the programmer is guaranteed that there is no out-of-band shared state between methods not in the same monitor.

While it is useful and convenient to place conceptually related methods in the same monitor, it can overly constrain the system. For example, consider three methods, m_0, m_1, and m_2, in the same monitor, where m_0 and m_1 share state and m_1 and m_2 share another state. Then m_0 and m_2 can execute in parallel, despite being in the same monitor, because they share no state.

We investigate how to automatically optimize concurrent Hoare-monitor programs in Section IV. Our approach uses a partial weighted MaxSAT (PWMS) [21] encoding of Hoare monitors to refine the number and assignment of locks with monitors: a single global lock per monitor may be replaced with multiple locks associated with subsets of methods. This is necessarily a global optimization, since on an embedded RTOS, there is generally a fixed number of total locks available, which introduces a global

constraint. We also prove in the Petri net model that under optimization, the three safety properties mentioned earlier continue to hold.

We present experimental results for our lock refinement optimization in Section V, and then describe an extended case-study of applying the optimization to an autopilot in Section VI.

Related work and conclusions are described in Sections VII and VIII, respectively.

II. Hoare Monitors

Hoare monitors are thread-safe constructs, comparable to modules, that enforce safe access to resources shared at the monitor scope using a mutex. First implemented by Hoare in Concurrent Pascal, they have since been implemented in other languages, ranging from C++11 to Python and Ruby.

A monitor is an enclosing structure that protects accesses to its shared resources (declared at the monitor scope), by defining some accessors (or procedures, or methods) that exclusively access those shared resources in a thread-safe way. Hence, declaring a monitor is done by providing some shared resources, and by defining the procedures that use those resources.

Procedures are written in a way to enforces thread-safety using a lock declared at the monitor level. All of a procedure's code is inside the locked environment, preventing unlocked access to shared resources. In this way, only one procedure in a given monitor can be executed at a given time, resulting in the absence of race conditions (all shared resources are protected by the monitor's lock). More specifically, each procedure of the monitor m is run, as defined in [13], according to the following scheme: take the monitor's lock, wait for some specific condition variable, execute the body, signal other procedures on a condition variable, release the lock.

```
// Acquire this monitor's lock.
acquire(lock_m);
// While the predicate we are waiting for is false.
while (!p) {
  // Wait on the lock_m and condition variable cv.
  wait(lock_m, cv);
}
// ... Critical section of code goes here ...
// Signal procedures on the condition variable cv2.
signal(cv2);
release(lock_m); // Release lock_m.
```

A. Tower: Hoare Monitors for Real-Time Systems

Tower is a domain-specific language for real-time Hoare-monitor based programming. The methods of Tower are called *handlers*. Channels communicate data used to signal the handlers of a monitor. There are four types of channels in Tower: *synchronous channels*, *periodic channels*, *signal channels*, and *initialization channels*. Synchronous channels have an *input* and *output* endpoint. One or more handlers listen on the output endpoint of the channel, and multiple handlers can write to the input end of a channel. A channel is first-in-first-out (FIFO) with a *depth*, which is the number of messages it can hold. The default depth is one.

The other channels have only an output endpoint; the input implicitly comes from the system. A periodic channel is declared for every periodic task rate, and the input implicitly comes from the system clock. Assuming the system is schedulable, a handler for an n microseconds periodic channel receives a notification every n microseconds. A signal channel's input comes from a system interrupt and drives interrupt service routines. Finally, an initialization channel's input comes from the scheduler and drives handlers that run once, at system initialization.

Tower automatically creates RTOS threads associated with each periodic, signal, and initialization channel. Every handler that listens to one of these kind of channels is executed in the associated thread. Handlers that listen to synchronous channels are not scheduled as threads but are library code called by scheduled threads.

In a Tower program, monitors are declared using the `monitor` keyword. Each monitor takes a string that names it as an argument, and contains a list of `handler` and `state` (for shared resources) keywords. State can optionally be initialized using the `stateInit` keyword. Each handler listens on a typed channel. A handler takes as an argument a channel, a name (i.e., a string), then a list of `callbacks`. Each callback contains the behavioral component code to execute. A callback takes a single argument, the value received over its enclosing handler's channel.

Callbacks are written in Ivory [9], a memory-safe systems language that shares a type system with Tower. Callbacks are executed in the order they are declared. In addition to performing arbitrary local computation and reading and writing the state variables within its enclosing monitor, a callback may write to one or more outbound channels. It does so by executing an `emit` command that takes a channel and a value as an argument.

As an example of a Tower program, consider Figure 1 and its graphical representation in Figure 2. The program blinks two LEDs, `led1` and `led2`. The program defines two tasks, one running at 500 milliseconds and one running at 10 milliseconds. The 500 milliseconds task drives two handlers, `flipflop` and `led1on`. The `flipflop` handler emits a Boolean on a channel and then stores the negation of the value into a monitor-scope shared resource. The `led2ctrl` handler reads the output of the channel the `flipflop` handler emitted on. It takes the Boolean passed on the channel; if it is true, then it turns `led2` on; otherwise it turns it off, by passing a state variable representing `led2` to the functions `ledOn` and `ledOff`, respectively (the definitions of the functions are elided here for space).

The second $500ms$ handler (`led1on`) turns `led1` on by calling the `ledon` function and then stores into a monitor-scope resource, `led1lit`, that `led1` is lit.

Finally, the `led1off` task runs at $10ms$ and if the

```
p500     <- period (500 ms)
p10      <- period (10 ms)
(tx, rx) <- channel

monitor "go" do
  stateInit "led2lit" false
  handler p500 "flipflop" do
    callback \_ -> do
      emit tx led2lit
      store led2lit (not led2lit)

monitor "led" do
  stateInit "led1lit" false
  state "led1"
  state "led2"
  handler p500 "led1on" do
    callback \_ -> do
      ledOn led1
      store led1lit true
  handler p10 "led1off" do
    callback \_ -> do
      if led1lit
         (ledOff led1)
      store led1lit false
  handler rx "led2ctrl" do
    callback \out -> do
      if out
        then (ledOn  led2)
        else (ledOff led2)
```

Figure 1: Tower example (with syntactic simplifications).

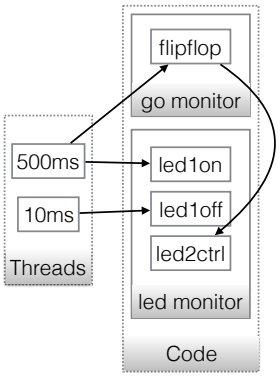


Figure 2: Graphical representation of the Tower program from Figure 1.

monitor-scope variable `led1lit` is true, then it turns `led1` off.

Tower uses Haskell [24] syntax; we have elided a few idiosyncrasies in the example in Figure 1. A few syntactic explanations are still in order: the `do` keyword introduces a sequence of instructions to be executed in order. Lambda is denoted by `\`, and a lambda expression, `\foo -> ...` denotes an anonymous function that takes `foo` as a formal parameter and is used in the function's body. If the argument is unused in the body, then the formal parameter is elided with an underscore (`_`).

B. Tower toolchain

Figure 3 shows the backend structure. Tower programs are reified and transmitted to several backends including POSIX, the FreeRTOS [2] and eChronos [7] RTOSes, and the seL4 microkernel [18]. For the eChronos and seL4 backends, glue code is generated via an intermediate *Architecture Analysis and Design Language* (AADL) [11] specification that is generated from Tower. An AADL-based tool developed by University of Minnesota generates operating system bindings from AADL.[1]

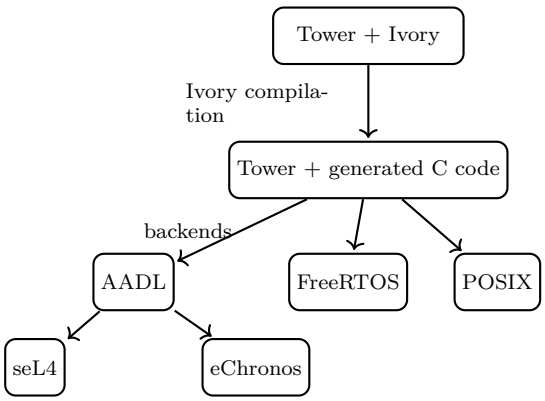


Figure 3: The Tower tool-chain.

$$
\begin{array}{lcl}
\langle monitor \rangle & ::= & \texttt{monitor}\ name\ \texttt{do}\ (\langle handler \rangle)^* \\
\langle handler \rangle & ::= & \texttt{handler}\ \langle channel \rangle\ name\ (\langle callback \rangle)^* \\
\langle channel\_id \rangle & ::= & integer \\
\langle channel \rangle & ::= & \langle channel\_id \rangle \\
 & | & \texttt{period}\ time \\
 & | & \texttt{signal}\ name\ deadline \\
 & | & \texttt{init} \\
\langle emitter \rangle & ::= & \texttt{emit}\ \langle channel\_id \rangle\ value \\
\langle callback \rangle & ::= & \texttt{callback}\ value\ \texttt{->}\ \texttt{do}\ (\langle emitter \rangle)^*
\end{array}
$$

Figure 4: Simplified Tower grammar, we abstracted out all Ivory code, except emit commands.

III. A Petri Nets Semantics for Tower

We formalize Tower to prove safety properties of its semantics, both before and after optimization. A simplified grammar for Tower is given in Figure 4.

A. Petri Nets

Petri nets are a classic formal model for concurrent systems [23]. We briefly introduce enough Petri net machinery to carry out a formalism of Tower.

A *Petri net* is a tuple (S, T, F, M_0) where S is the set of states, T the set of transitions, F the arcs ($F \subset (S \times T) \cup (T \times S)$), $M_0 : S \to \mathbb{N}$ the initial marking. A Petri net is intuitively a bipartite graph enriched with some labeling for nodes and edges. Note that it is also possible to enrich those Petri nets with capacities on each node and weights on arcs, which we do not need in the simplified Tower presented here.

[1] https://github.com/smaccm/smaccm

A *marking* is a an assignment of tokens to states. A transition t is *enabled* if each state s such that there is an arc $s \to t$ has a *token*. An enabled transition t in a marking M is *fired* when we modify M into M' such that each input state of the transition t loses one token, and each output state is given one token. A marking M is *reachable* from a marking M_0 if we can sequentially fire transitions $t_0, ..., t_n$ from M_0 such that the final marking obtained is equal to M. We say that a state s is *safe* if any marking in which s has two tokens is not reachable from the initial marking. A simple Petri net example is provided Figure 5.

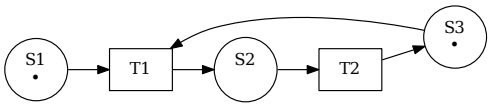
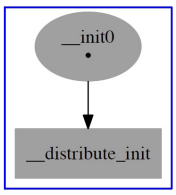


Figure 5: Example of a Petri net with three states (S1, S2, S3) and two transitions (T1, T2). The transition T1 is enabled in the initial marking M_0, (M_0 gives one token to S1 and one to S3).

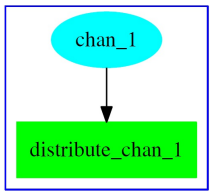
Assuming that there are no contradictory data regarding initial markings, we define the union of two Petri nets as the component-wise union of the nets. This allows us to build Petri nets in a modular way. More formally: $(S, T, F, M_0) \cup (S', T', F', M_0') = (S \cup S', T \cup T', F \cup F', M_0 \cup M_0')$

B. Denotational Semantics of Tower

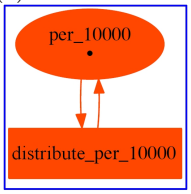
We formalize a Tower program as a Petri net, and then prove safe concurrency properties, defined in Section III-C, both before and after optimization. We operate by induction on the syntax (Figure 4), and construct small Petri subnets and then connect them together using the previously defined union operator on Petri nets. The result consists in a denotational Petri net semantics of the Tower framework, consisting in one function for each type of Tower construct: monitors (M), handlers (H), channels (L), emitters (E).



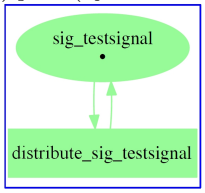
(a) The initial net.



(b) $petri(syncchan : \texttt{1})$



(c) $petri(period : \texttt{10000})$



(d) petri (signal: `testsignal`)

Figure 6: Illustration for the *init* net and and *petri* function.

As a convenience, we first define subnets to build up Tower channel semantics, as illustrated in Figure 6. In the *init* net, the `init` channel fires only once at the beginning of the execution of the program. The *petri* nets are parameterized by the channel. For periodic and signal channels, we have an enabled transition that can fire an unlimited number of times and will distribute one token to each handler listening on this channel. For synchronous channels, the transition can fire only when the incoming state received a token from an other handler.

$$petri(syncchan : id) = \left\{ \begin{array}{ll} states: & \{\texttt{chan\_}id\} \\ transition: & \{\texttt{distribute\_chan\_}id\} \\ arcs: & \{\texttt{chan\_}id \to \texttt{distribute\_chan\_}id\} \\ initial\ marking: & \{0\} \end{array} \right\}$$

$$petri(period : time) = \left\{ \begin{array}{ll} states: & \{\texttt{per\_}time\} \\ transition: & \{\texttt{distribute\_per\_}time\} \\ arcs: & \{\texttt{per\_}time \to \texttt{distribute\_per\_}time, \\ & \texttt{distribute\_per\_}time \to \texttt{per\_}time\} \\ initial\ marking: & \{1\} \end{array} \right\}$$

$$petri(signal : sig) = \left\{ \begin{array}{ll} states: & \{\texttt{sig\_}sig\} \\ transitions: & \{\texttt{distribute\_sig\_}sig\} \\ arcs: & \{\texttt{sig\_}sig \to \texttt{distribute\_sig\_}sig, \\ & \texttt{distribute\_sig\_}sig \to \texttt{sig\_}sig\} \\ initial\ marking: & \{1\} \end{array} \right\}$$

$$init = \left\{ \begin{array}{ll} states: & \{init0\} \\ transitions: & \{distribute\_init0\} \\ arcs: & \{init0 \to distribute\_init0\} \\ initial\ marking: & \{1\} \end{array} \right\}$$

We use these definitions now in encoding the semantics. Emitters are encoded as an arc from a handler to a channel. Channels are encoded with a state and a transition that fires each time a message has to be distributed (giving one token to each handler listening on that channel). Handlers are encoded as a succession of states and transitions, showing the locking and unlocking procedure, plus a state handler's callback computation. We leave nondeterministic the order in which callbacks are called.

$$M[\![\texttt{monitor} : name, \langle handler \rangle_i]\!] = (\bigcup_i H[\![\langle handler \rangle_i]\!]_{name}) \bigcup \left\{ \begin{array}{ll} states: & \{name\} \\ transitions: & \emptyset \\ arcs: & \emptyset \\ initial\ marking: & \{1\} \end{array} \right\}$$

$$H[\![\texttt{handler} : \langle channel \rangle, name, \langle emitter \rangle_i]\!]_{monitor} = (\bigcup_i E[\![\langle emitter \rangle_i]\!]_{name}) \bigcup L[\![\langle channel \rangle]\!]_{name} \bigcup \left\{ \begin{array}{ll} states: & \{name, \texttt{compute\_}name\} \\ transitions: & \{\texttt{lock\_}name, \texttt{release\_}name\} \\ arcs: & \{name \to \texttt{lock\_}name, \\ & monitor \to \texttt{lock\_}name, \\ & \texttt{lock\_}name \to \texttt{compute\_}name, \\ & \texttt{compute\_}name \to \texttt{release\_}name, \\ & \texttt{release\_}name \to monitor\} \\ initial\ marking: & \{0, 0\} \end{array} \right\}$$

$$L[\![syncchan : id]\!]_{handler} = petri(syncchan : id) \bigcup \left\{ \begin{array}{ll} states : & \emptyset \\ transitions : & \emptyset \\ arcs : & \{\texttt{distribute\_chan\_}id \rightarrow handler\} \\ initial\ marking : & \emptyset \end{array} \right\}$$

$$L[\![period : time]\!]_{handler} = petri(period : time) \bigcup \left\{ \begin{array}{ll} states : & \emptyset \\ transitions : & \emptyset \\ arcs : & \{\texttt{distribute\_per\_}time \rightarrow handler\} \\ initial\ marking : & \emptyset \end{array} \right\}$$

$$L[\![signal : sig]\!]_{handler} = petri(signal : sig) \bigcup \left\{ \begin{array}{ll} states : & \emptyset \\ transitions : & \emptyset \\ arcs : & \{\texttt{distribute\_sig\_}sig \rightarrow handler\} \\ initial\ marking : & \emptyset \end{array} \right\}$$

$$L[\![init]\!]_{handler} = init \bigcup \left\{ \begin{array}{ll} states : & \emptyset \\ transitions : & \emptyset \\ arcs : & \{\texttt{distribute\_init0} \rightarrow handler\} \\ initial\ marking : & \emptyset \end{array} \right\}$$

$$E[\![\texttt{emit} : id]\!]_{handler} = \left\{ \begin{array}{ll} states : & \emptyset \\ transitions : & \emptyset \\ arcs : & \{\texttt{release\_}handler \rightarrow \texttt{chan\_}id\} \\ initial\ marking : & \emptyset \end{array} \right\}$$

C. Safety Properties

There are three safety properties Tower programs should guarantee and that must be invariant under the optimization described in Section IV: the absence of channel cycles, the absence of race conditions, and deadlock freedom. We define these properties in terms of the Petri net semantics here.

1) Absence of Channel Cycles: Extract a graph from a Tower program by letting the nodes be handlers and the edges be channels between handlers. Intuitively, a channel cycle is a closed walk in the graph. A channel cycle is a special case of a deadlock caused by a circular data dependency among handlers. In Petri net formulation, a *channel cycle* is a finite sequence of nodes $n_1, m_1, n_2, ...$ such that:

- Each n_i is a state, each m_i is a transition.
- There exist arcs between each following nodes of the sequence.
- Each transition inside a period or a signal construct is unique (that means that we do not rearm a signal or loop on a period).
- There exist i and j such that $i \neq j$, m_i and m_j are channel transitions (i.e. defined by either the initial net or the *petri* function), and $m_i = m_j$.

This can be reformulated into the fact that there does not exist a non-trivial strongly connected component that contains a `distribute_chan_`x node.

2) Absence of Race Conditions: A race condition occurs when there is a concurrent access to some resource not protected by any lock. However, the only way to access those resources are through the callbacks within a handler, which can only be executed after a locking procedure that will guarantee that no other handler that has access to this resource could run simultaneously (indeed, shared resources are at a monitor scope).

Hence, the locking procedure for a handler in the monitor mon_i is translated into the transition called `lock_`mon_i, which, when fired, gives a token to the state `compute_`mon_i. This state symbolizes the callback computation: there is no access to shared resources outside this state.

As described, there are no nested locks in Tower. In the following sections, we will release this constraint by showing that we keep safety if for each handler, the set of resources that are being accessed is a subset of the set of resources protected by the locks acquired by this handler.

3) Deadlock Freedom: We define *deadlock* in a monitor to be a situation in which there exist handlers $H_1, ..., H_n$ ($n >= 2$), which have acquired locks for $X_1, ...X_n$, and are requesting locks for $Y_1, ..., Y_n$ where we have $Y_i \cap X_{i+1} \neq \emptyset$ (we consider that all indices are modulo n). This definition is consistent with the one given by Chandrasekaran et al. [5]. Moreover, we can define for each i, $y_i \in Y_i \cap X_{i+1}$, and suppose without loss of generality that each handler H_i is blocked on the atomic instruction `take_lock`(y_i).

The absence of deadlocks in a monitor mon_i can be interpreted as the following: for each subset of handlers $H_1, ..., H_n$ of the monitor mon_i, there is no reachable marking M in the Petri subnet $\mathcal{P}'$ from the marking M_0 (as defined in Section III-B) such that M has no enabled transition, where $\mathcal{P}'$ is obtained from $\mathcal{P}$ by deleting all other monitors, period, channel, `init` constructs, deleting the incoming arcs of each handler, adding a token to each handler (hence a handler will be executed without any external condition), and delete the handlers that are not part of the chosen subset. This reachability problem in a Petri net is known to be EXPSPACE$-hard$ [19]. However, given the fact that a handler cannot acquire the same lock twice, each state is therefore safe. The problem of detecting a deadlock in monitors therefore lies in co-NP (we have to guess a schedule of polynomial length that will deadlock).

IV. Lock Refinement

One solution to improve parallelism is to release the locking constraints on the handlers, by allowing parallel execution of handlers that do not access common shared resources. One approach is to create a lock per resource and require at the beginning of each handler it acquires all locks necessary before any callbacks are called. Unfortunately, embedded real-time operating systems do not provide an arbitrary number of available locks, and there may be many more shared state variables than locks. Furthermore, such fine-grained locking can cause the overhead of acquiring and releasing locks to be too high, such as in tight control loops. Having fewer locks than shared resources requires efficiently allocating shared resources to a fixed number of locks.

Before refining locks, we must discover which handlers use shared resources. Besides shared state variables declared at the monitor scope, handlers may also access

hardware resources directly (e.g., reading and writing to registers). In a general purpose language with pointers, a precise static analysis to determine all accesses to shared resources is not generally possible. As noted above, the callbacks within handlers in Tower are written in Ivory, a memory-safe embedded programming language [9]. Ivory *references* are statically guaranteed non-null pointers. Reference arithmetic or reference aliasing is not possible except through function calls. Registers are named and are accessed through an interface. These characteristics make a static analysis of handlers to discover the uses of shared resources straightforward and is done as an Ivory compiler pass. In particular, our analysis does not require an inter-procedural analysis, given that a shared resource can be passed to a function only as an argument. Looking at the arguments in top-level function calls in handlers is sufficient to over-approximate safely the shared resources used.

A. Lock Optimization

To begin, consider a matrix representation of the inputs, made of an $n \times m$ boolean matrix H, where n represents the number of handlers, m the number of resources, and $H_{i,j} \equiv \texttt{true}$ if and only if the handler i uses the resource j. We also specify a maximum number of locks l to assign. Hence in this matrix representation, we can define rows in the form H_i, and thus define the scalar product of two rows as:

$$H_i \bullet H_j \equiv \bigvee_{k=1}^{m} H_{i,k} \wedge H_{j,k} \tag{1}$$

to compute whether two handlers share any resources. Similarly, the output is in the form of an $l \times m$ Boolean matrix A, called an *attribution*, where $A_{i,j} \equiv \texttt{true}$ if and only if the resource j is attributed to the lock i. Note that stating that handlers i and j do not share any lock means exactly $(A^t \times H_i) \bullet (A^t \times H_j) \equiv false$. Consider the following invariant of matrix A, stating that a resource is attributed to exactly one lock:

$$\forall j \in \{1, ..., m\}, \exists! i \in \{1, ..., l\}, A_{i,j} \equiv true \tag{2}$$

The goal of the formulation is to find an attribution that increases parallelism in multicore configurations (we define precisely a metric on parallelism in Section V) while satisfying Property 2, and that rewards attributions that increase parallelism.

The reward function is defined as the product of the number of resources (written *nbResources*) of each handler and the frequency (written *Freq*) at which the handlers are called (i.e., the reward will be bigger if the pair uses a lot of resources and/or is run frequently). The intuition is that since we are dealing with real-time systems, greater weight should be given to threads that run frequently, modulo the number of resources they have. Because a handler may be called from multiple threads, we define an ordering on threads based on frequency, and assign the frequency of handlers to be the frequency of the *maximal* thread that calls it. The ordering is as follows, defined over the channels from the Tower grammar (Figure 4):

$$\begin{aligned}
\texttt{init} &< c \text{ for all channels } c \text{ s.t. } c \neq \texttt{init} \\
\texttt{period}t_0 &< \texttt{period}t_1 \text{ iff } t_0 > t_1 \\
c &< \texttt{signal } n\ d \text{ for all } c \text{ s.t. } c \neq \texttt{signal } n\ d \\
\texttt{signal } n_0\ d_0 &= \texttt{signal } n_1\ d_1
\end{aligned}$$

Intuitively, initialization threads run once, so have the lowest frequency. A periodic thread has a higher frequency if its period is smaller. And signal threads, which can be driven by interrupts, are assumed to have highest frequency. Furthermore, we do not distinguish signals with different deadlines. The reward function appears to be simple to compute—it relies on a simple graph analysis from the Tower compiler—and to work well in practice.

We operate over pairs of handlers that do not share resources, as determined by our static analysis, adding a reward each time the resources of the first handler are not attributed to the same lock as the resources of the second handler.

(In the following, the Kronecker delta, δ, is defined as being one if its two arguments are equal, zero otherwise.)

$$\begin{aligned}
\texttt{maximize:} & \sum_{i<j} \delta((A^t \times H_i) \bullet (A^t \times H_j), \texttt{false})\ W(i,j) \\
\texttt{over:} & \ (A_{i,j})_{i \in \{1,...,l\}, j \in \{1,...,m\}} \\
\texttt{subject to:} & \ A \text{ satisfies the property (2)} \\
\text{where:} & \ W(i,j) = Freq(H_i) \times Freq(H_j) \times \\
& \quad nbResources(H_i) \times nbResources(H_j)
\end{aligned}$$

We solve the optimization problem using a Partial Weighted MAXSAT formulation to ensure an upper bound on the number of locks used. MAXSAT is the problem of determining the maximum number of clauses in a conjunctive normal formula that can be satisfied. This is a variant of SAT which consists only in determining if all the clauses can be satisfied or not. Partial Weighted MAXSAT (PWMS) is a variant of MAXSAT in which we introduce weights to each clause, and segregate the clauses between *hard clauses*, which must be satisfied, and *soft clauses*, which may be satisfied. Many solvers for MAXSAT instances exist, and in this work we used OPENWBO[21], an open-source solver with Glucose 3.0 as the underlying SAT solver[1], [8].

We solve the optimization problem using Partial Weighted Max Sat (PWMS) on variables $A_{i,j}$ to have a Boolean matrix that satisfies the property written in (2) and encodes the PWMS problem as follows. Variables are the assignments of resources to locks; the hard clauses ensure that every resource is attributed to exactly one lock; and soft clauses state that for every pair of handlers

(i, j) that share no resources, and for every resource α that i uses and every resource β that j uses, respectively, minimize the assignments of α and β to the same lock, weighted by the frequency of the handlers' usage (weights are written as a subscript in the soft clauses).

$$\texttt{variables: } (A_{i,j})_{i\in\{1,\dots,l\},j\in\{1,\dots,m\}}$$

$$\texttt{hard clauses: } \bigwedge_{j=1}^{m}\Big(\bigvee_{i=1}^{l} A_{i,j}\Big)\bigwedge\Big(\bigwedge_{1\le i<k\le l} \neg A_{i,j}\vee\neg A_{k,j}\Big)$$

$$\texttt{soft clauses: } \bigwedge_{\substack{1\le i<j\le n\\ H_i\bullet H_j\equiv\texttt{false}}}\ \bigwedge_{\substack{1<\alpha<m\\ H_{i,\alpha}\equiv\texttt{true}}}\ \bigwedge_{\substack{1<\beta<m\\ H_{j,\beta}\equiv\texttt{true}}}\ \bigwedge_{k=1}^{l} \Big(\neg A_{\alpha,k}\vee\neg A_{\beta,k}\Big)_{Freq(H_i)\times Freq(H_j)}$$

Finally, as a post-processing step to improve efficiency, we define H_{L_i}, the set of handlers that have to take the lock L_i. We define a partial order $\sqsubseteq$ on locks such that $L_i \sqsubseteq L_j$ if and only if $H_{L_i} \subseteq H_{L_j}$. We finally apply some basic optimizations that reduce the final number of locks:

- Monitors with no resource do not generate any lock.
- Locks L_i and L_j for which $L_i \sqsubseteq L_j$ are merged together (more precisely, L_i is merged into L_j).

B. New semantics

Two modifications of the semantics given in Section III have to be made to address the new locking system. The first generalizes it to allow creating several locks per monitor. This is done by adding several more states initialized with one token, each of them representing one mutex that will lose its token when taken.

$$M[\![\texttt{monitor} : name, \langle handler\rangle_i, locks_i]\!] = \Big(\bigcup_i H[\![\langle handler\rangle_i]\!]_{name}\Big) \bigcup \left\{\begin{array}{ll} states: & \{(name\_lock)_i\}\\ transitions: & \emptyset\\ arcs: & \emptyset\\ initial\ marking: & \{(1)_i\}\end{array}\right\}$$

The second modification changes for each handler the locking procedure, by creating one transition per lock to acquire (we release all the locks at the same time, given that the unlocking order does not influence the safety properties).

$$H[\![\texttt{handler} : \langle c\rangle, name, \langle e\rangle_i, locks_i]\!]_{monitor} = \Big(\bigcup_i E[\![\langle e\rangle_i]\!]_{name}\Big) \bigcup L[\![\langle c\rangle]\!]_{name} \bigcup \left\{\begin{array}{ll} states: & \{name, \texttt{compute\_}name,\\ & (\texttt{locked\_}i\_name)_{i\neq max(i)}\}\\ tr.: & \{(\texttt{lock\_}i\_name)_i, \texttt{release\_}name\}\\ arcs: & \{name \to \texttt{lock\_}min(i)\texttt{\_}name,\\ & (monitor\_lock_i \to \texttt{lock\_}i\_name)_i,\\ & (\texttt{lock\_}i\_name \to \texttt{locked\_}i\_name)_{i\neq(max(i))},\\ & \texttt{lock\_}max(i)\texttt{\_}name \to \texttt{compute\_}name,\\ & \texttt{compute\_}name \to \texttt{release\_}name,\\ & (\texttt{release\_}name \to monitor\_lock_i)_i\}\\ i.m.: & \{0, 0, (0)_{i\neq max(i)}\}\end{array}\right\}$$

C. Proofs of Safety

Let us reconsider our three safety properties with respect to the optimization we have described.

First, lock refinement does not affect the message passing (modifications only happen inside the monitors); hence the absence of channel cycles is preserved in the new Petri net model.

Proof. More rigorously, let us consider a channel cycle $n_1, m_1, n_2, ..., n_p$ (such that the sequence respects the properties expressed in III-C1) in the original program before lock refinement: then we construct a new channel cycle in the Petri net after lock refinement by keeping all nodes and transitions, except that $name \to \texttt{lock\_}name \to \texttt{compute\_}name$ is replaced by $name \to \texttt{lock\_}min(i)\texttt{\_}name \to \texttt{locked\_}min(i)\texttt{\_}name \to ... \to \texttt{lock\_}max(i)\texttt{\_}name \to \texttt{compute\_}name$. We check easily that the new sequence indeed verifies the properties expressed in the definition of channel cycle: the length of the cycle is still finite, we alternate between states and transitions following arcs, we keep the uniqueness of transitions inside channel constructs and we still have m_i and m_j from the channel cycle before optimization that are present in the new sequence of nodes, except that their indexes increased while still being different from one another. The converse is trivially true by just remarking that the construct done above is reversible (more precisely, the previous construction gives an isomorphism between the channel cycles of the Petri nets before and after lock refinement). □

Second, race conditions can only happen after lock refinement if there are resources of global scope that get accessed outside any lock. The system is safe if for each handler, the set of resources that are being accessed is a subset of the set of resources protected by the locks acquired by this handler, which is equivalent to the soundness of the static analysis done previously. To check this property in terms of Petri nets, we extended the Petri nets by adding an extra labeling to handler states that indicates the resources used by the handler, and for each lock state, the resources that the lock protects. Those changes in the semantics are not presented here, for simplicity and readability purposes. At compile time, we check that the resources used by each handler are indeed a subset of the union of the resources protected by the locks taken by the handler.

Third, deadlock freedom is the least obvious of the three properties.

Proof. Define an ordering relation $\leq$ over locks, and enforce by convention that each handler will have to take the locks following the same order (this is enforced in the semantics by the transitions $\texttt{lock\_}min(i)\texttt{\_}name \to \texttt{locked\_}min(i)\texttt{\_}name \to ... \to \texttt{lock\_}max(i)\texttt{\_}name$). Suppose handlers $H_1, ..., H_n$ are deadlocked. Then by using the definition of deadlock given in section III-C3,

we can define for each H_i, X_i the set of locks acquired and Y_i the set of locks that are still to be acquired, and we have that $\exists y_i \in Y_i \cap X_{i+1}$. Without loss of generality, we can say that for each handler H_i, the transition `lock_`$y(i)$`_`$name$ is not enabled. By using the fact that we have an ordering relation on locks, we can say that $y_1 \leq y_2$ given that $y_1 \in X_2$ and $y_2 \in Y_2$. The same can be applied circularly, which gives $y_i \leq y_{i+1}$. Hence by antisymmetry and transitivity we can conclude that $y_1 = ... = y_n$, giving deadlock freedom by contradiction. □

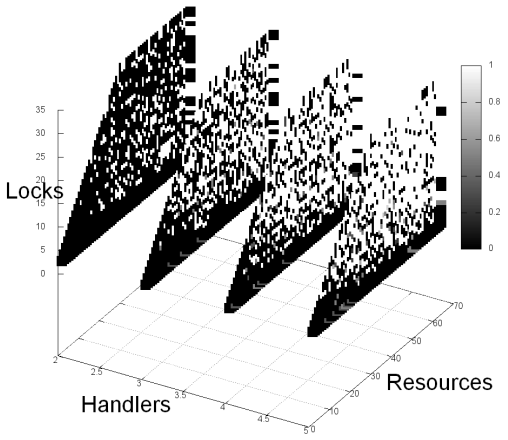
V. Experimental Results

In this section, our benchmarking shows how well the problem formulation scales using PWMS. In particular, given the complexity of the problem, we run PWMS for a fixed period of time, at which a solution is returned that may not be optimal.

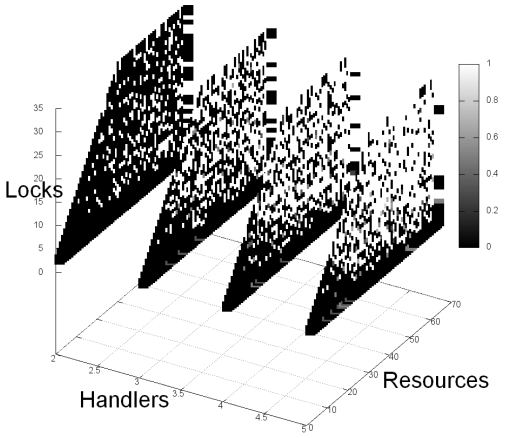
We first define a metric based on comparing the resulting parallelism to the theoretical maximum. We do so by defining two graphs in which the nodes are the handlers, and then compare their densities. The first graph has its edges defined by the relation $H_i \bullet H_j = false$ (i.e., the handlers H_i and H_j can run simultaneously in theory) and the second by the relation $(H_i \times A^t) \bullet (H_j \times A^t) = false$ (ie the handlers H_i and H_j will run simultaneously in executing the optimization). After computing the graph density of the first graph (and discarding the ones in which there is no parallelism possible (the density equal to zero), we apply our optimization and compute the density of the second graph and compute the *relative error* ($\Delta = \frac{\texttt{theoretical}-\texttt{experimental}}{\texttt{theoretical}}$) which will be our benchmarking main value.

In the benchmarking, we generate random Tower programs, run the lock refinement optimization on them, then record the relative error of the results. The essential question addressed in the benchmarking is how small a relative error can be achieved using PWMS. open-wbo supports setting a timeout. When the timeout is reached, if the hard clauses are satisfied, then the best result reached with respect to the weighted soft clauses is returned. We can only hope to obtain an approximation on large instances since lock optimization is NP-complete [10].

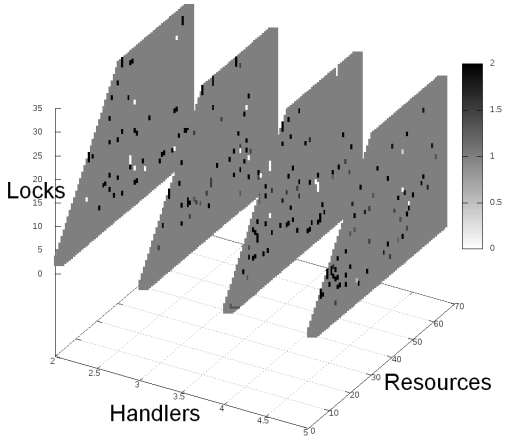
Each test case with R resources is generated by drawing a number of resources per handler P between one and R uniformly, and then for each handler draw P resources out of R (in particular, for each test case, all handlers have the same number of resources). To compare two concrete data points, we allocate either 60 or 900 seconds to each test case, the solver returning the best solution at the end of this timeout (or the final result if the solver returns before timeout). The results are shown in Figure 7. As can be seen, the optimization scales well with the number of resources, but not well with locks. Furthermore, in most cases tested, more time does not improve the results, suggesting that if a good optimization is not found quickly,



(a) 60 seconds.



(b) 900 seconds.



(c) 900s compared to 60s (differential image).

Figure 7: Comparison of the time spent on each test case for a same test set. Each test case is defined by its coordinates (number of handlers, total number of resources, number of locks allocated). The ranges chosen are: $handlers \in \{2, ..., 5\}$, $resources \in \{2, ..., 64\}$ and $locks \in \{2, ..., 32\}$. Figures (a), (b): we show the relative error as computed before (black: perfect computation, white: the optimization failed and no parallelism is found). Figure (c): we show the delta between (a) and (b). The grey color indicates no change, the black means we improved the results whereas the white shows that the result become worse.

it is likely not to be found even with substantially more time.[2]

The raw data can be found at https://github.com/GaloisInc/pwms-instances.

VI. Case-Study: The SMACCMPilot Autopilot

To demonstrate the scalability of our approach on a large code-base, we apply the optimization approach to the SMACCPilot autopilot. The *UAV* (Unmanned aerial vehicle) airframe is a quadcopter (3DR IRIS+), with two primary flight controllers, a core flight controller and a mission controller. The autopilot is open source.[3]

A. Autopilot Architecture

The flight computer hardware is the PX4 Pixhawk [25], the main processor for which is a 168Mhz STM32F427 ARM-v7M Cortex-M4 CPU. The flight computer manages sensor polling, sensor fusion, inner loop control, motor control, and direct pilot input (from a 2.4GHz radio). The flight computer software is written using Tower and there are backends to generate code for both the eChronos [7] and FreeRTOS [2] RTOSes.

The mission controller hardware is an Odroid-XU board with a custom IO board. The board runs the formally-verified seL4 microkernel [18]. The mission computer handles higher-level processing and outer-loop control. For example, it has a camera, WiFi, and an encrypted data link over a 915MHz radio to the ground control station. The mission computer and flight computer communicate over a CAN bus.

B. Optimizing SMACCMPilot

The autopilot flight controller module has 157 monitors, of which 32 have no shared resources (30 of them have only one handler), and 41 monitors have handlers that can run in parallel (i.e. the graph density is not null, as defined in section V). The total lines of software are just under 100K lines of code, not counting comments or empty lines. After running our optimization (allowing 60 seconds to the PWMS solver for each monitor), in 39 monitors out of 41, we achieved a perfect result, having a relative error of zero (as defined in section V). Of the two remaining monitors, in the monitor managing communication to a I/O coprocessor over high-speed serial via a direct memory-access controller (`px4io_driver`), we have a relative error of 0.17 (density of 0.68 instead of 0.82 in theory), and in the monitor managing inner loop control (`control`), the optimization did not manage to improve parallelism, yielding a relative error of 1. Those results can be explained by the huge instances generated for the last two monitors, as shown in the Figure 8.[4]

[2]Some additional noise was introduced into the benchmarks due to PWMS non-deterministically entering a sleep state and having to be killed off manually, which shows an abrupt degradation or improvement in the results for some specific instances.

[3]http://smaccmpilot.org/

[4]The PWMS instances can be found at https://github.com/GaloisInc/pwms-instances.

These results suggest that on a real code base developed using a Hoare-monitor style, many locks are not necessary, and there are generally significant optimization opportunities. In our case, much of the shared state is relatively localized to a small number of monitors.

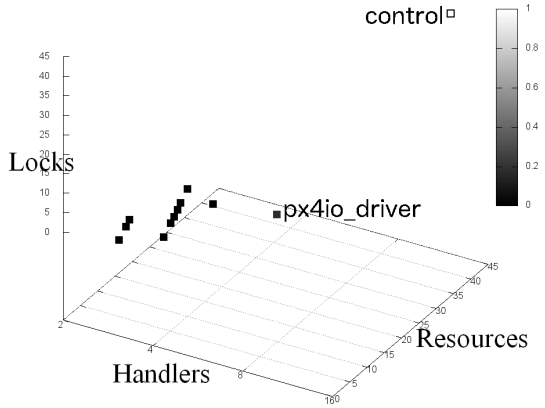


Figure 8: Representation of the 41 monitors. Each monitor is defined by its coordinates (number of handlers on a logarithmic scale, total number of resources, number of locks allocated). We show the relative error as computed before (black = 0, perfect computation, white = 1, the optimization failed and no parallelism is found) The empty square on top right corresponds to the `control` monitor. Its position shows the inability to solve the PWMS instance for it within 60 seconds.

VII. Related work

Our work can be placed within the context of the lock granularity debate in multicore processing [3]. Hoare monitors introduce very coarse-grained—but safe—locking for user applications. The benefit of fine-grained locking is that it can be more efficient, but it can also subtly introduce bugs. We refine locks automatically, up to a fixed number of locks, allowing programmers to combine the simplicity and elegance of Hoare monitors with more efficient concurrency in an embedded real-time setting.

Others [10], [6], [12] have addressed the problem of lock allocation for *atomic sections* [22]— a , with similar goals to us. Most related is the work by Emmi et al. in which the authors automatically allocate locks for atomic regions [10]. Their work considers general-purpose C programs, so they have a more sophisticated pointer analysis to ensure safety. They encode the problem using SAT; we arguably have a more natural encoding into the more expressive PWMS. While our analysis is arguably more coarse-grained, our SMACCMPilot case-study is 100k lines of code; theirs are over programs that are 2k or fewer lines with no more than 11 atomic regions.

While somewhat rare in the real-time literature, Jeffay uses a Hoare-monitor based solution in providing optimality results for scheduling preemptive sporadic tasks [14].

A large body of literature exists on formal models of concurrent systems [26], and we are agnostic regarding other models, such as Kahn process networks [17]. Our work is largely agnostic regarding the particular formalism, although we want a language expressive and precise enough to reason about the safety properties described in Section III-C. While not pursued in this work, a formal semantics paves the way to model-checking user-supplied assertions about concurrent embedded programs [15].

VIII. Conclusion

We have described and formalized Tower, a framework for specifying real-time Hoare monitors, as well as a systematic optimization technique intended to improve runtime efficiency. We have proved that this technique maintains key safety properties, which has been experimentally confirmed by tests on real hardware.

There are a variety of avenues for additional research. One way to improve the results would consist in investigating other reward functions. We used a naive approximation for the frequency of handler calls. There is a practical trade-off: a more refined reward function might improve performance in practice, while a simple reward function might make PWMS solving simpler. Finally, we believe Hoare-monitor based concurrency is interesting in its own right and deserves more experimentation.

Acknowledgments

This work is supported by DARPA under contract no. FA8750-12-9-0169. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

We thank Jason Dagit (Galois), Adam Foltzer (Galois), Iavor Diatchki (Galois), Marc Pouzet (ENS), Pat Hickey (Helium), Dumitru Potop-Butucaru (INRIA), Simon Winwood (Galois), and Eddy Westbrook (Galois) for their advice. Pat Hickey is the primary original author of Tower.

References

[1] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proceedings of the 21st International Jont Conference on Artifical Intelligence (IJCAI)*. Morgan Kaufmann, 2009.

[2] R. Barry, "FreeRTOS," Website, http://www.freertos.org/.

[3] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[4] P. Brinch Hansen, *Class Concept*. Prentice Hall, 1973, ch. 7.2. [Online]. Available: http://brinch-hansen.net/papers/1973b.pdf

[5] P. Chandrasekaran, S. K. K. B, R. L. Minz, D. D'Souza, and L. Meshram, "A multi-core version of FreeRTOS verified for datarace and deadlock freedom," in *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Oct 2014, pp. 62–71.

[6] S. Cherem, T. Chilimbi, and S. Gulwani, "Inferring locks for atomic sections," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2008, pp. 304–315.

[7] Data61, "eChronos," Website, 2016, https://ts.data61.csiro.au/projects/TS/echronos/.

[8] N. Eén and N. Sörensson, *An Extensible SAT-solver*. Springer, 2004.

[9] T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury, "Guilt free ivory," in *Proceedings of the ACM Symposium on Haskell*. ACM, 2015.

[10] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar, "Lock allocation," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, 2007, pp. 291–296.

[11] P. Feiler, D. Gluch, and J. Hudak, "The architecture analysis and design language (aadl): An introduction," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2006-TN-011, 2006. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7879

[12] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv, *Reasoning about Lock Placements*. Springer, 2012, pp. 336–356.

[13] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, Oct. 1974.

[14] K. Jeffay, "Analysis of a synchronization and scheduling discipline for real-time tasks with preemption constraints," in *Proceedings of the Real Time Systems Symposium*, 1989, pp. 295–305.

[15] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured petri nets and cpn tools for modelling and validation of concurrent systems," *International Journal on Software Tools for Technology Transfer*, vol. 9, pp. 213–254, 2007.

[16] M. Jones, "What really happened on Mars?" Website (posted email), December 1997, http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html.

[17] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, J. L. Rosenfeld, Ed. North Holland, Amsterdam, Aug 1974, pp. 471–475.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629596

[19] R. Lipton, *The Reachability Problem Requires Exponential Space*. Yale University, 1976.

[20] A. Lyons and G. Heiser, "It's time: OS mechanisms for enforcing asymmetric temporal integrity," *CoRR*, vol. abs/1606.00111, 2016. [Online]. Available: http://arxiv.org/abs/1606.00111

[21] R. Martins, V. Manquinho, and I. Lynce, *Open-WBO: A Modular MaxSAT Solver,*. Springer, 2014.

[22] B. McCloskey, F. Zhou, D. Gay, and E. Brewer, "Autolocker: Synchronization inference for atomic sections," in *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 2006, pp. 346–358.

[23] C. A. Petri, *Grundsätzliches zur Beschreibung Diskreter Prozesse*. Basel: Birkhäuser Basel, 1967, pp. 121–140. [Online]. Available: http://dx.doi.org/10.1007/978-3-0348-5879-3_10

[24] S. Peyton Jones *et al.*, "The Haskell 98 language and libraries: The revised report," *Journal of Functional Programming*, vol. 13, no. 1, pp. 0–255, Jan 2003.

[25] Pixhawk. (2016) Lorenz meier. Https://pixhawk.org/.

[26] G. Winskel and M. Nielsen, "Models for concurrency," in *Handbook of Logic in Computer Science (Vol. 4)*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, 1995, pp. 1–148.