# Model Checking Race-Freeness

Parosh Aziz Abdulla
Uppsala University, Sweden
<parosh@it.uu.se>

Frédéric Haziza
Uppsala University, Sweden
<daz@it.uu.se>

Mats Kindahl
Sun Microsystems,
Database Technology Group
<mats@sun.com>

## Abstract

*With the introduction of highly concurrent systems in standard desktop computers, ensuring correctness of industrial-size concurrent programs is becoming increasingly important. One of the most important standards in use for developing multi-threaded programs is the POSIX Threads standard, commonly known as* PThreads. *Of particular importance, the analysis of industrial code should, as far as possible, be automatic and not require annotations or other forms of specifications of the code.*

*Model checking has been one of the most successful approaches to program verification during the last two decades. The size and complexity of applications which can be handled have increased rapidly through integration with symbolic techniques. These methods are designed to work on finite (but large) state spaces. This framework fails to deal with several essential aspects of behaviours for multi-threaded programs: there is no bound* a priori *on the number of threads which may arise in a given run of the system; each thread manipulates local variables which often range over unbounded domains; and the system has a dynamic structure in the sense that threads can be created and killed throughout execution of the system. In this paper we concentrate on checking a particular class of properties for concurrent programs, namely* safety properties. *In particular, we focus on* race-freeness*, that is, the absence of race conditions (also known as data races) in shared-variable pthreaded programs.*

*We will follow a particular methodology which we have earlier developed for model checking general classes of infinite-state systems [1, 3, 6, 8, 9] and apply a symbolic backward reachability analysis to verify the safety property. Since we construct a model as an over-approximation of the original program, proving the safety property in the model implies that the property also holds in the original system. Surprisingly, it leads to a quite efficient analysis which can be carried out* fully automatically.

## 1. Introduction

The behaviours of concurrent (or multi-threaded) programs are highly nontrivial and hard to predict. It is important to develop rigorous methods to verify their correctness. It is now also widely accepted that verification methods should be *automatic*. This would allow engineers to perform verification, without needing to be familiar with the complex constructions and algorithms behind the tools.

In this paper, we will concentrate on a particular approach to verification of concurrent programs, namely that of *model checking* [4, 16]. The aim of model checking is to provide an algorithmic solution to the verification problem. Concurrent programs involve several complex features which often give rise to infinite state spaces. First, there is no bound *a priori* on the number of threads which may arise in a given run of the system. In addition, each thread manipulates local variables which often range over unbounded domains. Furthermore, a system has a dynamic configuration in the sense that threads can be created and terminated throughout execution of the system.

We concentrate on checking a particular class of properties for concurrent programs, namely *safety properties*. Intuitively, a safety property states that "nothing bad will ever occur during the execution of the system". In particular, we focus on *race-freeness*, i.e. the absence of race conditions (also called data races) in shared-variable concurrent programs.

A race condition is a situation in which one process changes a variable which another process has previously read and the other process does not get notified of the change. A process can, for example, write a variable that a second process reads, but the first process continues execution – namely races ahead – and changes the variable again before the second process sees the result of the first change. Another example: when a process checks a variable and takes action based on the content of the variable, it is possible for another process to "sneak in" and change the variable in between the check and the action in such a way that the action is no longer appropriate. Alternatively, one can define a race condition as the possibility of incorrect results in the presence on unlucky timing in concurrent programs, that is, getting the right answer relies on lucky timing.

Race conditions are of particular interest because they can lead to rather devious bugs. These bugs are extremely hard to track since they are non-deterministic and difficult to reproduce. The kind of errors caused by race condition are very subtle and often manifest themselves in the form of corrupted or incorrect variable data. Unfortunately, it often means that the error will not harm the system immediately, but it will manifest itself when some other code is executed, which relies on the data to be correct. This makes the process of locating the original race condition even more difficult. To avoid data corruption or incorrectness, the programmer uses synchronization techniques to constraint all possible process interleavings to only the desirable ones. Race condition usually unveil incorrectly synchronized program.

**Related Work.** Existing race checkers fall into three main categories: on-the-fly, ahead-of-time and post-mortem tools. They exhibit different strengths and can perform race detection, while our method focuses at the moment on race-freeness. The ahead-of-time approach encompasses static analysis and compile-time heuristics, while on-the-fly approaches are by nature dynamic. The post-mortem approach is a combination of static and dynamic techniques.

Type-based solutions provide a strong assurance to the programmer, in addition to the familiarity of a compiler-based solution [10]. If a program type-checks, it is guaranteed to be race-free. It is however necessary to examine the source code as a whole in order to draw conclusion about to find relations between the locks and the shared data that they protect. This is often alleviated by requiring annotations from the programmer, as to whether a function has an effect clause or not, such as “the caller must hold lock L”. Moreover, this only forces a programming discipline and it can also disallow some race-free programs.

Dynamic tools visit only feasible paths and have an accurate view of the values of the shared data, while static tools must be conservative. They are based on two main approaches: the lockset and the happens-before approaches. The lockset algorithm works on the assumption that shared variables should be protected by an appropriate lock and any failure to comply to that discipline is reported. Lockset tools tend to report many false positives. This technique has been first implemented in Eraser [17]. The happens-before technique [13] watches for any accesses of shared variables that do not have an implied ordering between them. This technique is highly dependant on the actual thread execution ordering, and instrumentation might bias the analysis. Moreover, it only reports a subset of all race conditions (however real ones). These two main approaches are often combined to get the best of both worlds, e.g. MultiRace and RaceTrack [19, 14].

Model checkers, such as Verisoft, Bandera and KISS [12, 7, 15, 5], check concurrent programs with a finite and fixed number of threads. They often require a user supplied abstraction.

A precise way to detect race conditions would be to search the state space (of a *model* of the program) for a state where multiple threads try to access and change the same variable. The technique is sound (i.e. being able to prove the race-freeness of a concurrent program), but more importantly it seems to be much more precise than the other techniques. It indeed detects the race conditions themselves instead of detecting violations of the locking discipline that can be used to prevent race conditions.

**Verification Method.** We first construct a model for concurrent programs, where each configuration (snapshot) of the system is represented by a petri net. The transitions in the petri net represent the instructions of the program. The configuration of the system changes dynamically during its execution, by firing transitions in the petri net. The system starts running from an *initial configuration* and we characterize a set of *bad configurations* which violate the given safety property (i.e. configurations which should not occur during the execution of the system). Checking the safety property amounts then to performing *reachability analysis*: Is it possible to reach a bad configuration from an initial configuration through a sequence of actions performed by different threads?

We will follow a particular methodology which we have earlier developed for model checking general classes of infinite-state systems [1]. The method is designed to be applied for verification of safety properties for infinite-state systems which are *monotonic* w.r.t. a *well quasi-ordering* on the set of configurations. The main idea is to perform symbolic backward reachability analysis to check safety properties for such systems.

**Outline.** We present in Section 2 and Section 3 the type of programs we consider and how we extract a model from them. In Section 4, we state the class of safety properties we want to check, and in Section 5 a solution. We finish with a small description of our experimental results in Section 6 and a conclusion.

## 2. Language

We analyze concurrent programs written in a stripped version of the C language and using POSIX threads. We call it here the SML language (Simple Multithreaded Language). We use a relatively small set of operations which follows closely those of an instruction set architecture (ISA). Intuitively, we filter out the C language and Pthreads constructs. An ISA includes arithmetic (e.g. *add* and *sub*) and logic instructions (e.g. *and*, *or*, *not*), data instructions (e.g. *load*, *store*) and control flow instructions (e.g. *goto*, *branch*). Arithmetic and logic instructions are simply pure CPU operations. We are interested in instructions touching the main memory and instructions controlling the flow. Hence, the language we allow abstracts away the CPU operations and narrows down the operations to a set of *movers*, i.e. loading from and storing to the main memory and operations related to controlling the flow, thread synchronization and thread bookkeeping.

We do not define formally the semantics of SML programs. They allow multiple threads to execute concurrently and manipulate two kinds of variables: local and global. A global variable is shared by all threads. A local variable is local to a given thread and cannot be accessed by other threads. A shared variable $x$ is a global variable that can be read and written by any thread. We abstract away the data and extract the mover instructions as *Read x* or *Write x* accordingly. Local variables are useful for the control flow only and therefore abstracted away. The *if*, *if-then-else* and *while* statements are allowed in the form of *branch* and *goto* combinations. *for*-loops can be unrolled and equivalently implemented with *while* and *goto*.

Threads can use locks for synchronization purposes through *acquire* and *release* primitives. A lock (also called *mutex*) is a special shared variable that has two values: it is either *free* or *busy*. A thread trying to acquire a lock will block if the lock is busy. It will acquire the lock (i.e. atomically set it to busy) if the lock was free and continue execution. Releasing the lock resets it to free.

Threads can use condition variables for synchronization purposes through *wait* and *signal* primitives. According to the POSIX semantics, wait shall be called while holding a lock or undefined behavior results. If a thread waits on a condition variable, it releases the lock and blocks its execution. A thread will try to re-acquire the lock (leading to eventual delay) and resume execution if the condition variable is signaled by another thread. We do not allow at the moment to broadcast a wake up signal (i.e. wake up all threads waiting on the condition variable). Signaling a condition variable, while no thread is waiting on that condition variable, has no effect. The wait instruction suffers from spurious wakeups as it may return when no thread specifically signalled that condition variable. We do not disallow this behaviour and will handle it in our model. It simply forces the programmer to check a predicate invariant upon return and it is not orthogonal to our analysis.

Finally, a thread can create another thread and continue its execution. We do not allow at the moment to wait for termination of newly created threads.

Figure 1 and Figure 2 show examples written in C using Pthreads and its equivalent using the SML language. We do not show the whole code but just the part of interest. Figure 1 shows a critical section problem where multiple threads try to update a counter. Only one thread is allowed to update the counter at a time, to avoid a data race. Figure 2 is a producers/consumers example. The producers update a shared buffer, if the buffer is empty and the consumer reads the buffer if the buffer is full.

```
int counter;
pthread_mutex_t L;

pthread_mutex_lock(L);
counter++;
pthread_mutex_unlock(L);
```

```
shared counter, L;

acquire L;
read counter;
write counter;
release L;
```

**Figure 1. Critical section problem in pthreaded code (left) and its SML counterpart (right).**

```
int buffer; pthread_mutex_t L;
pthread_cond_t cvEmpty, cvFull;

//Many Producers
pthread_mutex_lock(L);
while(true){ /*branch*/
  pthread_cond_wait(cvEmpty,L);
  buffer = data;
  pthread_cond_signal(cvFull);
 }
pthread_mutex_unlock(L);


//Many Consumers
pthread_mutex_lock(L);
while(true){ /*branch*/
  pthread_cond_wait(cvFull,L);
  val = buffer;
  pthread_cond_signal(cvEmpty);
 }
pthread_mutex_unlock(L);
```

```
shared buffer, L, cvEmpty, cvFull;

//Many Producers
acquire L;
while(true){ /*branch*/
  wait cvEmpty, L;
  write buffer;
  signal cvFull;
 }
release L;


//Many Consumers
acquire L;
while(true){ /*branch*/
  wait cvFull, L;
  read buffer;
  signal cvEmpty;
 }
release L;
```

**Figure 2. Producers/Consumers in pthreaded code (left) and its SML counterpart (right).**

## 3. Model

### 3.1. Petri Nets

A Petri net $\mathcal{N}$ is a tuple $(P, T, F)$ where $P$ is a finite set of *places*, $T$ is a finite set of *transitions* and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, such that $P \cap T = \emptyset$.

If $(p, t) \in F$, $p$ is said to be an *input* place of $t$ and if $(t, p) \in F$, $p$ is said to be an *output* place of $t$. We use $I(t) = \{p \in P |\ (p, t) \in F\}$ and $O(t) = \{p \in P |\ (t, p) \in F\}$ to denote the sets of input places and output places of $t$ respectively.

A *configuration* $c$ of a Petri net, often called a *marking* in the literature, is a multiset over $P$ and represents a valuation of the number of *tokens* in each place.

The transition system induced by a Petri net consists of the set configurations together with the transition relation defined on them. The operational semantics of a Petri net is defined through the notion of *firing* transitions. This gives a transition relation on the set of configurations. More precisely, a transition fires by removing tokens from its input places and creating new tokens which are distributed to its output places. A transition is *enabled* if each of its input places has at least one token. A transition may fire if it is enabled. [1]Formally, when a transition $t$ is enabled, we write $c \xrightarrow{t} c'$ if $c'$ is the result of firing $t$ on $c$. We define $\longrightarrow$ as $\bigcup_{t \in T} \xrightarrow{t}$ and use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. For sets $C_1$ and $C_2$ of configurations, we use $C_1 \longrightarrow C_2$ to denote that $c_1 \longrightarrow c_2$ for some $c_1 \in C_1$ and $c_2 \in C_2$.

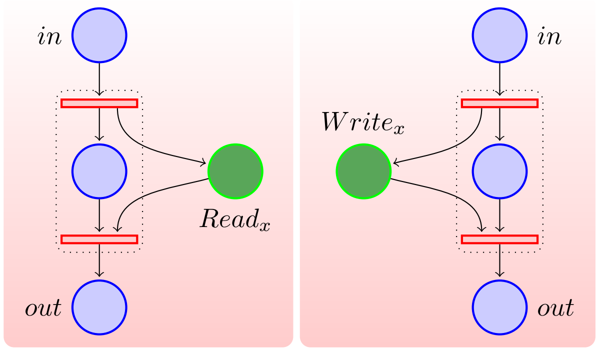
### 3.2. Modeling programs using Petri Nets

We model the flow of control in SML programs using Petri Nets. A concurrent program contains multiple separate threads of control and each thread is assigned a particular task, so-called *job type*, modeled by a petri net. SML programs have a finite number of job types. Multiple instances of a job type will be modeled by multiple tokens. Note that the structure of the petri net is then static.

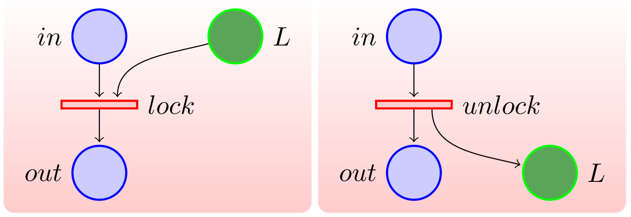
Transitions in the petri net correspond to thread statements and places are used to control the flow and the shared variable dependencies. This modeling formally captures the concurrency between threads using the concurrency constructs of a petri net, captures synchronization between threads (e.g. locks, access to shared variables, condition variables, ...) using appropriate mechanisms in the net, and formalizes the fact that data is abstracted in a sound manner. In the following, a place will be represented by a ○ and by a ● when it is shared. A transition will be represented as ▭.

[1]Petri nets allow multiple arcs from a place to a transition (and vice-versa), where as many tokens are removed (or created) when an enabled transition fires. But we will not use that construct.

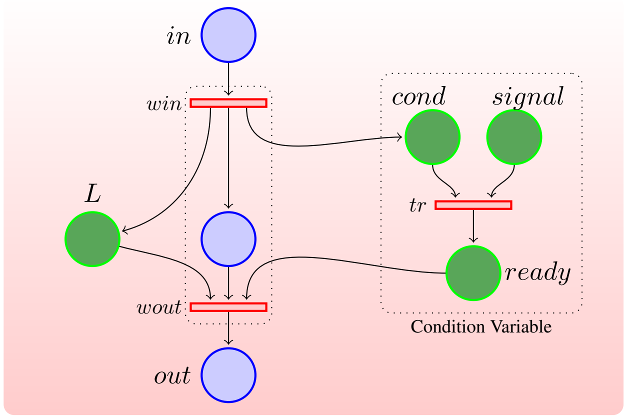
**Reading and Writing a shared variable.** A shared variable $v$ is associated with two places, $Read_v$ and $Write_v$. A thread places a token in $Read_v$ (resp. $Write_v$) if it is currently accessing the variable $v$ for reading (resp. writing). We model read and write accesses to shared variables with two transitions.



**Acquiring and releasing a lock.** There is a place $L$ associated with each lock. Intuitively, if $L$ contains a token, the lock is free, otherwise it is busy. This ensures that only one thread can hold the lock at a time. Note that $L$ is a global variable.

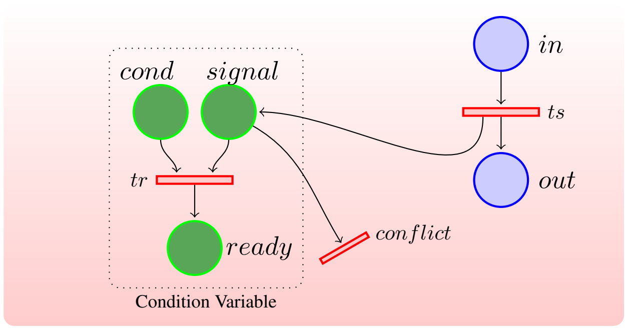


**Waiting on a condition variable.** A condition variable is modeled with 3 places, namely $cond$, $signal$ and $ready$, and 3 transitions as follows.



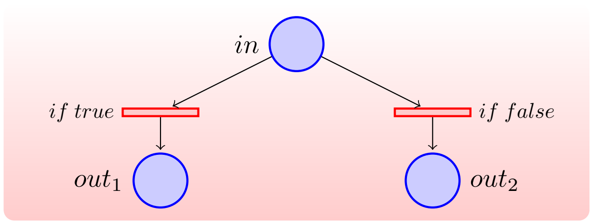
The transition $win$ releases the lock and places a token in the $cond$ place. The thread is then blocked since neither $wout$ nor $tr$ are enabled. If another thread places a token in $signal$, $tr$ is enabled. If $tr$ fires, the blocked thread is ready to fire $wout$ to resume execution, with an eventual delay for (re-)acquiring the lock.

**Signaling a condition variable.** A thread can wake up another blocked thread by placing a token in the $signal$ place.



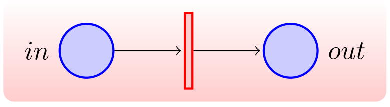
Signaling a condition variable in the model while no thread is waiting on it should have no effect. That is to say, the token in $signal$ should be consumed if the $cond$ place is empty. To alleviate the problem, we use a $conflict$ transition. Recall that an enabled transition only *may* fire. As it is not possible to model the firing of a transition based on the condition that a place is empty, we introduce an abstraction where signals might be lost (i.e. signaling a condition variable might not wake up other threads which are waiting on that condition variable). Nevertheless, it is an over-approximation so we do not bias correctness.

**Branching.** The *if*, *if-then-else* and *while* statements are easily implemented with *branch* and *goto*. We therefore only model those latter.

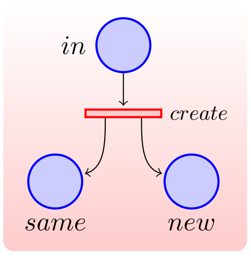


Since we abstract away all data, we cannot determine the branching based on a predicate evaluation. So we use an abstraction where the program takes both branches and model it with two transitions. This eventually introduces false behaviours, and increase the number of false positives. Nevertheless, it is again an over-approximation and does not bias the correctness argument. If, for example, the predicate in the while loop evaluates while reading some shared variable (e.g. $while(x < 10)$), we would model it as a cascade of a read instruction followed by a branch.

**Goto or jump.** This is only introduced as a convenience to work in conjunction with *branch* and model the control flow.



**Creating a new thread.**
A new thread is associated with a job type and an initial place. Firing the $create$ transition produces a new token that we place in the initial place of the new job type. The calling thread will continue its execution. Note that $same$ and $new$ place can coincide. It would be easy to extend the transitions to multiple arcs.



## 3.3. An example

As the petri net of a concurrent program written in the SML language can quickly grow in size, we only show a short example presented in Figure 3, which models the program from Figure 1.

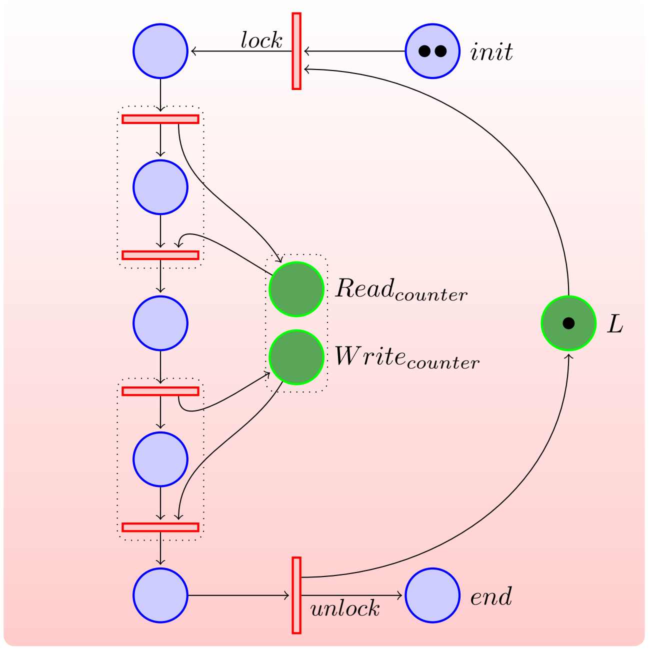
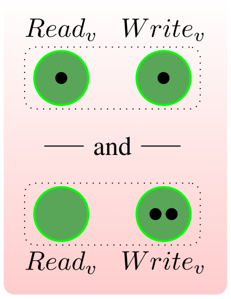


**Figure 3. Modeling the critical section problem with petri nets.**

## 4. Race-Freeness

### 4.1. Bad states and ordering



A data race occurs when multiple threads access a shared variable and at least one has the intention of changing it, without any synchronization constraints (i.e. event ordering). Consequently, given a variable $v$, we notice that a bad state (i.e. a race on $v$) is one configuration that contains at least two tokens in either $Read_v$ or $Write_v$ with one of them in the $Write_v$ place.

We therefore introduce the following (partial) pre-order $\preceq$ on configurations. For two configurations $c$ and $c'$, we say that $c \preceq c'$ if all the places in $c'$ contain more tokens than the respective places in $c$. That is, we can obtain $c$ by removing tokens from $c'$.

For a configuration $c$, we use $\widehat{c}$ to denote the *upward closure* of $c$, i.e. $\widehat{c} = \{c' |\ c \preceq c'\}$. For a set $C$ of configurations, we define $\widehat{C}$ as $\bigcup_{c \in C} \widehat{c}$.

Upward closed sets are attractive to use because they can be characterized by their minimal elements, which often makes it possible to have efficient symbolic representations of infinite sets of configurations.

### 4.2. Safety property

A set $C$ of configurations is said to be *reachable* if $C_{init} \xrightarrow{*} C$. Checking the safety property amounts to performing reachability analysis: is it possible to reach a bad configuration from an initial configuration through a sequence of actions performed by different threads? It can be shown using standard techniques [18, 11], that checking safety properties can be translated into instances of the following coverability problem.

Coverability
**Instance:**

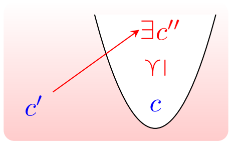
- A set of initial configurations $C_{init}$.
- An upward closed set of bad configurations $C_{fin}$.

**Question:** Is $\widehat{C_{fin}}$ reachable? (i.e. $C_{init} \xrightarrow{*} \widehat{C_{fin}}$ ?)

The main idea is to perform symbolic backward reachability analysis, using upward closed sets, to check for race-freeness for concurrent programs written in the SML language. A negative answer guarantees the absence of races in the program.

## 5. Backward Reachability Analysis

**Computing predecessors.** For a configuration $c$ and a transition $t$, we define $Pre_t(c) = \left\{c' |\ \exists c'' \in \widehat{c}, s.t.\ c' \xrightarrow{t} c''\right\}$.



$Pre_t(c)$ represents the set of configurations that could reach $\widehat{c}$ by firing $t$. Notice that the transition system induced by the petri net is monotonic with respect to $\preceq$. Indeed, consider the configurations $c_1$, $c_2$ and $c_3$, such that $c_1 \xrightarrow{t} c_2$ and $c_1 \preceq c_3$, then there exists a configuration $c_4$ where $c_3 \xrightarrow{t} c_4$ and $c_2 \preceq c_4$. That is to say, if we can fire a transition on a configuration, we can also fire it on a configuration with more tokens in the same places, and the results are also ordered accordingly.

It follows from the anti-symmetry property of $\preceq$ that each upward closed set has a unique generator. Consequently, $Pre_t(c)$ is upward closed and it has a generator. The generator is computed by adding a token to each place in $I(t)$ and by removing a token from each place in $O(t)$ that contained a token (or equivalently removing a token from each output place and resetting to zero the negative values).

We define the backward transition system on configurations, with respect to the pre-order $\preceq$, as follows. For configurations $c_1$ and $c_2$, we say that $c_1 \overset{t}{\rightsquigarrow} c_2$ iff $Pre_t(c_1) = \widehat{c_2}$ and we say that $c_1 \rightsquigarrow c_2$ if $c_1 \overset{t}{\rightsquigarrow} c_2$ for some $t \in T$. Note that it makes all transitions always backwards enabled. We define $Pre(c) = \bigcup_{t \in T} Pre_t(c) = \{c' |\ c \rightsquigarrow c'\}$ and $Pre(C)$ as $\bigcup_{c \in C} Pre(c)$. $Pre(C)$ represents the set of all configurations that could reach $C$ by firing a transition in the petri net. Note that, for a finite set $C$, $Pre(C)$ is an infinite set that can be represented by a finite set of minimal configurations (w.r.t. $\preceq$).

**Algorithm.** Starting from the finite set $C_{fin}$, we define the sequence $I_0, I_1, I_2, \ldots$ of sets by $I_0 = \widehat{C_{fin}}$ and $I_{j+1} = I \cup Pre(I_j)$. Intuitively $I_j$ denotes the set of configurations from which $C_{fin}$ is reachable in at most $j$ steps. Thus if we defined $Pre^*(C_{fin})$ to be $\bigcup_{j \geq 0} I_j$, then $C_{fin}$ is reachable if and only if $C_{init} \cap Pre^*(C_{fin}) \neq \emptyset$.

For a set $A$, we say that the pre-order $\sqsubseteq$ is a *well quasi-ordering (WQO)* on $A$ if the following property is satisfied: for any infinite sequence $a_0, a_1, a_2, \ldots$ of elements in $A$, there are $i, j$ such that $i < j$ and $a_i \sqsubseteq a_j$. Since we have $I_0 \subseteq I_1 \subseteq I_2 \subseteq \ldots$, and the pre-order $\preceq$ is WQO by Dickson's lemma, it follows by [2] that there is a $k$ such that $I_k = I_{k+1}$ and hence $I_l = I_k$ for all $l \geq k$, implying that $Pre^*(C_{fin}) = I_k$. Each $I_j$ is an infinite set, but we know that we can represent it by a finite set of (minimal) configurations.

The algorithm is guaranteed to terminate.

```
Input: Two sets $C_{init}$ and $C_{fin}$ of configurations.
Output: $C_{init} \xrightarrow{*} \widehat{C_{fin}}$ ?

WorkList := $C_{fin}$
Explored := $\emptyset$
while (WorkList $\neq \emptyset$) {
    remove some $c$ from WorkList
    if $\exists c' \in C_{init}, c \preceq c'$ {
        return true
    } else if $\exists c' \in$ Explored, $c' \preceq c$ {
        discard $c$
    } else {
        WorkList := WorkList $\bigcup Pre(c)$
        Explored :=
        $\{c\} \bigcup \{c' | c' \in$ Explored $\wedge (c \npreceq c')\}$
    }
}
return false
```

**Figure 4. Algorithm outline**

```
//Thread A                    // Thread B

pthread_mutex_lock(L);        if(data > 0){
data++;                          /*do this*/
pthread_mutex_unlock(L);      } else {
                                 /*do that*/
                              }
```

```
//Thread A                    // Thread B
                              pthread_mutex_lock(L);
pthread_mutex_lock(L);        if(data > 0){
data++;                          /*do this*/
pthread_mutex_unlock(L);      } else {
                                 /*do that*/
                              }
                              pthread_mutex_unlock(L);
```

**Figure 5. Check then act in pthreaded code. The read in the if statement was not "secured".**

## 6. Experiments

We present a few examples on which we applied the method. As the equivalent petri net for each example can quickly grow, we narrowed down the examples to a characteristic part, namely to test the presence or absence of data races. Some examples are classical examples of synchronization disciplines. We implemented a small prototype in Java and report in table 1 the results of the runs. We display the number of configurations kept at any time in the analysis (#Conf.), the number of configurations that have been subsumed (#Subsum., i.e. discarded by the algorithm because already simulated by other smaller configurations), and the number of iteration of the algorithm (#Iter.). Note that we also tested some examples that did contain a data race. We also added whether the analysis found a race or not. All examples ran in less than a second[2].

The `Counter` and `CounterWithLock` examples represent a shared counter which is incremented as depicted in Figure 1. The `CheckThenAct` and `CheckThenAct-Lock` examples represent a classical race condition described in Section 1 and depicted in Figure 5. Figure 2 depicts the `Prods/Cons` (`Prods/Cons 2` is another variant) as the classical producer/consumer programming style. Note that the `Lock-ReadWriteOnly` example shows a program that secures only the read and write accesses to shared variable. A thread in that program can indeed read a shared variable, store it in a local variable, change the local variable as it

[2]Performance is not exactly the focus in this paper, nor are limitations, but we wanted to show that it was not a bottleneck.

**Table 1. Experimental Results**

| Prog. | #Conf. | #Subsum. | #Iter. | Safe? |
|---|---|---|---|---|
| `Counter` | 10 | 3 | 4 | - |
| `CounterWithLock` | 14 | 7 | 6 | ✓ |
| `CheckThenAct` | 20 | 12 | 5 | - |
| `CheckThenAct-Lock` | 13 | 4 | 4 | ✓ |
| `Prods/Cons` | 310 | 645 | 19 | ✓ |
| `Prods/Cons 2` | 290 | 561 | 17 | ✓ |
| `Lock-ReadWriteOnly` | 25 | 13 | 8 | ✓ |

wishes, and store the result back into the shared variable. While there is no data race in that program, it is not a good programming practice, as the shared variable might have been updated, and the first read of the shared variable does not reflect its actual value. It has indeed been argued in [10] that the absence of data race is not a strong enough condition.

## 7. Conclusion

We have presented a method to model-check race-freeness of programs written in a subset of the C language, using POSIX threads. We model the behaviour of the system as Petri nets and take advantage of upward closure to efficiently represent infinite sets of configurations. Race-freeness is a safety property and we check it using a symbolic backward reachability analysis. It is based on a simple algorithmic principle and is fully automatic and sound. It is moreover guaranteed to terminate for the class of properties we consider.

We plan to extend the model to obtain a closer relationship to pthreaded programs. For instance, we will include waiting for termination of a specific thread and broadcasting of a wake up signal. We also would like to measure the usefulness of the method by examining the amount of false positives and the limitations with respect to the program size. Finally, we would like to report more precise program traces involving race conditions.

## References

[1] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. LICS '96, $11^{th}$ IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.

[2] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.

[3] P. A. Abdulla and A. Nylén. Timed Petri nets and BQOs. In *Proc. ICATPN'2001: 22nd Int. Conf. on application and theory of Petri nets*, volume 2075 of *Lecture Notes in Computer Science*, pages 53 –70, 2001.

[4] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Roby. Bandera: a source-level interface for model checking java programs. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 762–765. ACM, 2000.

[6] G. Delzanno. Automatic verification of cache coherence protocols. In Emerson and Sistla, editors, *Proc. $12^{th}$ Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer Verlag, 2000.

[7] J. Dingel. Computer-assisted assume/guarantee reasoning with verisoft. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 138–148. IEEE Computer Society, 2003.

[8] E. Emerson and K. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. LICS '98, $13^{th}$ IEEE Int. Symp. on Logic in Computer Science*, pages 70–80, 1998.

[9] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. LICS '99, $14^{th}$ IEEE Int. Symp. on Logic in Computer Science*, 1999.

[10] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12. ACM, 2003.

[11] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[12] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13. ACM, 2004.

[13] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[14] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190. ACM, 2003.

[15] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 14–24. ACM, 2004.

[16] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *5th International Symposium on Programming, Turin*, volume 137 of *Lecture Notes in Computer Science*, pages 337–352. Springer Verlag, 1982.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 27–37. ACM, 1997.

[18] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86, $1^{st}$ IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

[19] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234. ACM, 2005.