Algorithmic Verification of Synchronization with Condition Variables

Pedro de Carvalho Gomes¹, Dilian Gurov¹, and Marieke Huisman²

¹ KTH Royal Institute of Technology, Stockholm, Sweden
² University of Twente, Enschede, The Netherlands

Condition variables are a common synchronization mechanism present in many programming languages. Still, due to the combinatorial complexity of the behaviours the mechanism induces, it has not been addressed sufficiently with formal techniques. In this paper we propose a fully automated technique to prove the correct synchronization of concurrent programs synchronizing via condition variables, where under correctness we mean the liveness property: "If for every set of condition variables, every thread synchronizing under the variables of the set eventually enters its synchronization block, then every thread will eventually exit the synchronization".

First, we introduce SyncTask, a simple imperative language to specify parallel computations that synchronize via condition variables. Next, we model the constructs of the language as Petri Net components, and define rules to extract and compose nets from a SyncTask program. We show that a SyncTask program terminates if and only if the corresponding Petri Net always reaches a particular final marking. We thus transform the verification of termination into a reachability problem on the net, which can be solved efficiently by existing Petri Net analysis tools. Further, to relieve the programmer from the burden of having to provide specifications in SyncTask, we introduce an economic annotation scheme for Java programs to assist the automated extraction of SyncTask programs capturing the synchronization behaviour of the underlying program. We show that, for the Java programs that can be annotated according to the scheme, the above-mentioned liveness property holds if and only if the corresponding SyncTask program terminates. Both the SyncTask program extraction and the generation of Petri Nets are implemented in the STaVe tool. We evaluate the proposed verification framework on a number of test cases.

1 Introduction

Condition variables (CV) are a synchronization mechanism to coordinate multithreaded programs. Threads wait on a conditional variable, meaning they suspend their execution, until another thread notifies the CV, causing the waiting threads to resume their execution. The signaling is asynchronous: if no thread was waiting on the CV, then the notification has no effect. CVs are used in conjunction with locks: a thread must acquire the associated lock for notifying or waiting on a CV, and if notified, must reacquire the lock.

Many widely used programming languages feature condition variables. In Java, for instance, condition variables are provided both natively as an object’s
monitor \[8\], i.e., a pair of a lock and a CV, and in the concurrent API, as one-to-many Condition objects associated to a Lock object. Nevertheless, condition variables have not been addressed sufficiently with formal techniques, mainly because of the complexity of reasoning about asynchronous signaling. For instance, Leino et al. \[18\] acknowledge that verifying the absence of deadlocks when using condition variables is hard because a notification is “lost” if no thread is waiting on it. Thus, one cannot reason locally whether a waiting thread will eventually be notified. The correct usage of CVs involves both control-flow and data-flow aspects, and directly depends on the global thread composition, i.e., the type and quantity of threads executing in parallel.

In this work, we present a formal technique for verifying the synchronization of multithreaded programs with CVs, and its implementation as the SyncTask Verifier (STaVe). The synchronization property of interest is the following: “If for every set of condition variables, every thread synchronizing under the variables of the set eventually enters its synchronization block, then every thread will eventually exit the synchronization block”. To the best of our knowledge, the present work is the first to address a liveness property involving CVs. As the verification of such properties is undecidable in general, to stay within a decidable fragment, we limit our technique to programs with bounded data domains and numbers of threads. Still, the verification problem is subject to a combinatorial explosion of thread interleavings. Our technique alleviates the state-space explosion by isolating the relevant aspects of the synchronization.

First, we study the liveness property in the context of a synchronization specification language. To this end, we introduce SyncTask, a simple concurrent programming language where all computations occur inside synchronized code blocks. It has been designed to capture common patterns of CV usage, while abstracting away from all irrelevant details. SyncTask has a Java-like syntax and semantics, and features the relevant constructs for synchronization, such as locks, CVs, conditional statements, and arithmetic operations. However, it is non-procedural, data types are bounded, and it does not allow dynamic thread creation. These restrictions render the state-space of SyncTask programs finite, and make the termination problem decidable.

Next, we translate SyncTask programs into hierarchical Coloured Petri Nets (CPNs) \[9\]. These extend the basic Petri Nets with data and modularity. We have chosen CPN as the underlining program model for several reasons. Petri nets provide a good balance between expressiveness and analizability, and allow a concise modeling of a thread’s control flow. The model has been successfully used over the last decades for the modelling of concurrent systems, is theoretically well-developed, and is supported by a set of mature analysis tools.

We model the constructs of SyncTask as CPN components, and describe how to extract CPNs automatically from SyncTask programs. Then, we establish that a SyncTask program terminates if and only if the extracted CPN always reaches dead markings (i.e., CPN configurations without successors) where the tokens representing the threads are in a unique end place. In this way we transform the problem of termination of SyncTask programs into the computation of the
reachability graph of Petri Nets, and the check that: (i) there are no cycles in the graph (meaning unconditional termination), and (ii) the only dead markings are those where the end place contains all thread tokens. The complexity of these checks is linear in the size of the reachability graph, which can be computed efficiently by standard CPN analysis tools such as PIPE [6] or CPN Tools [12]. Also, in case that the condition does not hold, an inspection of the reachability graph easily provides the cause of the non-termination.

Then, we address the problem of verifying the correct synchronization of programs written in real concurrent programming languages by showing how SyncTask can be used to verify the correct usage of CVs in Java programs, if these are bounded. There is a consensus in Software Engineering that the synchronization must be as minimal as possible, both to minimize the risk of error conditions and to avoid the latency of blocking threads. As a consequence, many programs present a finite (though arbitrarily large) synchronization behaviour.

The analysis of synchronization in Java programs is undecidable in general. We therefore introduce an annotation scheme to model the expected synchronization using CVs of Java programs, and thus to assist STaVe to automatically extract SyncTask programs. For instance, the user must annotate the global thread composition, and provide the initial state of the variables accessed inside the synchronized blocks. The annotations allows us to define an automatic algorithm to extract SyncTask programs from the Java program. Finally, we establish that for the Java programs that can be annotated to define a SyncTask program, the liveness property discussed above is equivalent to termination of the respective SyncTask program, provided that the annotations are correct.

Figure 1 summarizes our approach.

![Diagram](image.png)

**Fig. 1: Scheme to prove the correct usage of CVs**

We have implemented both the extraction of SyncTask programs from annotated Java source code, and the translation from SyncTask programs to CPNs as the SyncTask Verifier (STaVe) tool. We validate STaVe on two test-cases, by generating CPNs from annotated Java programs and analyzing these with CPN Tools. The first test-case evaluates the scalability of the tool w.r.t. the number
of synchronizing threads. It is an implementation of a shared buffer, for which we performed experiments with different numbers of threads and buffer sizes. The results show the expected exponential blow-up of the state-space, but we were still able to analyze the synchronization of several dozens of threads. The second test-case evaluates the scalability of the tool w.r.t. the size of program code that does not affect the synchronization behaviour of the program. For this we annotated the Java source code of PIPE [6], another CPN analysis tool that is large, but exhibits a simple synchronization behaviour.

The remainder of the paper is as follows. Section 2 introduces the concepts of hierarchical Coloured Petri nets. Section 3 illustrates STaVe’s work-flow with a concrete Java example. SyncTask is defined in Section 4. Section 5 describes the mapping from annotated Java to SyncTask programs, Section 6 presents its translation into CPNs. Section 7 presents the correctness arguments. Section 8 presents test-cases. We discuss related work in Section 9, while Section 10 concludes and describes future work.

2 Hierarchical Coloured Petri Nets

We now introduce Coloured Petri Nets, the model used in the verification technique presented in this paper for checking the correct synchronization with condition variables. The technique and theoretical results are presented informally. Thus, here we briefly describe and illustrate the CPN concepts, and refer to [11, Chapters 4,6] for the formal definitions.

Petri Nets (PN) are bipartite directed graphs where nodes are either places, visually represented as ellipses, or transitions, represented as rectangles. Arcs connect places to transitions, and vice versa. A place contains a non-negative number of tokens, which we refer to as its marking. A PN configuration consists of a distribution of tokens over the places, and it is also commonly referred to as a net marking. We remove the ambiguity along the thesis, and use the term ‘marking’ only when referring to places.

A transition is enabled if all its incoming places are marked, i.e., there is at least one token in all its input places. An enabled transition may fire, i.e., atomically consume tokens from the input places and produce tokens on the output places. The choice of which of the enabled transitions fires is non-deterministic.

Inhibitor arcs extend PNs by enabling a transition if their incoming places are empty. Or equivalently, they ‘inhibit’ a transition if they are marked by a token. These arcs are useful for testing emptiness of a place, meaning, for example, the exhaustion of some resource. Inhibitor arcs are depicted with a bubble, instead of an arrow. PNs with inhibitor arcs are more expressive, and reachability becomes generally undecidable. However, if a PN with inhibitor arcs is bounded, meaning that there exists a finite upper-bound for the tokens in the net, then reachability is still decidable [2]. In this thesis we extract bounded nets; thus we still stay in the decidable fragment.

Hierarchical Coloured Petri Nets [9] extend plain PNs with data. They declare colour sets, and assign one for each place. Transitions are enabled if all input
Algorithmic Verification of Synchronization with Condition Variables

places contain at least one token of the same colour as the incoming arc. CPNs generalize standard PNs. That is, a PN is simply a CPN with a single colour.

CPNs provide a modular concept called subpage for the declaration and instantiation of components, which is analogous to subroutines in procedural languages. A subpage receives and returns tokens on its in- and out-port places, similarly to a procedure receiving parameters and returning a value. We depict ports as doubly outlined ellipses with the direction indicated on the lower corner. The instantiation of a subpage is modelled as a substitution transition (ST), and is analogous to a procedure invocation. It has incoming and outgoing arcs from and to its in- and out-socket places, which are assigned to matching in- and out-ports, respectively, in the subpage. STs are depicted as doubly outlined rectangles, with instantiated subpage name on the bottom.

Finally, fusion places are another modular concept, which are analogous to global variables. These enable the instantiation of the same place in several substitution transitions, and are graphically represented with a centered rectangle with the place’s name.

Example 1 (Hierarchical Coloured Petri Net). Figure 2 shows a hierarchical CPN net that models a flow of passengers on an airplane. Passengers are separated into two categories: VIP, which have higher priority to board, and NORMAL, which have low priority. Upon boarding, the airline increments the number of passengers, so it can provide the exact amount of meals. After boarding, there is no distinction of treatment between the passengers w.r.t. being served a meal, or exiting the plane. Despite being an over-simplified model, the CPN contains all concepts mentioned above, and we now explain them.

The CPN contains two pages. The first is the Queue top page, which defines the queuing and boarding of passengers. The services provided to the passengers after they have boarded are modelled with the substitution transition Service, which has Boarded and Landed as in- and out-sockets, respectively. The subpage Service models the passengers’ service and landing. For simplicity, the subpage has been named to the ST that instantiates it, just like its in- and out-ports have been named to the in- and out-sockets that they are assigned to.

The colour set CLIENT defines the two passenger types. VIP passengers queue in the High Priority place and are initially fifteen, as denoted by its marking on the top; NORMAL passengers are queued in the Low Priority place, with marking containing fifty five NORMAL tokens. The Board High transition is enabled as long as there are tokens in High Priority, while Board Low is disabled by the inhibitor arc. Moreover, whenever either of the transitions is enabled, it adds a token of colour FOOD to the fusion place Meals. Notice that the place is present in both Queue and Service, and represents the same entity. That is, the addition or removal of a token from Meals in one of the pages is reflected in the other.
3 Overview of the Approach

In this section we illustrate our verification method by presenting the artifacts that STAVe manipulates: an annotated Java program, its corresponding SyncTask program, and Coloured Petri Net. We then describe the CPN analysis.

The Java program in Figure 3 implements a shared Buffer, Producer and Consumer threads synchronize via the implicit monitor associated with the buffer object b to add or remove elements, and wait if the buffer is full or empty, respectively. Waiting threads are woken up by notifyAll after an operation is performed on the buffer, and compete for the monitor to resume execution.

The annotations are provided in comment blocks, and delimit the expected synchronization. The @syncblock annotations include the synchronized blocks to the observed synchronization behaviour, and @monitor and @resource map local references to global aliases. The annotation @resource above Buffer starts the definition of a resource type, i.e., an abstraction of a data type that is accessed in the synchronization. @value, @object and @capacity define the resource’s abstract state, and @operation and @predicate define how the class methods operate on the state. The @syncTask annotation above main starts the declaration of locks, CVs and resources, and @thread annotations add the following objects to the global thread composition.

The SyncTask program in Figure 4 was automatically extracted by STAVe from the Java program in Figure 3. The two thread types, Consumer and Producer,
Algorithmic Verification of Synchronization with Condition Variables

```java
01 class Producer extends Thread {
    Buffer buffer;
    Producer(Buffer b){buffer=b;}
    public void run() {
        /*@syncblock
         *@monitor buffer -> m
        */
        synchronized(buffer) {
            while (buffer.full())
                buffer.wait();
            buffer.add();
            buffer.notifyAll();
        } }
    } }

03 class Consumer extends Thread {
    Buffer buffer;
    Consumer(Buffer b){buffer=b;}
    public void run() {
        /*@syncblock
         *@monitor buffer -> m
        */
        synchronized(buffer) {
            while (buffer.empty())
                buffer.wait();
            buffer.remove();
            buffer.notifyAll();
        } }
    } }

05 /*@resource @capacity cap
    @object els->b_els
    @value els->b_els */
    class Buffer {
        final int cap;
        void remove(){if (els>0)els--;}
        void add(){if (els<cap)els++;}
        boolean full(){return els==cap;}
        boolean empty(){return els==0;}
    } }

07 static void main(String[] s) {
    Buffer b = new Buffer();
    b.els = 1; b.cap = 1;
    Consumer c1 = new Consumer(b);
    Consumer c2 = new Consumer(b);
    Producer p = new Producer(b);
    c1.start();p.start();c2.start();
    } }

15 class Consumer extends Thread {
    Buffer buffer;
    Consumer(Buffer b){buffer=b;}
    public void run() {
        /*@syncblock
         *@monitor buffer -> m
        */
        synchronized(buffer) {
            while (buffer.empty())
                buffer.wait();
            buffer.remove();
            buffer.notifyAll();
        } }
    } }

17 /*@resource @capacity cap
    @object els->b_els
    @value els->b_els */
    class Buffer {
        final int cap;
        void remove(){if (els>0)els--;}
        void add(){if (els<cap)els++;}
        boolean full(){return els==cap;}
        boolean empty(){return els==0;}
    } }

19 static void main(String[] s) {
    Buffer b = new Buffer();
    b.els = 1; b.cap = 1;
    Consumer c1 = new Consumer(b);
    Consumer c2 = new Consumer(b);
    Producer p = new Producer(b);
    c1.start();p.start();c2.start();
    } }

21 /*@resource @capacity cap
    @object els->b_els
    @value els->b_els */
    class Buffer {
        final int cap;
        void remove(){if (els>0)els--;}
        void add(){if (els<cap)els++;}
        boolean full(){return els==cap;}
        boolean empty(){return els==0;}
    } }
```

Fig. 3: Annotated Java program synchronizing via shared buffer
Fig. 4: SyncTask program extracted from annotated Java

```java
1 Thread Producer {
   synchronized(m_lock){
      while(b_els==max(b_els))
         wait(m_cond);
      if(b_els<max(b_els))
         b_els=(b_els+1);
      else
         skip;
      notifyAll(m_cond);
   }
}

11 Thread Consumer {
   synchronized(m_lock){
      while((b_els==0))
         wait(m_cond);
      if((b_els>0))
         b_els=(b_els-1);
      else
         skip;
      notifyAll(m_cond);
   }
}

21 main {
   Lock m_lock();
   Cond m_cond(m_lock);
   Int b_els(0,1,1);
   start(1,Producer);
   start(2,Consumer);
}
```

same name. The WhileProducer0 subpage (Figure 5d) presents the component for while. Among others, it contains the buffer_els fusion place (also present in SyncTask0), representing the global variable buffer_els. It has colour INT0, denoting the variable bounds, and contains one token, representing that the variable is initially full.

The hierarchical version is composed of twenty one subpages. This leads to lots of redundant graphical elements, e.g., many instantiation of fusion places, and the layout of all subpages becomes cumbersome. Thus, we also present the complete non-hierarchical version in Figure 6 to give an intuitive notion of the program model. It is important to stress, however, that the hierarchical and non-hierarchical representations of CPNs are semantically equivalent.

We generate the non-hierarchical net by replacing syntactically the STs with their respective subpages. That is, we expand a subpage inside its parent page, and collapse the in-ports and in-sockets, and out-ports and out-sockets. Notice that the inverse process of moving sup-parts of a CPN to subpages, and representing them with STs is also possible. We preserve the notation for fusion places to help the reader identifying these places in the hierarchical model. However, we typically collapse all instantiations of a fusion place into a single normal place in a non-hierarchical net.

The analysis of the net (which we explain in Section 8) shows that there are no cycles in its reachability graph, and it has a single dead configuration, with the marking of End being three thread tokens. Thus, the program terminates for the given initial values.

4 SyncTask

SyncTask abstracts from most features of full-fledged programming languages. For instance, it does not have objects, procedures, exceptions, etc. However, it features the relevant aspects of thread synchronization. We now describe the language syntax, types, and semantics.
The SyncTask syntax is presented in Figure 7. A program has two main parts: \textit{ThreadType}, which declares the different types of parallel execution flows, and \textit{Main}, which contains the variable declarations and initializations and defines how the threads are composed, i.e., it statically declares how many threads of each type are spawned.

Each \textit{ThreadType} consists of one or more adjacent \textit{SyncBlocks}, which are mutually exclusive code blocks, guarded by a lock. A code block is defined as a sequence of statements, which may even be another \textit{SyncBlock}. Notice that this
allows nested SyncBlocks, thus enabling the definition of complex synchronization schemes with more than one lock.

There are four primitive types: booleans (Bool), bounded integers (Int), reentrant locks (Lock), and condition variables (Cond). Expressions are evaluated as in Java. The boolean and integer operators are the standard ones, while max and min return a variable’s bounds. Operations between integers with different bounds (overloading) are allowed. However, an out-of-bounds assignment leads the program to an error configuration.

Condition variables are manipulated by the unary operators wait, notify, and notifyAll. Currently, the language provides only two control flow constructs: while and if-else. These suffice for the illustration of our technique, while the addition of other constructs is straightforward.
The Main block contains the global variable declarations with initializations (VarDecl*), and the thread composition (StartThread*). A variable is defined by its type and name, followed by the initialization arguments. The number of parameters varies per type: Lock takes no arguments; Cond is initialized with a lock variable; Bool takes either a true or a false literal; Int takes three integer literals as arguments: the lower and upper bounds, and the initial value, which must be in the given range. Finally, start takes a positive number and a thread type, signifying the number of threads of that type it spawns.

4.2 Structural Operational Semantics

We now describe the structural operational semantics of SyncTask, to provide the means for establishing a formal relationship between the language and the proposed verification mechanism.

The semantic domains are defined as follows. Booleans are represented as usual. Integer variables are triples $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, where the first two elements are the lower and upper bound, and the third is the current value. A lock $o$ is a pair $\langle \text{Thread id} \cup \{\bot\} \rangle \times \mathbb{N}$ of the id of the thread holding the lock (or $\bot$, if none), and a counter of how many times it was acquired. A condition variable $d$ simply stores its respective lock, which is retrieved with the auxiliary function lock$(d)$.

SyncTask contains global variables only and all memory operations are synchronized. Thus, we assume the memory to be sequentially consistent [15]. Let $\mu$ represent a program’s memory. We write $\mu(l)$ to denote the value of variable $l$, and $\mu[l \mapsto v]$ to denote the update of $l$ in $\mu$ with value $v$.

A thread state is either running ($R$) if the thread is executing, waiting ($W$) if it has suspended the execution on a CV, or notified ($N$) if another thread has woken up the suspended thread. The states $W$ and $N$ also contain the CV a thread is/was waiting on, and the number of locks it must reacquire to proceed with the execution. The auxiliary function waitset($d$) returns the id’s of all threads waiting on a CV $d$.

We represent a thread as $(\theta, t, X)$, where $\theta$ denotes its id, $t$ the executing code, and $X$ its state. We write $T = (\theta_i, t_i, X_i)|(\theta_j, t_j, X_j)$ for a parallel thread composition, with $\theta_i \neq \theta_j$. Also, $T|(\theta, t, X)$ denotes a thread composition, assuming that $\theta$ is not defined in $T$. For convenience, we abuse set notation to denote the composition of threads in the set; e.g., $T_W = \{ (\theta, t, W, d, n) \}$ represents the composition of all threads in the wait set of $d$. A program configuration is a pair $(T, \mu)$ of the threads’ composition and its memory. A thread terminates if the program reaches a configuration where its code $t$ is empty ($\epsilon$); a program terminates if all its threads terminate.

The initial configuration is defined by the declarations in Main. As expected, the variable initializations set the initial value of $\mu$. For example, Int $i(lb, ub, v)$ defines a new variable such that $\mu(i) = (lb, ub, v)$, $lb \leq v \leq ub$, and Lock $o$ initialized a lock $\mu(o) = (\bot, 0)$. The thread composition is defined by the start declarations; e.g., start$(2, t)$ adds two threads of type $t$ to the thread composition: $(\theta, t, R)|(\theta', t, R)$.
For readability, we just present the rules for the synchronization state and increases the lock counter. Both rules replace \(\text{synchronized}\) any other thread and the counter is zero. Rule \([s2]\) represents lock reentrancy with an empty wait set; the behaviour is the same as for the \(\text{CV}\)'s lock, and releases it. The rules \([nf1]\) and \([na1]\) apply when a thread notifies a \(\text{CV}\) with \(\mu_\text{lock}(d) = 0\). By rule \([rd]\), a thread reacquires all the locks it had relinquished, changes the state to \(\text{R}\), and resumes the execution after the control point where it invoked \(\text{wait}\).

Figure 8 presents the operational rules, with superscripts \(a-h\) denoting conditions. For readability, we just present the rules for the synchronization statements, as the rules for the remaining statements are standard (see [5], § 3.4-8]).

In rule \([s1]\), a thread acquires a lock, if available, i.e., if it is not assigned to any other thread and the counter is zero. Rule \([s2]\) represents lock reentrancy and increases the lock counter. Both rules replace \(\text{synchronized}\) with a primed version to denote that the execution of synchronization block has begun. Rule \([s3]\) applies to the computation of statements inside synchronized blocks, and requires that the thread holds the lock. Rule \([s4]\) preserves the lock, but decreases the counter upon exiting a synchronized block. In rule \([s5]\), a thread finishes the execution of a synchronized block, and relinquishes the lock.

In the \([wt]\) rule, a thread changes its state to \(W\), stores the counter of the \(\text{CV}\)'s lock, and releases it. The rules \([nf1]\) and \([na1]\) apply when a thread notifies a \(\text{CV}\) with an empty wait set; the behaviour is the same as for the \(\text{skip}\) statement. By rule \([nf2]\), a thread notifies a \(\text{CV}\), and one thread in its wait set is selected non-deterministically, and its state is changed to \(N\). Rule \([na2]\) is similar, but all threads in the wait set are awaken. By the rule \([rd]\), a thread reacquires all the locks it had relinquished, changes the state to \(\text{R}\), and resumes the execution after the control point where it invoked \(\text{wait}\).
Finally, we define a SyncTask program to have a correct synchronization iff it terminates.

5 From Java To SyncTask

We now present the annotation language for delimiting the bounded synchronization behaviour of Java programs. It relies on the knowledge about the expected synchronization, and the programmer provides hints for STaVe to automatically map the synchronization to a SyncTask program.

The annotations are provided in a tree structure, which follows the Java abstract syntax tree (AST). An annotation binds to a specific set of AST nodes. That is, the declaration starts in a comment block immediately above the node declaration, with additional annotations inside the node’s body. Annotations share common keywords (though with a different semantics), and overlap in the node types they may bind to. The ambiguity is resolved by the first keyword (called a switch) found in the comment block. Comments that do not start with a keyword are ignored.

Figure 9 presents the annotation language. The text inside square brackets is an optional argument, and the text inside parentheses tells which Java AST node the annotation binds to. The top-level annotations are divided into three categories: resource, synchronization and initialization.

A resource is a data type that is manipulated in the synchronization. It abstracts the state of a data structure to a bounded integer, which is potentially a ghost variable (as in [16]), and defines how the methods operates on it. For example, the annotation abstracts a linked list or a buffer (as in Figure 3) by its size. In case a resource is mapped into a ghost variables, we say that the variable extends the program memory. Resources bind to classes only, and the

---

**Resource annotation:**

```plaintext
@resource (classes)
@object [Id -> Id]
@value [Id -> Id]
@capacity [Id -> Id]
@defaultval Int
@defaultcap Int
@predicate (methods)
@inline [@maps Id -> @
@code -> @
@operation (methods)
@inline [@maps Id -> @
@code -> @
```

**Synchronization annotation:**

```plaintext
@syncblock [Id] (synchronized blocks)
@threadtype Id -> Id
@resource Id : ResourceId
@lock Id -> Id
@condvar Id -> Id
@monitor Id -> Id
```

**Initialization annotation:**

```plaintext
@synctask [Id] (methods)
@resource Id -> Id
@lock Id -> Id
@condvar Id -> Id
@monitor Id -> Id
@thread [Int : Id]
```

---

Fig. 9: Annotation language from Java programs
switch @resource starts the declaration. @value and @capacity define, respectively, which class member, or ghost variable, stores the abstract state, and its maximum value. The keyword @operation binds to method declarations, and expresses that the method potentially alters the resource state. For example, that is the case for the methods add and remove in Figure 3. Similarly, @predicate binds to methods, defines that the method returns a predicate about the state, and is exemplified with methods empty and full.

There are two ways to synthesize the annotated method’s behaviour. @code tells STaVe not to process the method, but instead to associate it to the code enclosed between @ and }. @inline tells STaVe to try to infer the method declaration, with the potential aid of @maps, which syntactically replaces a Java node (e.g., a method invocation) with a SyncTask code snippet. The above-mentioned methods from Figure 3 exemplify the annotation, with STaVe automatically inlining them in the SyncTask program in Figure 4.

The synchronization annotation defines the observation scope. It binds to synchronized blocks and methods, and the switch @syncblock starts the declaration. Nested synchronization blocks and methods are not annotated; all its information is defined in the top-level annotation. The keywords @lock and @condvar define which mutex and condition object to observe. @monitor has the combined effect of both keywords for an object’s monitor, i.e., a pair of a lock and a CV. Here, @resource maps a local variable to the alias of the global object being observed.

Initialization annotations define the global pre-condition for the elements involved in the synchronization, i.e., they define the lock, condition variable and resource declarations with initial value, and the global thread composition. It binds to methods, and the switch @synctask starts the declaration. Here, @resource, @lock, @condvar and @monitor define the objects being observed, and assign global aliases to them. Finally, @thread defines that the following object corresponds to a spawned thread that synchronizes within the observed synchronization objects.

6 From SyncTask to CPNs

Next, we present the modelling of SyncTask constructs as CPN components (following definitions from Section 2), and describe how the net is assembled. Our extraction of hierarchical CPN from SyncTask programs is a variant of the one described in [21].

In our modelling, the colour set THREAD associates a colour to each Thread type declaration, and an individual thread is represented by a token with a colour from the set. Some components are parametrized by THREAD, meaning that they declare transitions, arcs, or places for each thread type. For illustration purposes, we present the parametrized components in an example scenario with three thread types: blue (B), red (R), and yellow (Y). The production rules in Figure 7 are mapped into hierarchical CPNs components, where STs represent the non-terminals on the right-hand side.
Figure 10a shows the top-level component for SyncTask programs. It has the in-socket Start, which contains all threads tokens in the initial configuration, connected by arcs (one per colour) to the STs denoting the thread types, and the out-socket End, which collects the terminated thread tokens. It may also contain the fusion places that represent global variables, as buffer els in Figure 5a. Each declaration in ThreadType* (Figure 10b) generates a distinct component representing a thread’s control flow. The ST named s1 instantiates a SyncBlock* declaration, with the in-port assigned to Start, and the out-port assigned to End. The sequential composition of consecutive SyncBlocks, and of statements inside a Block declaration is presented in Figure 10c.

Figure 11 shows the components for the control flow statements. An if-else statement is modelled by an in-port connected to two transitions, each denoting the evaluation of the control expression to true or false, followed by an in-socket to the respective ST denoting the respective ‘then’ or ‘else’ block, and arcs connecting to out-port. The while statement is modelled by an in-port, denoting a control point immediately before the expression evaluation, connected to two transitions: one is enabled if the expression is false, followed by an out-port denoting the control point after the loop; the other one is enabled if the expression is true, followed by an in-socket to the ST denoting the loop body, and an arc to the in-port, denoting expression re-evaluation.
Figure 11 shows the components for the synchronization primitives. A `SyncBlock` is modelled with a single in-port, a transition denoting lock acquisition, an in-socket denoting the critical section entrance, a ST denoting the body declaration, a transition denoting the lock release, and an out-socket denoting the exit from the critical section. A `wait` is modelled as a transition that produces two tokens: one into the place modelling the CV, and one into the place modelling the lock, representing its release; a place for the woken-up threads, and a transition to reacquire the lock, and an out-port, denoting the control point where threads resume the execution. A `notify` is modelled by a transition that is enabled if the CV is empty, plus one transition and one out-port per colour, modelling the non-deterministic choice of which thread to wake, and the routing of tokens to the place to reacquire the lock. A `notifyAll` is similar, but the transition that checks if the CV is empty is enabled after all thread tokens have been woken up.

CPN Tools is integrated with an ML-based engine [12] for expressions evaluation, analogously to model checkers and SMT-solvers. Thus, in the current modelling, boolean and integer expressions are conveniently translated to ML expressions, and assigned to transitions (for branching) and arcs (for assignments).

The global variable declaration `VarDecl*` generates a place containing a single token for each `Lock` object. An empty place denotes that some thread holds the lock. We define the colour set `CPOINT` with colours representing the control points with a `wait` statement. A `Condition` variable generates an empty place denoting the waiting set, with colour set `CONDITION`. Here, colours are pairs of `THREAD` and `CPOINT`. Both data are necessary to route correctly woken-up threads to the correct place where they resume execution. A `Bool` variable generates a place with colour set `BOOL`. An `Int` variable generates a place and
a new colour set of bounded integers, with colours being the integer numbers within the variable’s range.

The initialization in `main` does not produce places or transitions. It simply declares the initial set of tokens for the places representing variables, and the number and colours of thread tokens. As seen in the `Start` place in Figure 5, a marking is depicted textually on top of the place. It declares pairs of tokens and colours, with `++` being the separator.

7 Correctness Arguments

The synchronization property of interest here is that “every thread synchronizing under a set of condition variables eventually exits the synchronization”. We work under the assumption that every such thread eventually reaches its synchronization block. There exist techniques (such as [19]) for checking the liveness
property that a given thread eventually reaches a given control point; checking validity of the above assumption is therefore out of the scope of the present work.

The following definition of correct synchronization applies to a one-time synchronization of a Java program. However, if it can be proven that the initial conditions are the same every time the synchronization scheme is spawned, then the scheme is correct for an arbitrary number of invocations. This may be proven by showing that a Java program always resets the variables observed in the synchronization before re-spawning the threads.

**Definition 1 (Synchronization Correctness).** Let \( P \) be a Java program with a one-time synchronization such that every thread eventually reaches the entry point of its synchronization block. We say that \( P \) has a correct synchronization if every thread eventually reaches the first control point after the block.

We defined both synchronization correctness and the termination of the corresponding SyncTask program relative to the correctness of the annotations provided by the programmer. We conjecture that STaVe can be integrated with a suitable functional verification tool to check the correctness of the annotations. Further, we assume the memory model of synchronized actions in a Java program to be sequentially consistent. Again we rely on an external tool to inspect that this property is not violated, for instance, to check that, for a given a set of locks, an observed variable is only accessed by a thread holding all locks in this set.

We now connect synchronization schemes of annotated Java programs with SyncTask programs. We shall assume that the programmer has correctly annotated the program, identifying its threads and synchronization artifacts as described earlier.

**Theorem 1 (SyncTask Extraction).** A correctly annotated Java program has a correct synchronization if its corresponding SyncTask terminates.

**Proof.** Let \((T, \mu)\) be a configuration of a SyncTask program, where \(T\) is the thread composition, and \(\mu\) is the memory. Also, \(T(\theta) = (t, X)\) represents a thread \(\theta\) with code \(t\) and state \(X\), as defined in Section 4.2.

Let \((\tilde{T}, \tilde{\mu})\) be a configuration of an annotated Java program, where \(\tilde{T}\) is the thread composition, and \(\tilde{\mu}\) is the extended memory of an annotated Java program with (potential) ghost variables. Also, let \(\tilde{T}(\theta) = (t, X, \sigma)\) represent a thread as described above, plus a stack \(\sigma\). Upper bars are used here to stress that a definition refers to a Java program.

This definition of configuration is a simplification of the one introduced in [5, § 3.3]. We have assumed sequential consistency. Thus, we abstract from their parametric notion of event space (depicted with \(\eta\)) to instantiate different memory models. As a consequence, updates that were previous parametrized in \(\eta\), such as lock acquisitions and releases, are now represented directly in the memory.

Let \(S\) be the function that extracts a SyncTask program from an annotated Java program. Also, let \(\text{dom}\) return the domain of a function. E.g., \(\text{dom}(\mu)\) is
the set of variables for which $\mu$ is defined. We define the abstraction function $\alpha(\bar{\mu})$ as the projection of $\bar{\mu}$ containing exactly the same variables as the ones in $\mu$. I.e., $(\alpha \circ \bar{\mu})(l) \iff \mu(l), \forall l \in \text{dom}(\mu)$. We define the set of visible actions with the ones where variables in $\alpha(\bar{\mu})$ are updated; other transitions are silent.

We define the binary relation $R$ as a set of pairs of an annotated Java program configuration, and a SyncTask program configuration, as presented in Definition 2, and claim that it is a weak bisimulation in the standard fashion: for a pair of configurations, the common variables between the two programs always contain the same value. Moreover, whenever there is a transition that updates a common variable in one of the programs, there is also a transition in the other program that updates the same variable, and again the common variable has the same value in both programs.

**Definition 2 (Relation between annotated Java and SyncTask).**

\[
R \overset{\text{def}}{=} \{(\bar{T}, \bar{\mu}), (T, \mu) \mid \forall \theta \in \text{dom}(T), T(\theta) = (t, X, \sigma), T(\theta) = (S(\bar{t}), X) \land \forall l \in \text{dom}(\mu), (\alpha \circ \bar{\mu})(l) = \mu(l)\}
\]

Intuitively, the annotations define a bidirectional mapping from (some of the) program variables and ghost variables into equivalent bounded variables in SyncTask. Thus, to show that $R$ is a weak bisimulation it requires to show that: (I) the initial values of variables in $\text{dom}(\mu)$ are the same for $\bar{\mu}$ and $\mu$, and (II) assuming that variables in $\text{dom}(\mu)$ are only updated inside annotated synchronized blocks, prove that any operation that updates a common variable yields the same result for the Java program, and for the SyncTask program.

To prove (I) it is sufficient to show that the initial values in the Java program are the same as the ones provided in the initialization annotation, as described in 5. As we mention above, we assume that annotations are correct, i.e., that the initial values are the same for both the Java and SyncTask programs. Nevertheless, existing techniques [17] and tools [18], can establish that the initial values are the same.

The proof of (II) requires to show that updates in a common variable yield the same value for both programs. A full proof goes by case analysis on the Java and SyncTask instructions sets. Each case must show that, for a configuration pair where $\alpha(\bar{\mu}) = \mu$, the operational rules for a given Java instruction $t$, and for the corresponding SyncTask instruction $S(\bar{t})$ will lead to configurations with $\mu'$ and $\mu$, s.t. $\alpha(\bar{\mu'}) = \mu'$. As mention above, the operational semantics presented in Section 4 for SyncTask has been defined to mimic closely the semantics defined in [5] for Java, and presentation of proof cases is straightforward. Nevertheless, the Java instructions set is large, and the listing of all cases is long and tedious. Thus, we show two cases, and leave the complete analysis as future work.

**Case 1. notify.**

By [5, § 3.10], there are two\(^3\) operational rules for notify: [notify2], when the wait set for $d$ contains at least one thread, and [notify3], when the set is empty. We show the case for [notify2]; the other is analogous.

\(^3\) [notify1] applies to exceptional flow, and is thus ignored.
Let \((\bar{T}, \bar{\mu})\) be a configuration of an annotated Java program, and \((T, \mu)\) be a configuration of the corresponding SyncTask s.t. \(\alpha(\bar{\mu}) = \mu\). Also, let there be two threads \(\theta, \theta'\) s.t. the thread composition is as follows:

\[
\bar{T} = \ldots |(\theta, d.\text{notify}(d), R, \bar{\sigma})|((\theta', \bar{\nu}, (W, d, n)), \bar{\sigma}')
\]

Then, since \(S(d.\text{notify}(d)) = \text{notify}(d)\) there’s also a thread composition for the SyncTask program as follows:

\[
T = \ldots |(\theta, \text{notify}(d), R)|((\theta', S(\bar{\nu}'), (W, d, n))
\]

We have that \((\bar{T}, \bar{\mu}), (T, \mu) \in \mathcal{R}\) since \(\bar{T}(\theta) = (\bar{\nu}, X, \bar{\sigma})\) and \(T(\theta) = (\bar{\nu}, X, \bar{\sigma})\) for \(\bar{\nu} = d.\text{notify}(d)\) and \(X = R\); also \(\bar{T}(\theta') = (\bar{\nu}', X', \bar{\sigma}')\) and \(T(\theta') = (\bar{\nu}', X', \bar{\sigma}')\) for any \(\bar{\nu}'\) and for \(X' = W\).

By the operational rule \([\text{notify2}]\), the successor is \((\bar{T}_s, \bar{\mu})\), with thread composition as follows:

\[
\bar{T}_s = \ldots |(\theta, \epsilon, R, \bar{\sigma})|((\theta', \bar{\nu}', (N, d, n)), \bar{\sigma}')
\]

The successor of \((T, \mu)\) is \((T_s, \mu)\), given by the SyncTask’s rule \([\text{nf2}]\) as follows:

\[
T_s = \ldots |(\theta, \epsilon, R)|((\theta', S(\bar{\nu}'), (N, d, n))
\]

We have that \(\bar{T}_s(\theta) = (\bar{\nu}_s, X_s, \bar{\sigma})\) and \(T_s(\theta) = (S(\bar{\nu}_s), X_s)\) with \(S(\bar{\nu}_s) = \bar{\nu}_s = \epsilon\) and \(X_s = R\); also \(\bar{T}_s(\theta') = (\bar{\nu}', X', \bar{\sigma}')\) and \(T_s(\theta') = (S(\bar{\nu}'), X')\) for any \(\bar{\nu}'\), and for \(X' = N\). Thus, we have that \((\bar{T}_s, \bar{\mu}), (T_s, \mu) \in \mathcal{R}\).

Case 2. \texttt{wait}.

By \([5] \S 3.10\), there is our operational rule for \texttt{wait}: \([\text{wait2}]\), where the thread puts itself on the wait set of \(d\), and releases the lock associated to \(d\). A \texttt{wait} instruction for Java (and also its SyncTask equivalent) alters the memory. Thus, in contrast to the simplification used for \texttt{notify}, we present this case using the complete definition of configuration.

Let again \((\bar{T}, \bar{\mu})\) be a configuration of an annotated Java program, and \((T, \mu)\) be a configuration of the corresponding SyncTask s.t. \(\alpha(\bar{\mu}) = \mu\). Let there be a thread \(\theta\) s.t. the configuration is as follows:

\[
\bar{T} = \ldots |(\theta, d.\text{wait}(d), R, \bar{\sigma})\quad \bar{\mu} = \bar{\mu}'[\text{lock}(d) \mapsto (\theta, n)] \quad \text{where } n > 0
\]

Then, since \(S(d.\text{wait}(d)) = \text{wait}(d)\) there’s also a thread composition for the SyncTask program as follows:

\[
T = \ldots |(\theta, \text{wait}(d), R)\quad \mu = \mu'[\text{lock}(d) \mapsto (\theta, n)]
\]

We have that \((\bar{T}, \bar{\mu}), (T, \mu) \in \mathcal{R}\) since \(\bar{T}(\theta) = (\bar{\nu}, X, \bar{\sigma})\) and \(T(\theta) = (\bar{\nu}, X, \bar{\sigma})\) for \(\bar{\nu} = d.\text{wait}(d)\) and \(X = R\).

\[^4\] \([\text{wait1}]\) applies to exceptional flow, and is thus ignored.
By the operational rule [wait2], the successor is \((T_s, \mu_s)\) as follows:

\[
T_s = \ldots | (\theta, \epsilon, W, d, n) \quad \mu_s = \mu'[\text{lock}(d) \mapsto (\bot, 0)]
\]

The successor of \((T, \mu)\) is \((T_s, \mu_s)\), defined by the operational rule [wt] as follows:

\[
T_s = \ldots | (\theta, \epsilon, (W, d, n)) \quad \mu_s = \mu'[\text{lock}(d) \mapsto (\bot, 0)]
\]

We have that \(\overline{T_s}(\theta) = (\overline{t_s}, X_s, \overline{\sigma})\) and \(T_s(\theta) = (S(\overline{t_s}), X_s)\) with \(S(\overline{t_s}) = \overline{t_s} = \epsilon\) and \(X_s = (W, d, n)\); we also have that \((\alpha \circ \mu_s)(d) = \mu_s(d)\). Thus, again we have \((\overline{T_s}, \overline{\mu_s}), (T_s, \mu_s) \in R\). □

We now enunciate the result that reduces termination of a SyncTask program to a reachability problem on its corresponding CPN.

**Theorem 2 (SyncTask Termination).** A SyncTask program terminates iff its corresponding CPN unavoidably reaches a dead configuration in which the \textit{End} place has the same marking as the \textit{Start} place in the initial configuration.

**Proof (Sketch).** A CPN declares a place for each SyncTask variable. Moreover, there is a clear correspondence between the operational semantics of a SyncTask construct and its corresponding CPN component. It can be shown by means of weak bisimulation that every configuration of a SyncTask program is matched by a unique sequence of consecutive CPN configurations. Therefore, if the \textit{End} place in a dead configuration has the same marking as the \textit{Start} place in the initial configuration, then every thread in the SyncTask program terminates its execution, for every possible scheduling (note that the non-deterministic thread scheduler is simulated by the non-deterministic firing of transitions). □

CPN termination itself can be verified algorithmically by computing the reachability graph of the generated CPN and checking that: (i) the graph has no cycles, and (ii) the only reachable dead configurations are the ones where the marking in the \textit{End} place is the same as the marking in the \textit{Start} place in the initial configuration.

**8 The STaVe tool**

STaVe implements the parsing of annotated Java (and also SyncTask) source programs for generating ASTs of SyncTask programs, and the extraction of hierarchical CPNs from the ASTs. The tool has been written in Java, and is available at [7].

STaVe processes the annotations in an intricate scheme. It takes the annotated Java program as input, and uses the JavaParser library to generate the AST. Then it converts the JavaParser’s AST into the one of the OpenJDK compiler, to take advantage of its symbol table querying, type checking and code optimization. We have adopted JavaParser for the parsing because it associates the comments per-AST node, while OpenJDK’s parser discards annotations of
a finer granularity than methods. This allows the annotation of synchronized blocks. Next, STaVe traverses the Java AST three times, processing resource, initialization, and synchronization, in this order, to finally output the SyncTask AST. Then, it extracts the CPN from the AST following the mapping between SyncTask syntax and CPN components described in Section 6.

Two parts of STaVe turned out to be useful for others projects, and became spin-offs. The first is JavaParser2JCTree [3], a library that translates JavaParser ASTs to OpenJDK ASTs. The second is libcpntools [4], a library that generates hierarchical CPNs in the CPN Tools’s XML-based file format.

We now describe the experimental evaluation of our framework. This includes the process of annotating Java programs, extraction of the corresponding CPNs, and the analysis of the nets using CPN Tools. To evaluate the scalability of STaVe w.r.t. the size of the part of program that does not affect the synchronization, we take a test case that is a rather large program, but which has simple synchronization behaviour. Next, to evaluate the scalability of our tool w.r.t. the number of synchronizing threads, we take the example program from Section 3 and instantiate it with varying parameters.

In the first test case we annotate PIPE [6] (version 4.3.2), a CPN analysis tool written in Java. It contains a single synchronization scheme using CVs, where a thread that sends logs to a client via a socket waits for a server thread to establish the connection, and then notifies. This test case illustrates that synchronization involving CVs is typically simple and bounded. Manually annotating the program took just a few minutes, once the synchronization scheme was understood. The CPN extraction time was also negligible, and the verification process took just a few milliseconds to establish the correctness.

In the second test case, we performed experiments on the example presented in Section 3 with a varying number of threads, buffer capacity, and initial value. The experiments were performed in a Windows 7 virtual machine with 8GB of RAM and 2 processors, running on top a Linux machine with 16GB of RAM and a quad-core Intel i5 CPU of 1.30GHz.

1: CalculateOccGraph ();
2: CalculateSccGraph ();
3: NoOfNodes ();
4: fun terminal node =
   Mark.SyncTask_0 'End 1 node =
   Mark.SyncTask_0 'Start 1 InitNode;
5: not (List.null(ListDeadMarkings ()))
   andalso List.all (terminal) (ListDeadMarkings ())
   andalso (NoOfNodes () = SccNoOfNodes ());

Fig. 13: ML code for generating and querying a CPN state space
CPN Tools provides an ML API\cite{10} for generating and querying the state space. We collect our statistics by executing the code in Figure\cite{13}. Line 1 generates the state graph, from which the command on Line 2 computes the strongly connected components. Line 3 queries the reachable configurations. Line 4 defines a function which receives a configuration as parameter, and checks if the End place (in the SyncTask.0 subpage) has the same marking as the Start place in the initial configuration (InitNode). Line 5 checks the three termination conditions, namely: whether there is at least one dead configuration; whether all dead configurations respect the condition defined on Line 4, and whether the number of strongly connected components is equal to the size of the state graph, implying the absence of cycles.

Table\cite{1} presents the practical evaluation for a number of initial configurations. We observe an expected correlation between the number of tokens representing threads, the size of the state space, and the verification time. However, we have observed an unexpected influence of varying buffer capacities and initial states. We conjecture that the initial configurations that model high contention, i.e., many threads waiting on CVs, induce a larger state space. The experiments also show how the termination depends on the thread composition and initial state. That is, a single change in any parameter may affect the verification result.

9 Related Work

Leino et al.\cite{18} propose a compositional technique to verify the absence of deadlocks in concurrent systems with both locks and channels. They use deductive reasoning to define which locks a thread may acquire, or to impose an obligation for a thread to send a message. The authors acknowledge that their quantitative approach to channels does not apply to CVs, as messages passed through a channel are received synchronously, while a notification on a condition variable is either received, or else is lost.

Popeea and Rybalchenko\cite{19} present a compositional technique to prove termination of multi-threaded programs, which combines predicate abstraction and refinement with rely-guarantee reasoning. The technique is only defined for programs that synchronize with locks, and it cannot be easily generalized to support CVs. The reason for this is that the thread termination criterion is the absence of infinite computations; however, a finite computation where a waiting thread is never notified is incorrectly characterized as terminating.

Wang and Hoang\cite{20} propose a technique that permutes actions of execution traces to verify the absence of synchronization bugs. Their program model considers locks and condition variables. However, they cannot verify the property considered here, since their method does not permute matching pairs of wait-notify. For instance, it will not reorder a trace where, first, a thread waits, and then, another thread notifies. Thus, their method cannot detect the case where the notifying thread is scheduled first, and the waiting thread suspends the execution indefinitely.
Table 1: Statistics for Producer/Consumer

<table>
<thead>
<tr>
<th>Threads</th>
<th>Buffer capacity</th>
<th>Buffer elements</th>
<th>Terminates</th>
<th>Reachable Configurations</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer</td>
<td>Consumer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>42</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>no</td>
<td>43</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>91</td>
<td>32</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>0</td>
<td>no</td>
<td>157</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>283</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td>440</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>448</td>
<td>37</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1</td>
<td>yes</td>
<td>950</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>yes</td>
<td>968</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>1</td>
<td>yes</td>
<td>1395</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>2152</td>
<td>69</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td>2131</td>
<td>71</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td>3938</td>
<td>112</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>6</td>
<td>no</td>
<td>6573</td>
<td>183</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>16</td>
<td>no</td>
<td>24883</td>
<td>1097</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>29143</td>
<td>1308</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>1</td>
<td>yes</td>
<td>29573</td>
<td>1331</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td>64075</td>
<td>2867</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>24</td>
<td>no</td>
<td>78191</td>
<td>4524</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>16</td>
<td>no</td>
<td>84630</td>
<td>4699</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>1</td>
<td>yes</td>
<td>133824</td>
<td>7917</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>5</td>
<td>yes</td>
<td>164921</td>
<td>9952</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>197563</td>
<td>70614</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>1</td>
<td>no</td>
<td>211702</td>
<td>131226</td>
</tr>
</tbody>
</table>

Kaiser and Pradat-Peyre \[13\] propose the modelling of Java monitors in Ada, and the extraction of CPNs from Ada programs. However, they do not precisely describe how the CPNs are verified, nor provide a correctness argument about their technique. Also, they only validate their tool on toy examples with few threads. Our tool is validated on larger test cases, and on a real program.

Kavi \textit{et al.} \[14\] present PN components for the synchronization primitives in the Pthread library for C/C++, including condition variables. However, their modelling of CVs just allows the synchronization between two threads, and no argument is presented on how to use it with more threads.

Westergaard \[21\] presents a technique to extract CPNs for programs in a toy concurrent language, with locks as the only synchronization primitive. Our work borrows much from this work \textit{w.r.t.} the CPN modelling and analysis. However, we analyze full-fledged programming languages, and address the complications of analyzing programs with condition variables.
Finally, Van der Aalst et al. [1] present strategies for modelling complex parallel applications as CPNs. We borrow many ideas from this work, especially the modelling of hierarchical CPNs. However, their formalism is over-complicated for our needs, and we therefore simplify it to produce more manageable CPNs.

10 Conclusion

In this work we introduce a technique to prove the correct synchronization of parallel Java programs using condition variables. Correctness here means that if all threads reach their synchronization blocks, then all will eventually terminate the synchronization (and proceed with the execution). To the best of our knowledge, it is the first work to address the formal verification of programs using condition variables.

The technique models the synchronization using a simple language named SyncTask, which captures the relevant aspects of using condition variables. Then, we model the language features as Petri Net components, and propose a strategy to extract Coloured Petri Nets from SyncTask programs. We defined both the language and CPN generation such that we transform the termination of SyncTask programs into the problem of state-space exploration of CPNs.

The STaVe implements the extraction of CPNs via SyncTask programs, from annotated Java programs. We validate our technique on a number of test-cases, using CPN Tools as back-end.

As future work, we plan to extend the annotation language to extract SyncTask program from C and C++ programs synchronizing via POSIX condition variables. Moreover, we would like to support other CPN analysis tools, such as PIPE. It is also in our plans the output for model checkers, such as Spin (Promela) or NuSMV, and compare the performance of the different back-ends.

References