Specification and Verification of Synchronization with Condition Variables

Pedro de Carvalho Gomes[1], Dilian Gurov[1], and Marieke Huisman*[2]

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

In this paper we propose a technique to specify and verify the *correct synchronization* of concurrent programs with condition variables. We define correctness as the liveness property: "every thread synchronizing under a set of condition variables eventually exits the synchronization", under the assumption that every such thread eventually reaches its synchronization block. Our technique does not avoid the combinatorial explosion of interleavings of thread behaviors. Instead, we alleviate it by abstracting away all details that are irrelevant to the *synchronization behavior* of the program, which is typically significantly smaller than its overall behavior. First, we introduce SyncTask, a simple imperative language to specify parallel computations that synchronize via condition variables. We consider a SyncTask program to have a correct synchronization iff it terminates. Further, to relieve the programmer from the burden of providing specifications in SyncTask, we introduce an economic annotation scheme for Java programs to assist the *automated extraction* of SyncTask programs capturing the synchronization behavior of the underlying program. We prove that every Java program annotated according to the scheme (and satisfying the assumption) has a correct synchronization iff its corresponding SyncTask program terminates. We show how to transform the verification of termination into a standard reachability problem over Colored Petri Nets that is efficiently solvable by existing Petri Net analysis tools. Both the SyncTask program extraction and the generation of Petri Nets are implemented in our STAVE tool. We evaluate the proposed framework on a number of test cases as a proof-of-concept.

1 Introduction

Condition variables (CV) are a commonly used synchronization mechanism to coordinate multithreaded programs. Threads *wait* on a CV, 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 is 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, they are provided both natively as an object's *monitor* [6], i.e., a pair of a lock and a CV, and in the concurrent API, as one-to-many

* Supported by ERC grant 258405 for the VerCors project.

`Condition` objects associated to a `Lock` object. The mechanism is typically employed when the progress of threads depends on the state of a shared variable, to avoid busy-wait loops that poll the state of this shared variable. 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.* [14] acknowledge that verifying the absence of deadlocks when using CVs is hard because a notification is "lost" if no thread is waiting on it. Thus, one cannot verify locally whether a waiting thread will eventually be notified. Furthermore, the synchronisation conditions can be quite complex, involving both control-flow and data-flow aspects as arising from method calls; their correctness thus depends on the *global thread composition*, i.e., the type and number of parallel threads. All these complexities suggest the need for *programmer-provided annotations* to assist the automated analysis, which is the approach we are following here.

In this work, we present a formal technique for specifying and verifying that "every thread synchronizing under a set of condition variables eventually exits the synchronization", under the assumption that every such thread eventually reaches its synchronization block. The assumption itself is not addressed here, as it does not pertain to correctness of the synchronization, and there already exist techniques for dealing with such properties (see e.g. [16]). Note that the above correctness notion applies to a *one-time synchronization* on a condition variable only; generalizing the notion to repeated synchronizations is left for future work. 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, 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 problem by *delimiting the relevant aspects of the synchronization.*

First, we consider correctness of synchronization in the context of a *synchronization specification language.* As we target arbitrary programming languages that feature locks and condition variables, we do not base our approach on a subset of an existing language, but instead introduce *SyncTask*, a simple concurrent programming language where all computations occur inside synchronized code blocks. We define a SyncTask program to have a correct synchronization iff it terminates. The SyncTask language has been designed to capture common patterns of CV usage, while abstracting away from 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 address the problem of verifying the correct usage of CVs in real concurrent programming languages by showing how SyncTask can be used to capture the synchronization of a Java program, provided it is bounded. There is a consensus in Software Engineering that synchronization in a concurrent program

must be kept to a minimum, both in the number and complexity of the synchronization actions, and in the number of places where it occurs. This avoids the latency of blocking threads, and minimizes the risk of errors, such as dead- and livelocks. As a consequence, many programs present a finite (though arbitrarily large) synchronization behavior. To assist the automated extraction of finite synchronization behavior from Java programs as SyncTask programs, we introduce an *annotation scheme*, which requires the user to (correctly) annotate, among others, the initialization of new threads (i.e., creation of `Thread` objects), and provide the initial state of the variables accessed inside the synchronized blocks. We establish that for correctly annotated, bounded Java programs, correctness of synchronization is equivalent to termination of the extracted SyncTask program.

As a proof-of-concept of the algorithmic solvability of the termination problem for SyncTask programs, we show how to transform it into a reachability problem on hierarchical Colored Petri Nets[3] (CPNs) [7]. We define how to extract CPNs automatically from SyncTask programs, following a previous technique from Westergaard [18]. 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*. Standard CPN analysis tools can efficiently compute the reachability graphs, and check whether the termination condition holds. Also, in case that the condition does not hold, an inspection of the reachability graph easily provides the cause of non-termination.

We implement the extraction of SyncTask programs from annotated Java and the translation of SyncTasks to CPNs as the STaVe tool. We evaluate the tool on two test-cases, by generating CPNs from annotated Java programs and analyzing these with CPN Tools [8]. The first test-case evaluates the scalability of the tool w.r.t. the size of program code that does not affect the synchronization behavior of the program. The second test-case evaluates the scalability of the tool w.r.t. the number of synchronizing threads. 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.

In summary, this work makes the following contributions: (i) the SyncTask language to model the synchronization behavior of programs with CVs, (ii) an annotation scheme to aid the extraction of the synchronization behavior of Java programs, (iii) an extraction scheme of SyncTask models from annotated Java programs, (iv) a reduction of the termination problem for SyncTask programs to a reachability problem on CPNs, (v) an implementation of the framework by means of STaVe, and (vi) its experimental evaluation.

The remainder of the paper is organized as follows. Section 2 introduces SyncTask. Section 3 describes the mapping from annotated Java to SyncTask,

[3] The choice of formalism has been mainly based on the *simplicity* of CPNs as a general model of concurrency, rather than on the existing support for efficient model checking. For the latter, model checking tools exploiting parametricity or symmetries in the models may prove more efficient in practice.

SyncTask ::= *ThreadType* Main*
ThreadType ::= `Thread` *ThreadName* `{` *SyncBlock** `}`
Main ::= `main {` *VarDecl* StartThread** `}`
StartThread ::= `start(`*Const*`,` *ThreadName*`);`
Expr ::= *Const* | *VarName* | *Expr* ⊕ *Expr*
| `min(`*VarName*`)` | `max(`*VarName*`)`
VarDecl ::= *VarType VarName*`(`*Expr**`);`
VarType ::= `Bool` | `Int` | `Lock` | `Cond`
SyncBlock ::= `synchronized (`*VarName*`)` *Block*
Block ::= `{` *Stmt** `}`
Assign ::= *VarName* `=` *Expr* `;`
Stmt ::= *SyncBlock* | *Block*
| *Assign* | `skip;`
| `while` *Expr Stmt*
| `if` *Expr Stmt* `else` *Stmt*
| `notify(`*VarName*`);`
| `notifyAll(`*VarName*`);`
| `wait(`*VarName*`);`

Fig. 1: SyncTask Syntax

while Section 4 presents the translation into CPNs, and presents test-cases. We discuss related work in Section 5. Section 6 concludes and suggests future work.

2 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.

2.1 Syntax and Types

The SyncTask syntax is presented in Figure 1. A program has two main parts: *ThreadType**, which declares the different types of parallel execution flows, and *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 *ThreadType* consists of adjacent *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 *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.

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

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

```
21 main {
    Lock m_lock();
23  Cond m_cond(m_lock);
    Int b_els(0,1,1);
25  start(1,Producer);
    start(2,Consumer);
27 }
```

Fig. 2: Modelling of synchronization via a shared buffer in SyncTask

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.

Example 1 (SyncTask program). The program in Figure 2 models synchronization via a shared buffer. `Producer` and `Consumer` represent the synchronization behavior: threads synchronize via the CV `m_cond` 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 `main` block contains variable declarations and initialization. The lock `m_lock` is associated to `m_cond`. `b_els` is an integer in the interval [0,1] (initially set to `1`), and represents the number of elements in the buffer. One `Producer` and two `Consumer` threads are spawned with `start`.

2.2 Structural Operational Semantics

We now define the semantics of SyncTask, to provide the means for establishing formal correctness results.

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 $(Thread\_id \cup \{\bot\}) \times \mathbb{N}$ 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 [11]. Let μ 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 μ 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 d

$$[\text{s1}]^{a} \quad T|(\theta, \texttt{synchronized(}o\texttt{)}\ b, R), \mu \longrightarrow T|(\theta, \texttt{synchronized'(}o\texttt{)}\ b, R), \mu[o \mapsto (\theta, 1)]$$

$$[\text{s2}]^{b} \quad T|(\theta, \texttt{synchronized(}o\texttt{)}\ b, R), \mu \longrightarrow T|(\theta, \texttt{synchronized'(}o\texttt{)}\ b, R), \mu[o \mapsto (\theta, n+1)]$$

$$[\text{s3}]^{b} \quad \frac{T|(\theta, b_1, R), \mu \longrightarrow T|(\theta, b_2, X), \mu'}{T|(\theta, \texttt{synchronized'(}o\texttt{)}\ b_1, R)), \mu \longrightarrow T|(\theta, \texttt{synchronized'(}o\texttt{)}\ b_2, X), \mu'}$$

$$[\text{s4}]^{c} \quad \frac{T|(\theta, b, R), \mu \longrightarrow T|(\theta, \epsilon, R), \mu'}{T|(\theta, \texttt{synchronized'(}o\texttt{)}\ b, R)), \mu \longrightarrow T|(\theta, \epsilon, R), \mu'[o \mapsto (\theta, n-1)]}$$

$$[\text{s5}]^{d} \quad \frac{T|(\theta, b, R), \mu \longrightarrow T|(\theta, \epsilon, R), \mu'}{T|(\theta, \texttt{synchronized'(}o\texttt{)}\ b, R), \mu \longrightarrow T|(\theta, \epsilon, R), \mu'[o \mapsto (\bot, 0)]}$$

$$[\text{wt}]^{e} \quad T|(\theta, \texttt{wait(}d\texttt{)}, R), \mu \rightarrow T|(\theta, \epsilon, (W, d, n)), \mu[\mathit{lock}(d) \mapsto (\bot, 0)]$$

$$[\text{nf1}]^{ef} \quad T|(\theta, \texttt{notify(}d\texttt{)}, R), \mu \rightarrow T|(\theta, \epsilon, R), \mu$$

$$[\text{nf2}]^{eg} \quad T|(\theta, \texttt{notify(}d\texttt{)}, R)|(\theta', t', (W, d, n)), \mu \rightarrow T|(\theta, \epsilon, R)|(\theta', t', (N, d, n)), \mu$$

$$[\text{na1}]^{ef} \quad T|(\theta, \texttt{notifyAll(}d\texttt{)}, R), \mu \rightarrow T|(\theta, \epsilon, R), \mu$$

$$[\text{na2}]^{eg} \quad T|(\theta, \texttt{notifyAll(}d\texttt{)}, R)|T^{d}_{W}, \mu \rightarrow T|(\theta, \epsilon, R)|\{(\theta', t', (N, d, n))|(\theta', t', (W, d, n)) \in T^{d}_{W}\}, \mu$$

$$[\text{rd}]^{h} \quad T|(\theta, t, (N, d, n)), \mu \rightarrow T|(\theta, t, R), \mu[\mathit{lock}(d) \mapsto (\theta, n)]$$

$^{a}\mu(o) = (\bot, 0)$ $^{b}\mu(o) = (\theta, n) \wedge n > 0$ $^{c}\mu(o) = (\theta, n) \wedge n > 1$ $^{d}\mu(o) = (\theta, 1)$

$^{e}\mu(\mathit{lock}(d)) = (\theta, n) \wedge n > 0$ $^{f}\mathit{waitset}(d) = \emptyset$ $^{g}\mathit{waitset}(d) \neq \emptyset$ $^{h}\mu(\mathit{lock}(d)) = (\bot, 0)$

Fig. 3: Operational rules for synchronization

that a thread is/was waiting on, and the number n of times it must reacquire the lock to proceed with the execution. The auxiliary function $\mathit{waitset}(d)$ returns the id's of all threads waiting on a CV d.

We represent a thread as (θ, t, X), where θ 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 θ is not defined in T. For convenience, we abuse set notation to denote the composition of threads in the set; e.g., $T^{d}_{W} = \{(\theta, t, (W, d, n))\}$ represents the composition of all threads in the wait set of d. A *program configuration* is a pair (T, μ) of the threads' composition and its memory. A thread terminates if the program reaches a configuration where its code t is empty (ϵ); 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 μ. 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()` initializes 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: $(\theta, t, R)|(\theta', t, R)$.

Figure 3 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 [2, § 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 `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 CV's lock, and releases it. The rules [nf1] and [na1] apply when a thread notifies a CV with an empty wait set; the behavior is the same as for the `skip` statement. By rule [nf2], a thread notifies a 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 awoken. By the rule [rd], a thread reacquires all the locks it had relinquished, changes the state to R, and resumes the execution after the control point where it invoked `wait`.

Finally, we define a SyncTask program to *have a correct synchronization* iff it terminates.

3 From Annotated Java To SyncTask

The annotation process supported by STAVE relies on the programmer's knowledge about the intended synchronization, and consists of providing hints to the tool to automatically map the synchronization to a SyncTask program. In this section we present an *annotation scheme* for writing such hints, and sketch a correctness argument for the extraction.

3.1 An Annotation Language for Java

An annotation in STAVE binds to a specific type of Java declaration (e.g., classes or methods). The annotation starts in a comment block immediately above a declaration, with additional annotations inside the declaration's body. Annotations share common keywords (though with a different semantics), and overlap in the declaration 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 4 presents the annotation language. Arguments given within square brackets are optional, while text within parentheses tells which declaration types the annotation binds to. The programmer has to (correctly) provide, by means of annotations, the following *three types of information*: resources, synchronization and initialization.

Resource annotation:

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

Synchronization annotation:

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

Initialization annotation:

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

Fig. 4: Annotation language for Java programs

A *resource* is a data type that is manipulated by the synchronization. It abstracts the state of a data structure to a bounded integer, which is potentially a *ghost variable* (as in [12]), and defines how the methods operate on it. For example, the annotation abstracts a linked list or a buffer to its size. In case a resource is mapped to a ghost variable, we say that the variable *extends* the program memory. Resources bind to classes only, and the 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 specifies that the method potentially alters the resource state. Similarly, `@predicate` binds to methods and specifies that the method returns a predicate about the state.

There are two ways to extract an annotated method's behavior. `@code` tells StaVe not to process the method, but instead to associate it to the code enclosed between `@{` and `}@`, while `@inline` tells StaVe to try to infer the method declaration with the potential aid of `@maps`, which syntactically replaces a Java command (e.g., a method invocation) with a SyncTask code snippet.

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` annotates that a local variable is a reference to a global object in the heap, which is observed and is represented by an alias.

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. They bind to methods, and the switch `@synctask` starts the declaration. Here, `@resource`, `@lock`, `@condvar` and `@monitor` define the objects being observed,

```
01 class Producer extends Thread {
    Buffer buffer;
03  Producer(Buffer b){buffer=b;}
    public void run() {
05  /*@syncblock
       @monitor buffer -> m
07     @resource buffer:Buffer */
    synchronized(buffer) {
09   while (buffer.full())
       buffer.wait();
11   buffer.add();
     buffer.notifyAll();
13 } } }

15 class Consumer extends Thread {
    Buffer buffer;
17  Consumer(Buffer b){buffer=b;}
    public void run() {
19   /*@syncblock
        @monitor buffer -> m
21      @resource buffer:Buffer */
     synchronized(buffer) {
23    while (buffer.empty())
       buffer.wait();
25    buffer.remove();
      buffer.notifyAll();
27 } } }
```

```
29 /*@resource @capacity cap
      @object els->b_els
31    @value els->b_els */
   class Buffer {
33  int els; final int cap;
    /* @operation @inline */
    void remove(){if (els>0)els--;}
    /* @operation @inline */
37  void add(){if (els<cap)els++;}
    /* @predicate @inline */
39  boolean full(){return els==cap;}
    /* @predicate @inline */
41  boolean empty(){return els==0;}
    /*@synctask Buffer
       @monitor b -> m
43     @resource b->b_els */
    static void main(String[] s) {
45   Buffer b = new Buffer();
     b.els = 1; b.cap = 1;
47   /* @thread */
     Consumer c1 = new Consumer(b);
     /* @thread */
49   Consumer c2 = new Consumer(b);
     /* @thread */
51   Producer p = new Producer(b);
     c1.start();p.start();c2.start();
53 } }
```

Fig. 5: Annotated Java program synchronizing via shared buffer

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. The object's type must have been annotated with a synchronization annotation.

Example 2 (Annotated Java). The SyncTask program in Figure 2 was generated from the Java program in Figure 5. We now discuss how the annotations delimit the expected synchronization. The example also illustrates the extraction.

The `@syncblock` annotations (lines 5/19) add the following `synchronized` blocks to the observed synchronization behavior, and its arguments `@monitor` and `@resource` (lines 6/20 and 7/21, respectively) map local references to global aliases. The `@resource` annotation (line 29) starts the definition of a resource type. `@value`, `@object`, `@capacity` (lines 29/30/31) define how the abstract state is represented by a bounded integer; in this example, the state is equivalent to `els`, which is an abstraction of the number of elements in a buffer. The `@operation` (lines 34/36) and `@predicate` (lines 38/40) annotations define how the methods operate on the state. Notice that the annotated methods have been inlined in Figure 2, i.e., `add` is inlined in lines 5 and 6. The `@synctask` annotation

above `main` starts the declaration of locks, CVs and resources, and `@thread` annotations add the underneath objects to the global thread composition.

3.2 Synchronization Correctness

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 [16]) 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 if 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 $\mathcal{P}$ 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 $\mathcal{P}$* has a correct synchronization *iff 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. Although out of the scope of the present work, the annotations can potentially be checked, or partially generated, with existing static analysis techniques. Further, we assume the memory model of synchronized actions in a Java program to be sequentially consistent.

We now connect synchronization schemes of annotated Java programs with SyncTask programs. We shall assume that the programmer has correctly annotated the program, as described in Section 3.1.

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

Proof (Sketch). To prove the result, we define a binary relation R between the configurations of the Java program and its SyncTask, and show it to be a *weak bisimulation* (see [15]), implying that the SyncTask program eventually reaches a terminal configuration (i.e., all threads terminate) *if and only if* the original Java program has a correct synchronization. We refer to the accompanying technical report [5] for the full formalization, and for the most interesting cases, namely the `notify` and `wait` instructions.

The Java annotations define a bidirectional mapping between (some of) the Java program variables and ghost variables and the corresponding bounded variables in SyncTask. Thus, we define R to relate configurations that agree on *common* variables. Similarly, we define the set of *visible transitions* as the ones that

update common variables, and treat all other transitions as *silent*. We argue that R is a weak bisimulation in the standard fashion: We establish that (i) the initial values of the common variables are the same for both programs, and (ii) assuming that observed variables in a Java program are only updated inside annotated synchronized blocks, we establish that any operation that updates a common variable has the same effect on it in both programs.

To prove (i) it suffices to show that the initial values in the Java program are the same as the ones provided in the initialization annotation, as described in Section 3. (Here we rely on the correctness of the annotations; however, existing techniques such as [13,14] can potentially be used for checking this.) The proof of (ii) requires to show that updates to a common variable yield the same result in both programs. It goes by case analysis on the Java instructions set. Each case shows that for any configuration pair of R, the operational rules for the given Java instruction and for the corresponding SyncTask instruction lead to a pair of configurations that again agree on the common variables. As the semantics of SyncTask presented in Section 2 has been designed to closely mimic the Java semantics defined in [2], the elaboration of this is straightforward. □

4 Verification of Synchronization Correctness

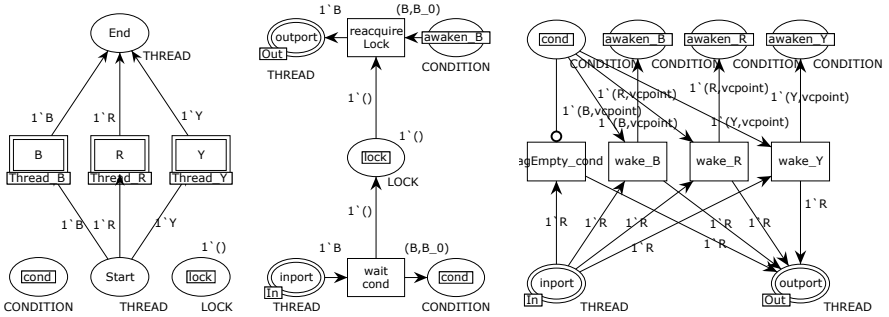
In this section we show how termination of SyncTask programs can be reduced to a reachability problem on Colored Petri Nets (CPN), and present an experimental evaluation of the verification with STaVe and CPN Tools.

4.1 SyncTask Programs as Colored Petri Nets

Various techniques exist to prove termination of concurrent systems. For SyncTask, it is essential that such a technique efficiently encodes the concurrent thread interleaving, the program's control flow, synchronization primitives, and basic data manipulation. Here, we have chosen to reduce the problem of termination of SyncTask programs to a reachability problem on hierarchical CPNs extracted from the program. CPNs allow a natural translation of common language constructs into CPN components (for this we re-use results from Westergaard [18]), and are supported by analysis tools such as CPN Tools. We assume some familiarity with CPNs, and refer the reader to [7] for a detailed exposition.

The color set THREAD associates a color to each `Thread` type declaration, and a thread is represented by a token with a color 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 1 are mapped into hierarchical CPN components, where *substitute transitions* (STs; depicted as doubly outlined rectangles) represent the non-terminals on the right-hand side. Figure 6a shows the component for the start symbol *SyncTask*. The Start place contains all thread tokens in



(a) *SyncTask* **(b)** `wait` **(c)** `notify`

Fig. 6: Top-level component and condition variables operations

the initial configuration, connected by arcs (one per color) to the STs denoting the thread types, and End, which collects the terminated thread tokens. It also contains the places that represent global variables.

Figure 6b shows the modelling of `wait`. The transition `wait cond` produces two tokens: one into the place modelling the CV, and one into the place modelling the lock, representing its release. The other transition models a notified thread reacquiring the lock, and resuming the execution. Figure 6c shows the modelling of `notify`. The `Empty_cond` transition is enabled if the CV is empty, and the other transitions, with one place per color, model the non-deterministic choice of which thread to notify. The component for `notifyAll` (not shown) is similar.

The initialization in *Main* declares the initial set of tokens for the places representing variables, and the number and colors of thread tokens. A `Lock` creates a place containing a single token; it being empty represents that some thread holds the lock. The color set CPOINT represents the control points of `wait` statements. A `Condition` variable gives rise to an empty place representing the waiting set, with color set CONDITION. Here, colors are pairs of THREAD and CPOINT. Both data are necessary to route correctly notified threads to the correct place where they resume execution.

4.2 SyncTask Termination as CPN Reachability

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* End *place has the same marking as the* 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 End place in a dead configuration has the same marking as the 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 End place is the same as the marking in the Start place in the initial configuration.

4.3 The STAVE Tool

We have implemented the parsing of annotated Java programs to generate SyncTask programs, and the extraction of hierarchical CPNs from SyncTask, as the STAVE [4] tool. 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.

Our first test case evaluates the scalability of STAVE w.r.t. the size of the part of program that does *not* affect the synchronization. For this, we annotated PIPE [3] (version 4.3.2), a rather large CPN analysis tool written in Java. It contains a single (and simple) synchronization scheme using CVs: a thread that sends logs to a client via a socket waits for a server thread to establish the connection, and then to notify. 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 negligible, and the verification process took just a few milliseconds to establish the correctness.

Our second test case evaluates the scalability of STAVE w.r.t. the number of threads. We took the example program from Section 2, and instantiated it with a varying number of threads, buffer capacity, and initial value. Table 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. Less expected for us was the observed influence of the 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 termination depends on the thread composition and the initial state. Hence, a single change in any parameter may affect the verification result.

5 Related Work

Leino *et al.* [14] propose a compositional technique to verify the absence of deadlocks in concurrent systems with both locks and channels. They use deductive

Table 1: Statistics for Producer/Consumer

Initial Configuration				Analysis		
Threads		Buffer		SyncTask	Reachable CPN	Time (ms)
Producer	Consumer	capacity	elements	Terminates	Configurations	
1	2	1	1	yes	42	31
1	2	2	0	no	43	28
2	2	1	0	yes	91	32
7	1	5	0	no	157	33
3	3	1	0	yes	283	32
6	5	5	4	yes	968	40
7	6	7	1	yes	1395	54
6	5	1	1	no	2131	71
7	6	1	1	no	3938	112
11	9	7	6	no	6573	183
17	16	16	16	no	24883	1097
11	11	1	0	yes	29143	1308
14	13	7	1	yes	29573	1331
14	13	1	1	no	64075	2867
26	24	25	24	no	78191	4524
18	18	5	1	yes	133824	7917
16	21	5	5	yes	164921	9952
18	18	1	1	yes	197563	70614
20	18	2	1	no	211702	131226

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 [16] 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 [17] 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.

Kaiser and Pradat-Peyre [9] 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 *et al.* [10] 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 [18] 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 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.

6 Conclusion

We presented a technique to prove the correct synchronization of Java programs using condition variables. Correctness here means that if all threads reach their synchronization blocks, then all will eventually terminate the synchronization. Our technique does not avoid the exponential blow-up of the state space caused by the interleaving of threads; instead, it alleviates the problem by isolating the synchronization behavior.

We introduced SyncTask, a simple language to capture the relevant aspects of synchronization using condition variables. Also, we define an annotation scheme for programmers to map the expected synchronization in a Java program to a SyncTask program. We establish that the synchronization is correct w.r.t. the above-mentioned property *iff* the corresponding SyncTask terminates. As a proof-of-concept, to check termination we define a translation from SyncTask programs into Colored Petri Nets such that the program terminates *iff* the net invariably reaches a special configuration. The extraction of SyncTask from annotated Java programs, and the translation to CPNs, is implemented as the STaVe tool. We validate our technique on some test-cases using CPN Tools.

Our current results hold for a number of *restrictions* on the analysed programs. In future work we plan to address and relax these restrictions, integrate special-purpose static analysers for the separate types of required annotations, incorporate more sophisticated model checkers for checking termination of SyncTask programs, and perform a more diverse experimental evaluation and comparison with other verification techniques.

References

1. van der Aalst, W., Stahl, C., Westergaard, M.: Strategies for modeling complex processes using colored Petri nets. In: Transactions on Petri Nets and Other Models of Concurrency VII, LNCS, vol. 7480, pp. 6–55. Springer Berlin Heidelberg (2013)
2. Cenciarelli, P., Knapp, A., Reus, B., Wirsing, M.: An event-based structural operational semantics of multi-threaded Java. In: Formal Syntax and Semantics of Java, LNCS, vol. 1523, pp. 157–200. Springer Berlin Heidelberg (1999)
3. Dingle, N.J., Knottenbelt, W.J., Suto, T.: PIPE2: A tool for the performance evaluation of generalised stochastic Petri nets. SIGMETRICS 36(4), 34–39 (2009)
4. Gomes, P.: SyncTAsk VErifier. `http://www.csc.kth.se/~pedrodcg/stave` (2015)
5. Gomes, P.d.C., Gurov, D., Huisman, M.: Algorithmic verification of multithreaded programs with condition variables. Tech. rep., KTH Royal Institute of Technology (October 2015), `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-176006`
6. Hoare, C.A.R.: Monitors: An operating system structuring concept. Commun. ACM 17(10), 549–557 (Oct 1974)
7. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer Publishing Company, Incorporated, 1st edn. (2009)
8. Jensen, K., Kristensen, L., Wells, L.: Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer 9(3-4), 213–254 (2007)
9. Kaiser, C., Pradat-Peyre, J.F.: Weak fairness semantic drawbacks in Java multithreading. In: Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies. pp. 90–104. Springer-Verlag (2009)
10. Kavi, K., Moshtaghi, A., Chen, D.j.: Modeling multithreaded applications using Petri nets. International Journal of Parallel Programming 30(5), 353–371 (2002)
11. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. 28(9), 690–691 (Sep 1979)
12. Leavens, G., Baker, A., Ruby, C.: JML: A notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems, Eng. and Comp. Sci., vol. 523, pp. 175–188. Springer US (1999)
13. Leino, K.R., Müller, P.: A basis for verifying multi-threaded programs. In: Proceedings of the 18th European Symposium on Programming Languages and Systems. pp. 378–393. ESOP '09, Springer-Verlag, Berlin, Heidelberg (2009)
14. Leino, K.R.M., Müller, P., Smans, J.: Deadlock-free channels and locks. In: European Conference on Programming Languages and Systems. pp. 407–426. ESOP'10, Springer-Verlag (2010)
15. Milner, R.: Communicating and mobile systems: the π-calculus, chap. 6, pp. 52–53. Cambridge University Press, New York, NY, USA (1999)
16. Popeea, C., Rybalchenko, A.: Compositional termination proofs for multi-threaded programs. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 237–251. TACAS'12, Springer-Verlag (2012)
17. Wang, C., Hoang, K.: Precisely deciding control state reachability in concurrent traces with limited observability. In: Verification, Model Checking, and Abstract Interpretation, LNCS, vol. 8318, pp. 376–394. Springer Berlin Heidelberg (2014)
18. Westergaard, M.: Verifying parallel algorithms and programs using coloured Petri nets. In: Transactions on Petri Nets and Other Models of Concurrency VI, LNCS, vol. 7400, pp. 146–168. Springer Berlin Heidelberg (2012)