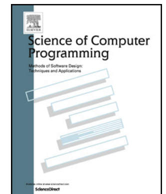
Science of Computer Programming 163 (2018) 174–189

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico



Specification and verification of synchronization with condition variables

Pedro de C. Gomes [a,*], Dilian Gurov [a], Marieke Huisman [b,1], Cyrille Artho [a]

[a] KTH Royal Institute of Technology, Stockholm, Sweden
[b] University of Twente, Enschede, the Netherlands

A R T I C L E   I N F O

Article history:
Received 17 May 2017
Received in revised form 2 May 2018
Accepted 2 May 2018
Available online 8 May 2018

Keywords:
Concurrency
Formal verification
Java
Condition variables

A B S T R A C T

This paper proposes a technique to specify and verify the *correct synchronization* of concurrent programs with condition variables. We define correctness of synchronization as the liveness property: "every thread synchronizing under a set of condition variables eventually exits the synchronization block", under the assumption that every such thread eventually reaches its synchronization block. Our technique does not avoid the combinatorial explosion of interleavings of thread behaviours. Instead, we alleviate it by abstracting away all details that are irrelevant to the *synchronization behaviour* of the program, which is typically significantly smaller than its overall behaviour. 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 behaviour of the underlying program. We show that every Java program annotated according to the scheme (and satisfying the assumption mentioned above) has a correct synchronization iff its corresponding SyncTask program terminates. We then show how to transform the verification of termination of the SyncTask program into a standard reachability problem over Coloured 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.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

Condition variables in concurrent programs. 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 signalling is asynchronous: the effect of the notification can be delayed. If no thread is waiting on the CV, then the notification has no effect. CVs are used in conjunction with

* Corresponding author.
E-mail addresses: pedrodcg@kth.se (P. de C. Gomes), dilian@kth.se (D. Gurov), m.huisman@utwente.nl (M. Huisman), artho@kth.se (C. Artho).

1 Supported by ERC grant 258405 for the VerCors project.

https://doi.org/10.1016/j.scico.2018.05.001
0167-6423/© 2018 Elsevier B.V. All rights reserved.

```
01 class Utilizer extends Thread {
    synchronized(lock) {
03   while (!resource_available) {
      lock.wait();
05   }
    }
07 }
```

```
   class Provider extends Thread {
09  synchronized(lock) {
     // prepare resource
11   resource_available = true;
     lock.notify();
13  }
   }
```

Fig. 1. A simple Java program using `wait`/`notify`.

locks; a thread must have acquired 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* [1], i.e., a pair of a lock and a CV, and in the `java.util.concurrent` library, as one-to-many `Condition` objects associated to a `Lock` object. C/C++ have similar mechanisms provided by the POSIX thread (Pthread) library, and C++ features CVs natively since 2011 [2] as the `std::condition_variable` class. 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.

Example 1 *(Condition variables in Java).* Fig. 1 shows a simple example with two threads: The first thread, *Utilizer*, wants to use a shared resource. The resource is guarded with a common lock (line 2) to ensure that only one thread, the lock holder, can change the state of the resource. Because no high-level constructs like `await(resource_available)` exist in Java, the *Utilizer* thread has to check if the condition holds by using a conditional statement (line 3). If the condition is false, the *Utilizer* suspends itself by calling `wait` in line 4. This call implicitly relinquishes the lock, to allow another thread to access it and modify the condition variable. At some point, another thread may make the resource available. That thread then has to signal the state change to the condition variable. In our example, thread *Provider* uses the same lock to access the shared variable, and calls `notify` to signal a change in line 12.

As a result of that signal, one of the waiting threads is woken up. It has first to re-check the condition, since it might have been re-invalidated by another thread in the meantime. To do this, the lock is (implicitly) re-acquired. In case another thread has already consumed the resource, and `resource_available` is again *false*, the `while` loop in line 3 is re-entered. Otherwise, the waiting thread may proceed under the assumption that `resource_available` is *true*. This assumption holds if all accesses to the shared condition variable are protected by a common lock, i.e., if the whole program is data race free.

The `notify` method wakes up any one thread that is waiting at the time the notification is sent; there is no mechanism to ensure that a particular thread gets woken up. If multiple waiting threads may check or use shared conditions in different ways (for example, by using a function over multiple shared variables), the notifying thread should call `notifyAll`, to ensure each waiting thread gets woken up once and can re-check the condition variable to see if the "right" condition is true.

Waiting threads may get interrupted in real Java programs, so they have to guard any call to `wait` with a `try`/`catch` block, to catch an `InterruptedException`. Furthermore, the Java Specification [3, § 17.2] permits (but discourages) JVM implementations to perform spurious wake-ups, and reinforces the coding practice of invoking `wait` inside loops guarded by a logical condition necessary for thread progress. We elide these functionalities in our paper.

Writing correct programs using condition variables is challenging, mainly because of the complexity of reasoning about asynchronous signalling. Nevertheless, condition variables have not been addressed sufficiently with formal techniques, to no small part due to this complexity. For instance, Leino et al. [4] 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 synchronization 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., [5]). 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 a bounded number 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*.

SyncTask. 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. It has 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.

Verification of concurrent programs. Next, we address the problem of verifying the correct usage of CVs in real concurrent programming languages. We show how SyncTask can be used to capture the synchronization of a Java program, provided it is bounded. Object-oriented languages similar to Java, such as C++ and C#, can be analyzed likewise. 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 [6,7]. This avoids the latency of blocking threads, and minimizes the risk of errors, such as dead- and live-locks. As a consequence, many programs present a finite (though arbitrarily large) synchronization behaviour. That is, the number of variables involved in the synchronization, and their data domains are bounded.

Implementation. To assist the automated extraction of finite synchronization behaviour 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 to provide the initial state of the variables accessed inside the synchronized blocks. We establish that for correctly annotated Java programs with bounded synchronization behaviour, 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 Coloured Petri Nets[2] (CPNs) [8]. We define how to extract CPNs automatically from SyncTask programs, following a previous technique from Westergaard [9]. 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.

Evaluation. 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 [10]. The first 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. 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 behaviour of programs with CVs, (ii) an annotation scheme to aid the extraction of the synchronization behaviour 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.

Outline. The remainder of the paper is organized as follows. Section 2 introduces SyncTask. Section 3 describes the mapping from annotated Java to SyncTask, while Section 4 presents the translation into CPNs. Section 5 presents STaVe and its experimental evaluation. We discuss related work in Section 6. Section 7 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 Fig. 2. 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.

[2] 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. 2. SyncTask syntax.

```
01 Thread Producer {
    synchronized(m_lock){
03   while(b_els==max(b_els)){
      wait(m_cond);
05   }
     if (b_els<max(b_els)) {
07    b_els=(b_els+1);
     } else {
09    skip;
     }
11   notifyAll(m_cond);
    }
13 }
```

```
   Thread Consumer {
15  synchronized(m_lock){
     while((b_els==0)){
17    wait(m_cond);
     }
19   if((b_els>0)) {
      b_els=(b_els-1);
21   } else {
      skip;
23   }
     notifyAll(m_cond);
25  }
   }
```

```
27 main {
    Lock m_lock();
29  Cond m_cond(m_lock);
    Int b_els(0,7,1);
31  start(2,Consumer);
    start(1,Producer);
33 }
```

Fig. 3. Modelling of synchronization via a shared buffer in SyncTask.

Each *ThreadType* consists of adjacent *SyncBlocks*, which are critical sections defined by a code block and 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.

The *Main* block contains the global variable declarations with initializations (*VarDecl**), and the thread composition (*StartThread**). A variable is defined by declaring 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 that it spawns.

Example 2 *(SyncTask program).* The program in Fig. 3 models synchronization via a shared buffer. `Producer` and `Consumer` represent the synchronization behaviour: 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 a bounded integer in the interval [0,7], with initial value set to `1`, and represents the number of elements in the buffer. One `Producer` and two `Consumer` threads are spawned with `start`.

Notice that this SyncTask program simulates the usage of a Java monitor since it uses a pair of lock and CV for synchronization. However, it could be more efficiently implemented with two CVs associated to the same lock: one to notify when the buffer is full, and another when it is empty. This alternative approach simulates the usage of `Condition` and `Lock` from the `java.util.concurrent` concurrency package.

2.2. Structural operational semantics

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

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 defined as $(Thread_id \times \mathbb{N}^{+}) \cup \perp$, which is either $\perp$ if the lock is free, or a pair of the id of the thread holding the lock, and a counter of how many times the lock was acquired by this thread.

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

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

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

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

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

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

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

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

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

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

[rs][h] $T|(\theta, t, (N, d, n)), \mu \rightarrow T|(\theta, t, R), \mu[lock(d) \mapsto (\theta, n)]$

[a] $\mu(o) = \bot$ [b] $\mu(o) = (\theta, n)$ [c] $\mu(o) = (\theta, n) \wedge n > 1$ [d] $\mu(o) = (\theta, 1)$

[e] $\mu(lock(d)) = (\theta, n)$ [f] $waitset(d) = \emptyset$ [g] $waitset(d) \neq \emptyset$ [h] $\mu(lock(d)) = \bot$

Fig. 4. Operational rules for synchronization.

A condition variable d only maps to its associated lock (*Lock* is the data domain); here is where the one-to-many relation from locks to CVs is defined. The auxiliary function $lock(d)$ returns the associated lock to d. Note that the set of threads waiting on a condition variable is *not* stored on the CV itself; below we define that this is stored at the thread state.

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, but the lock has not been reacquired yet. The states W and N also contain the CV d 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 $waitset(d)$ 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 thread 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., $T_W^d = \{(\theta, t, (W, d, n))\}$ 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. We say that a SyncTask program has a *correct synchronization* iff it terminates.

The initial configuration is defined with 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 $\mu(o) = \bot$. The thread composition is defined by the `start` declarations; e.g., `start(2,t)` adds two threads of type `t` to the thread composition: $(\theta, t, R)|(\theta', t, R)$.

Fig. 4 presents the operational rules, with superscripts $^{a-h}$ denoting conditions. Rule names with prefixes s, wt, nf, na and rs are short for synchronized, wait, notify, notifyAll and resume, respectively. We only define the rules for the synchronization statements, as the rules for the remaining statements are standard [12, § 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] decreases the counter upon terminating the execution of a synchronized block, but preserves the lock. 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 behaviour 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

Resource annotation:

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

Synchronization annotation:

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

Initialization annotation:

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

Fig. 5. Annotation language for Java programs.

to N. Rule [na2] is similar, but all threads in the wait set are awoken. By the rule [rs], 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`.

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, illustrate SyncTask extraction on an example, define our notion of *synchronization correctness* for Java programs, and characterize the notion as termination of the corresponding SyncTask program.

3.1. An annotation language and annotation scheme 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.

Fig. 5 presents the annotation language. Arguments given within square brackets are optional, allowing the programmer to (attempt to) leave their inference to STAVE, while text within parentheses tells which declaration types the annotation binds to. The programmer has to provide, by means of annotations, the following *three types of information*: resources, synchronization and initialization. Below, we describe these information types, and how they should be provided, i.e., our *annotation scheme*.

A *resource* annotates data types of variables that are manipulated by the synchronization and influence its progress, such as loop guards. The annotation defines an abstraction of the data structure state into a bounded integer, and how the methods operate on it. Potentially the bounded integer is a *ghost variable* (as in [13]), and in this case we say that the variable *extends* the program memory. For example, the annotation abstracts a linked list or a buffer to its size. More elaborated, compound data types may be annotated, such as stacks or lists containing elements from a bounded domain. However, if a thread's progress depends on an element's value, then the structure cannot be abstracted into a single bounded integer; instead, we require an initialization annotation (see below) for each element of the data structure.

Resources bind to classes only. The switch `@resource` starts the declaration. In case that a resource definition is spread across several classes (because of inheritance), it requires a common `ResourceId` for each annotated class. The `@object` keyword is optional and instructs STAVE that the data structure to analyze is a given variable or field in the annotated class. `@value` defines which class member, or ghost variable, stores the abstract state. Both allow an optional mapping to an alias *Sid*, which becomes mandatory in case the resource is defined in more than one class. `@capacity` defines the upper bound for `@value`. `@defaultval` and `@defaultcap` define the resource's default `@value` and `@capacity`, respectively; these may be overwritten in the *initialization annotation* (see below). 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 behaviour. `@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. The inline is potentially aided by `@maps` declarations, 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. Similarly to the `@resource` switch, a common `ThreadId` is required

```
01 class Producer extends Thread {
    Buffer pbuf;
03  Producer(Buffer b){pbuf=b;}
    public void run() {
05   /*@syncblock
     @monitor pbuf -> m
07   @resource pbuf:Buffer->b_els*/
     synchronized(pbuf) {
09    while (pbuf.full())
       pbuf.wait();
11    pbuf.add();
      pbuf.notifyAll();
13   }
    }
15 }
   class Consumer extends Thread {
17  Buffer cbuf;
    Consumer(Buffer b){cbuf=b;}
19  public void run() {
     /*@syncblock
21   @monitor cbuf -> m
     @resource cbuf:Buffer->b_els*/
23   synchronized(cbuf) {
      while (cbuf.empty())
25     cbuf.wait();
      cbuf.remove();
27    cbuf.notifyAll();
     }
29  }
   }
```

```
31 /*@resource @capacity cap
   @object els -> els
33 @value els -> els */
   class Buffer {
35  int els; final int cap;
    /* @operation @inline */
37  void remove(){if (els>0)els--;}
    /* @operation @inline */
39  void add(){if (els<cap)els++;}
    /* @predicate @inline */
41  boolean full(){return els==cap;}
    /* @predicate @inline */
43  boolean empty(){return els==0;}
    /*@synctask Buffer
45  @monitor b -> m
    @resource b:Buffer->b_els */
47  static void main(String[] s) {
     Buffer b = new Buffer();
49   b.els = 1; b.cap = 7;
     /* @thread */
51   Consumer c1 = new Consumer(b);
     /* @thread */
53   Consumer c2 = new Consumer(b);
     /* @thread */
55   Producer p = new Producer(b);
     c1.start();
57   p.start();
     c2.start();
59  }
   }
```

Fig. 6. Annotated Java program synchronizing via shared buffer.

in case the annotation is defined in more than one method or block. Nested, inner synchronization blocks and methods are not annotated; all the required information has to be provided at the top-level annotation. Here, `@resource` is *not* a switch, and thus has a different meaning. It defines that a local variable *Id* is a reference to a shared object of an (optional) annotated resource type (*ResourceId*), and is referenced by an alias `Sid` across other `@syncblock` declarations. The keywords `@lock` and `@condvar` define which mutex and condition variable object are observed. `@monitor` has the combined effect of both keywords for an object's monitor, i.e., a pair of a lock and a condition variable. Similarly to `@resource`, these require a mapping an alias that is common to other synchronization declarations.

Initialization annotations define the global pre-condition for the elements involved in the synchronization, i.e., they define initial values for locks, condition variables and resource declarations. They also define the global thread composition, i.e., how many and which type of threads participate in the synchronization. Initializations bind to methods, and the switch `@synctask` starts the declaration. Here, `@resource`, `@lock`, `@condvar` and `@monitor` instantiate with program variables the shared aliases defined at `@syncblock`. Finally, `@thread` defines that the following object corresponds to a spawned thread that synchronizes within the observed synchronization objects. The object's type is automatically detected, and must have been annotated with a synchronization annotation. Alternatively, the annotation can be followed by a thread type and a number indicating how many of these are spawned, so that the thread instantiation becomes less verbose.

Some of the above information STAVE is capable of inferring itself; the remaining information needs to be provided by the programmer. STAVE will always indicate when the provided hints are insufficient. This is discussed in more detail in Section 5.

Example 3 *(Annotated Java program).* The SyncTask program in Fig. 3 was generated from the Java program in Fig. 6. We now discuss how the annotations delimit the expected synchronization, indirectly illustrating the SyncTask extraction.

The `@syncblock` annotations (lines 5/20) add the following `synchronized` blocks to the observed synchronization behaviour, and its arguments `@monitor` and `@resource` (lines 6/21 and 7/22, respectively) map local references to shared aliases. The `@resource` annotation (line 31) starts the definition of a resource type. `@value`, `@object`, `@capacity` (lines 31/32/33) define how the abstract state is represented by a bounded integer. Here, to keep the running example simple, the abstract state has been chosen to be equal to the bounded integer `els`. However, in a typical buffer implementation the abstraction would be from the buffer content to a ghost variable containing the number of elements in the buffer. The `@operation` (lines 36/38) and `@predicate` (lines 40/42) annotations define how the methods operate on the state. Notice that the annotated methods have been inlined in Fig. 3, i.e., `add` is inlined in lines 6–10. 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.

The annotations provided in this example were sufficient for STAVE to infer that different variables that are spread along the code actually point to the relevant artifacts. Furthermore, STAVE was either able to infer or inline the other information it needed (methods' control flow, initializations, etc.), or the information was provided in the annotations.

```
static void main(String[] s) {          while (true) {
 Buffer b = new Buffer();                  c1 = new Consumer(b);
 b.els = 1;                                c2 = new Consumer(b);
 b.cap = 7;                                p = new Producer(b);
 Consumer c1;                              c1.start(); p.start(); c2.start();
 Consumer c2;                              c1.join(); c2.join(); p.join();
 Producer p;                               b.els = 1; b.cap = 7;
                                          }
                                         }
```

Fig. 7. Example of support for sessions.

Annotations can be understood as program *invariants* in the usual static analysis sense. That is, as control-point invariants which hold every time program execution is at a given control point (at which the annotation is placed). A program is then considered to be *correctly annotated* whenever the provided annotations hold. Although outside the scope of the present work, the annotations can potentially be checked, or partially generated, with existing static analysis techniques, such as [14, 4]. We shall henceforth assume that the programmer has correctly annotated the program. Furthermore, we shall assume the memory model of synchronized actions in a Java program to be sequentially consistent.

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 [5] 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, the notion easily generalizes to programs that operate in *sessions* by repeatedly re-spawning the synchronizing threads (i.e., the one-time synchronization scheme), provided that the synchronization variables are reset at the start of each session. Fig. 7 illustrates this notion with a modified version of the `main` method from Example 3.

We should stress that we use the term *correctness* here to refer exclusively to the property mentioned above; we do not refer with it to other undesirable synchronization phenomena, such as data race freedom.

Definition 1 *(Synchronization correctness).* Let $\mathcal{P}$ be a Java program with a one-time synchronization, where 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 exit point of the block.

We now connect synchronization schemes of correctly annotated Java programs with SyncTask programs.

Theorem 1 *(Characterization). 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 corresponding SyncTask program, and show it to be a *weak bisimulation* (see [15]) for a suitably chosen notion of observable and silent transitions between configurations. One aspect of the choice is that the annotations guarantee that the control flow of the original program is preserved, and thus, no infinite silent behaviours are possible within the synchronization. Therefore, a weak bisimulation relation is adequate and sufficient to establish the desired progress property. We refer to the accompanying technical report [16] for the full formalization and for the most interesting proof 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 their *common* variables. Similarly, we define the set of *observable transitions* as the ones that update common variables, and treat all remaining 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 3.1. The proof of (*ii*) requires to show that updates to a common variable yield the same result in both programs. This 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 [12], the elaboration of this is straightforward. □

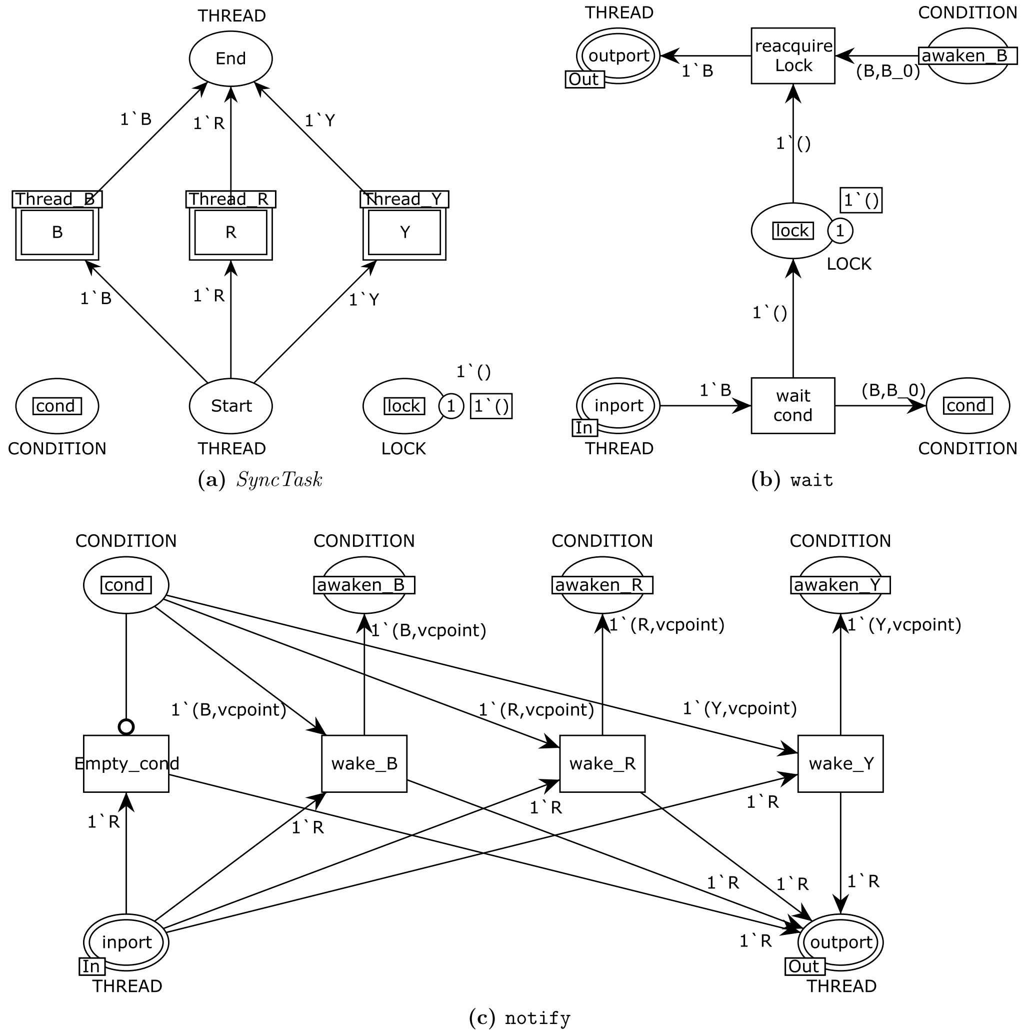


Fig. 8. Top-level component and condition variables operations.

4. Verification of synchronization correctness

In this section we show how termination of SyncTask programs can be reduced to a reachability problem on Coloured Petri Nets (CPN).

4.1. SyncTask programs as Coloured 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 are supported by analysis tools such as CPN Tools, and allow a natural translation of common language constructs into CPN components. For this we reuse results from Westergaard [9], and only had to model the constructs involving CVs that we present below. We assume some familiarity with CPNs, and refer the reader to [8] for a detailed exposition.

The colour set THREAD associates a colour to each `Thread` type declaration, and a 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 Fig. 2 are mapped into hierarchical CPN components, where *substitute transitions* (STs; depicted as doubly outlined rectangles) represent the non-terminals on the right-hand side. Fig. 8a shows the component for the start symbol *SyncTask*. The Start place contains all thread tokens in the initial configuration, connected by arcs (one per colour)

to the STs denoting the thread types, and End, which collects the terminated thread tokens. It also contains the places that represent global variables.

Fig. 8b 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. Fig. 8c shows the modelling of `notify`. The `Empty_cond` transition is enabled if the CV is empty, and the other transitions, with one place per colour, 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 colours of thread tokens. A `Lock` creates a place containing a single token; it being empty represents that some thread holds the lock. The colour set CPOINT represents the control points of `wait` statements. A `Condition` variable gives rise to an empty place representing the waiting set, with colour set CONDITION. Here, colours 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.

5. The STaVe tool

In this section we present the implementation of our tool, discuss its capabilities to infer some of the information needed for the translation to SyncTask, and present the results of our experimental evaluation.

5.1. Implementation

We have implemented the parsing of annotated Java programs to generate SyncTask programs, and the extraction of hierarchical CPNs from SyncTask, as the STaVe tool. It has been written in Java, and is available at [17].

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. For instance, the use of JavaParser allows the annotation of `synchronized` blocks. Next, STaVe traverses the Java AST three times to extract the SyncTask program's AST. The first pass processes resource annotations, and extracts information about how threads operate on shared variables. The second pass processes synchronization annotations, and uses the information from the previous pass to generate the control flow structure of the threads. The third pass processes initialization annotations, and checks if the declared variables and thread types have been properly parsed in the previous steps. After the SyncTask AST is created, it is traversed following the mapping described in Section 4 to generate the corresponding CPN.

Two parts of STaVe turned out to be useful in itself, i.e., useful for other projects. 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.

[3] Available at https://github.com/pcgomes/javaparser2jctree.
[4] Available at https://github.com/pcgomes/libcpntools.

5.2. Static analysis

Some of the information about the synchronization behaviour of the analyzed program, which is needed for the extraction of the SyncTask program, can be deduced by STAVE itself. Basically, this is the information which the Java compiler can deduce. Thus, the tool can automatically (the examples in parentheses refer to Fig. 6):

- deduce initialization involving constants: the number of threads, a resource capacity, etc. (lines 50–55);
- deduce simple control-flow of the synchronization blocks, including the case of method invocations without recursion;
- name a SyncTask construct from its originating Java counterpart, as for instance, an annotated synchronized block will be named after the Java class that defines it (class `Consumer`);
- assign automatically a label to variables with the same name and type, even if declared and used in distinct files and/or methods;
- infer information that involves the class hierarchy, as for instance, it is able to understand a "resource" that has some methods defined in a parent class, while other methods in the annotated class.

Our tool could be extended with several additional, specialized static analyses that would automate the inference of various types of information, needed for the translation to SyncTask. The main candidate would be a *pointer analysis*, which would infer when two variables in distinct parts of the code invariably point to the same object. Currently the tool requires the user to "tie" such variables using labels. That is, the user manually assigns a global label to a Java variable, and the label will become the name of the respective SyncTask variable. For instance, lines 6, 21 and 45 in Fig. 6 define that the Java variables named `buffer`, `buffer` and `b` in their respective methods, actually reference the same object `m` (which is a label to refer to that object).

5.3. Experimental evaluation

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 usage of STAVE and the annotation process in a real-world program. For this, we annotated PIPE [18] (version 4.3.2), a rather large CPN analysis tool written in Java. It contains a single (and simple) synchronization scheme with two threads using CVs: when there is a new connection attempt from a remote client, a thread establishes the connection and then notifies the shared CV; the other thread writes logs to the client, and waits on the CV if the socket is not ready. This test case illustrates that synchronization involving CVs is typically simple and bounded. It also exemplifies a session synchronization since the only variable, a boolean that flags if the socket is ready, has the same value (false) at the start of each session. We stress, however, that STAVE analyzes it as being a one-time synchronization. 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 correctness.

Our second test case evaluates the scalability of our approach using STAVE and state-space exploration (with CPN Tools) w.r.t. the number of threads. We took Example 3, and instantiated it with a varying number of threads, buffer capacity, and initial value.

As a reference, we used Java Pathfinder to analyze the same program. Java Pathfinder [19] (JPF) is an obvious choice for analyzing Java programs with wait/notify, as it can detect the same types of deadlock (lack of progress) that STAVE analyzes. JPF supports the full bytecode instruction set and can analyze the full state space of concurrent applications that have no native methods (methods that execute machine code libraries on the host system). For native methods, model classes can be provided to replace them with equivalent code in Java, but this is often a complex task [20].

When using STAVE, its back-end, CPN Tools, generated the state graph, which we later queried using its ML-based API [21]. We remark that, different from the preliminary version of this paper [22], here we take into account the time of a mandatory initialization phase called *Enter the State Space*. As expected, this leads to higher verification times. As before, we collect our statistics by considering the state-space generation, computation of the strongly connected components, and verification of the three termination conditions. Namely: whether there is at least one dead configuration; whether, for all dead configurations, the End place has the same marking as the Start place in the initial configuration; and whether the number of strongly connected components is equal to the size of the state graph, implying the absence of cycles.

The experiments were executed in a Linux machine with 16 GB of RAM and a quad-core Intel i5 CPU of 1.30 GHz. The JPF experiments were executed with version 8.0 rev 32, on Java 1.8.0_121. We gave JPF 4 GB of heap space (an amount that was never fully used) and ran the experiments without a timeout of one hour. In addition to the execution times, JPF shows the number of explored states and the number of executed bytecode instructions. The CPN Tools experiments were performed with version 4.0.1 in a Windows 7 virtual machine running under VirtualBox version 5.1.32 with 8 GB of RAM and 2 processors.

Table 1
Statistics for Producer/Consumer. For given configurations, the number of program states and the analysis time is shown for both tools. For Java Pathfinder, we also show the number of bytecode instructions executed during the whole analysis.

Problem size				Analysis results					
Threads		Buffer			Java Pathfinder			STaVe/CPN tools	
producers	consumers	capacity	elements	terminates?	# states	# instr.	time [mm:ss]	# states	time [mm:ss]
1	2	1	1	yes	1,466	43,603	0:01	42	0:05
1	2	2	0	no	22	3,878	0:00	43	0:05
2	2	1	0	yes	10,533	294,823	0:03	91	0:05
3	3	1	0	yes	613,052	21,035,480	2:12	283	0:05
4	3	1	0	yes	4,864,766	187,705,560	20:08	448	0:05
4	3	1	1	no	64	4,754	0:00	440	0:06
6	5	1	0	yes	timeout after one hour			2,152	0:07
6	5	1	1	no	122	5,740	0:00	2,131	0:05
6	5	5	1	yes	timeout after one hour			950	0:06
6	5	5	4	yes	timeout after one hour			968	0:05
7	1	5	0	no	74	4,946	0:00	157	0:05
7	6	1	1	no	154	6,260	0:00	3,938	0:06
7	6	7	1	yes	timeout after one hour			1,395	0:06
11	11	1	0	yes	timeout after one hour			29,143	0:18
11	9	7	6	no	172	7,564	0:00	6,573	0:07
14	13	1	1	no	434	10,404	0:00	64,075	0:51
14	13	7	1	yes	timeout after one hour			29,573	0:16
16	21	5	5	yes	timeout after one hour			164,921	3:48
17	16	16	16	no	131	10,077	0:00	24,833	0:13
18	18	1	1	yes	timeout after one hour			197,563	5:25
18	18	5	1	yes	timeout after one hour			133,824	2:34
20	18	2	1	no	704	14,120	0:00	217,702	6:09
22	21	16	16	no	364	12,590	0:00	84,603	0:51
26	24	25	24	no	199	13,615	0:00	78,191	0:39

Table 1 presents the practical evaluation for a number of initial configurations with varying number of threads (*Producer* and *Consumer*), buffer *capacity* and position[5] (*elements*). Column *terminates?* shows if an initial program configuration has correct synchronization w.r.t. Definition 1. For the cases where JPF timed out, the presented results come from the STaVe/CPN tools analysis only. As expected, the other results match and come from both analysis. The term *state* replaces *CPN configuration* at STaVe statistics to avoid confusion with the concept shown in *Problem size*, and to facilitate the comparison between the state-space sizes. Times presented as 0:00 mean *less than one second*.

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 which model high contention, i.e., many threads waiting on CVs, induce a larger state space. This effect is particularly strong with Java Pathfinder, which has to execute all relevant configurations explicitly as program code. 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.

5.4. State space explosion with Java Pathfinder

To confirm the trend of sharply exploding state spaces for unfalsifiable instances, we ran JPF with a number of additional configurations of Example 3.

5.4.1. Correct configurations

Table 2 shows configurations in which JPF detected no errors. We tested an initial configuration with a various number of producer and consumer threads and various buffer sizes, with one initial element. While the total state space in configurations of up to six threads in total is easily tractable (JPF takes between a few seconds and two minutes), larger configurations are problematic. Configurations with seven threads took between 15 and 20 minutes to complete, while eight threads could not complete within one hour.

[5] As defined in https://docs.oracle.com/javase/8/docs/api/java/nio/Buffer.html.

Table 2
Run times of JPF to analyze the full state space in various scenarios. The buffer was filled with one element in its initial state. We show scenarios with very similar outcomes by showing the range of parameters and measured results (as minimum... maximum values).

Threads		Buffer	Analysis result		
prod.	cons.	capacity	# states	# instructions	time [mm:ss]
1	1	1...10	186... 230	8,112... 8,887	0:00... 0:00
1	2	1...10	1,466... 1,608	43,597... 46,186	0:01... 0:01
2	1	2...10	1,744... 1,844	52,638... 54,697	0:01... 0:01
2	2	1...10	10,981... 12,449	339,300... 387,612	0:03... 0:04
2	3	1...10	82,806... 83,396	2,714,476... 2,814,139	0:17... 0:19
3	1	3...10	12,825... 13,045	418,015... 423,135	0:04... 0:04
3	2	2...10	86,701... 90,241	3,082,752... 3,125,704	0:18... 0:19
3	3	1...10	585,200... 646,643	22,420,306... 23,968,679	2:11... 2:23
3	4	1...10	3,963,321...4,735,873	161,745,713...183,001,505	18:11...21:01
4	1	4...10	85,277... 85,753	3,119,794... 3,132,367	0:18... 0:18
4	2	3...10	563,183... 589,886	22,484,301... 23,038,076	2:08... 2:14
4	3	2...10	3,820,353...4,644,356	163,788,830...189,499,018	16:22...19:53
4	4	1		timeout after one hour	
5	1	5...10	536,276... 537,296	21,800,221... 21,830,503	2:02... 2:02
5	2	4...10	3,491,907...3,600,149	153,771,259...156,364,195	15:14...15:24
5	3	3		timeout after one hour	

Table 3
Run times of JPF to detect faults in various scenarios. The buffer was filled with one element in its initial state. We show scenarios with very similar results by showing the range of parameters and measured results (as minimum... maximum values).

Threads		Buffer	Analysis result			
prod.	cons.	capacity	trace length	# states	# instructions	time [mm:ss]
1	3...10	1...10	25...74	26...75	4,078...5,520	0:00
2	1	1	10	11	3,831	0:00
3	1	1...2	14	15	4,020...4,041	0:00
3	2	1	37	38	4,282	0:00
4	1	1...3	18...20	19...21	4,211...4,252	0:00
4	2	1...2	43...47	44...48	4,473...4,498	0:00
4	3	1	63	64	4,748	0:00
5	1	1...4	22...26	23...27	4,402...4,461	0:00
5	2	1...3	49...57	50...58	4,664...4,714	0:00
5	3	1...2	69...73	70...74	4,939...4,964	0:00
5	4	1	91	92	5,232	0:00
			...			
10	9	1	261	262	7,922	0:00

5.4.2. Faulty configurations

Table 3 shows configurations in which JPF detected a deadlock, where a producer or consumer thread could not proceed because the buffer was either full or empty, respectively, and no active threads that could change that condition were available. We tested an initial configuration with a various number of producer and consumer threads and various buffer sizes, with one initial element.

A larger number of threads increases the state space only slightly; this is mostly visible by a longer error trace in cases where more threads are involved. Still, the number of states is always small, and JPF finds the error right away, as shown by the very small number of instructions executed, and a run time that was always below one second (see Table 3).

Therefore, it can be seen that JPF is very effective at finding defects, and competitive with SyncTask in terms of run-time in cases where defects are present. For cases that are correct, JPF scales to a couple of threads, but it fails if the number of threads grows larger. Given that no annotations are required for JPF, it is therefore a good choice to try an example in JPF first, before annotating it to try to prove liveness in larger cases.

6. Related work

We present related methods and tools that are based on the following approaches:

1. software model checking, a systematic analysis of all possible outcomes by executing the software under all schedules;
2. deductive reasoning, using compositional techniques to reason about the behaviour of concurrent programs;
3. abstract interpretation, and in particular thread-modular treatments;
4. schedule synthesis and permutation, where a safe schedule is to be found, or subsets of all thread interleavings are investigated;
5. and a conversion of the program structure to Petri Nets.

6.1. Approaches based on software model checking

Java Pathfinder [19] is closely related to our work in that it checks all possible outcomes of different thread interleavings of a concurrent Java program. By default, it checks whether any assertion failure or uncaught exception occurs, and whether a program exhibits a deadlock state, which is a state where at least one active thread exists that cannot continue because it is blocked on a resource. A thread may block on a resource because it may wait for input from a file or network channel, try to obtain a lock, or wait for a signal inside `wait`. The latter type of deadlock corresponds to the one analyzed by STAVE.

Java Pathfinder optimizes the state space search by matching equivalent program states and by ignoring interleavings that do not affect the global program state [19]. Unlike our tool, Java Pathfinder executes the full bytecode of the Java application under test, so it generally does not scale to programs with many threads. However, by executing the actual bytecode, it does not require annotations to check against livelocks in programs using condition variables (CVs). A drawback of Java Pathfinder is that it cannot execute native methods. Large applications typically need elaborate model libraries to execute functionality such as network communication [20], whereas STAVE only considers annotations, which can be modelled to take into account any complex libraries.

In principle, Java Pathfinder could handle a simplified program (equivalent to the SyncTask program) better than the full program, because the abstraction would eliminate native code and reduce the complexity of the program. It may be possible to isolate subsets of the full program by using the SyncTask annotations, but this is left as future work.

Musuvathi et al. [23] present `CHESS`, a tool that systematically tests thread interleaving to try to uncover subtle concurrency bugs. The tool supports the Windows 32 API, which features CVs. Our work shares similarities to this one, such as the exploration of the space of thread interleaving. However, `CHESS` is concerned with program safety, i.e., a program shall not reach an error state. The present work, on the other hand, focus on a liveness property, i.e., every waiting thread will eventually be notified and progress.

6.2. Approaches based on deductive reasoning

Leino et al. [4] 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 [5] 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.

6.3. Approaches based on abstract interpretation

A powerful framework for the static analysis of programs is *abstract interpretation*, which allows programs to be (abstractly) executed in specialized abstract domains to obtain algorithmically sound facts about their behaviour. The framework is flexible in that it allows precision of the analyses to be traded for performance, and *vice versa*.

To deal with the combinatorial explosion of multi-threaded programs, some works develop *thread-modular* analyses to achieve scalability. Miné [24] for instance, considers locks (mutexes) as explicit synchronization primitives, and includes a yield statement. The locks are not reentrant: acquiring an already acquired lock has no effect, and similarly releasing a lock that is not acquired by a thread. No procedures are considered (but inlining can be used for non-recursive procedural programs), and no dynamic thread creation. The aim of the proposed method is to discover data races.

In recent follow-up work, Monat and Miné [25] extend the analysis to relational domains, in a flow-sensitive manner, to achieve a higher precision. The focus of the work is on numeric properties of small, but intricate mutual exclusion algorithms. The experimental results show that the method scales well, and allows the analysis of several hundreds of (small) threads.

Other works also use a thread-modular analysis to detect potentially unsafe accesses. High-level data races denote unsafe access patterns to tuples of values [26]. Local atomicity violations denote unsafe uses of shared data [27,28]. Both types of atomicity violations have recently been unified [29]. Atomicity violations show that the value of a CV may not always be correct w. r. t. the global state of the program.

Another analysis that is close to ours is a data race detection tool based on key concurrency operations extracted from the given program [30]. Similarly to our tool, that approach builds an abstract model that contains all relevant concurrency operations on shared data. Like STaVe's analysis, theirs is not completely thread-modular.

As already mentioned, one strong point of the above-mentioned methods is that most of them are thread-modular. The mutual dependencies are handled by data-flow analysis or rely-guarantee style reasoning, which means that an iterative fixed-point computation is performed that invokes the thread-modular analyses on the threads in rounds, until global stabilization.

However, data race and atomicity analyses do not cover the signalling between threads, and therefore do not completely cover the semantics of CVs. Since wait-notify synchronization is inherently non-local, it does not lend itself naturally to completely thread-modular analyses. Furthermore, it is not obvious how the analysis has to be set up to compute the interferences (as the local effects are called) in the case of CVs, and how precise this can be made.

6.4. Schedule synthesis and permutation

Raychev et al. [7] present an algorithm that takes as input a non-deterministic parallel program, and synthesizes a synchronization specification using CVs (and other synchronization primitives) so that the program becomes deterministic, in the sense that it produces the same output for the same input, regardless of the scheduling. This work differs substantially from ours since we do not focus on deterministic programs (in the above sense), and we extract a synchronization specification rather than create one. However, the two works share similarities. For instance, both focus on programs with constant number of threads due to the complexity of reasoning about the asynchronous signalling of CVs. Also, they abstract away from other sources of non-determinism than thread interleaving.

Wang and Hoang [31] 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.

6.5. Conversion to Petri Nets

Kaiser and Pradat-Peyre [32] 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. [33] 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 [9] 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. [34] 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.

7. Conclusion

We present 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 behaviour.

We introduce 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 Coloured 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. Experiments show that our approach scales well to programs with many threads, at the expense of requiring detailed annotations of the original Java program.

Our current results hold for a number of *restrictions* on the analyzed programs. In future work we plan to address and relax these restrictions, integrate special-purpose static analyzers 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] C.A.R. Hoare, Monitors: an operating system structuring concept, Commun. ACM 17 (10) (1974) 549–557, https://doi.org/10.1145/355620.361161.
[2] International Organization for Standardization, Information Technology – Programming Languages – C++, Standard, International Organization for Standardization, Sep. 2011.
[3] J. Gosling, B. Joy, G.L. Steele, G. Bracha, A. Buckley, The Java Language Specification, Java SE 8 Edition, 1st edition, Addison–Wesley Professional, 2014.
[4] K.R.M. Leino, P. Müller, J. Smans, Deadlock-free channels and locks, in: European Conference on Programming Languages and Systems, ESOP'10, Springer-Verlag, 2010, pp. 407–426.
[5] C. Popeea, A. Rybalchenko, Compositional termination proofs for multi-threaded programs, in: Tools and Algorithms for the Construction and Analysis of Systems, TACAS'12, Springer-Verlag, 2012, pp. 237–251.
[6] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, SIGPLAN Not. 43 (3) (2008) 329–339, https://doi.org/10.1145/1353536.1346323.
[7] V. Raychev, M. Vechev, E. Yahav, Automatic synthesis of deterministic concurrency, in: F. Logozzo, M. Fähndrich (Eds.), Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20–22, 2013, Proceedings, Springer, Berlin, Heidelberg, 2013, pp. 283–303.
[8] K. Jensen, L.M. Kristensen, Coloured Petri Nets: Modelling and Validation of Concurrent Systems, 1st edition, Springer Publishing Company, Incorporated, 2009.
[9] M. Westergaard, Verifying parallel algorithms and programs using coloured Petri nets, in: Transactions on Petri Nets and Other Models of Concurrency VI, in: Lecture Notes in Computer Science, vol. 7400, Springer, Berlin, Heidelberg, 2012, pp. 146–168.
[10] K. Jensen, L. Kristensen, L. Wells, Coloured Petri nets and CPN tools for modelling and validation of concurrent systems, Int. J. Softw. Tools Technol. Transf. 9 (3–4) (2007) 213–254, https://doi.org/10.1007/s10009-007-0038-x.
[11] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Trans. Comput. 28 (9) (1979) 690–691, https://doi.org/10.1109/TC.1979.1675439.
[12] P. Cenciarelli, A. Knapp, B. Reus, M. Wirsing, An event-based structural operational semantics of multi-threaded Java, in: Formal Syntax and Semantics of Java, in: Lecture Notes in Computer Science, vol. 1523, Springer, Berlin, Heidelberg, 1999, pp. 157–200.
[13] G. Leavens, A. Baker, C. Ruby, JML: a notation for detailed design, in: H. Kilov, B. Rumpe, I. Simmonds (Eds.), Behavioral Specifications of Businesses and Systems, in: Eng. Comp. Sci., vol. 523, Springer US, 1999, pp. 175–188.
[14] K.R. Leino, P. Müller, A basis for verifying multi-threaded programs, in: Proceedings of the 18th European Symposium on Programming Languages and Systems, ESOP '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 378–393.
[15] R. Milner, Communicating and Mobile Systems: The π-Calculus, Cambridge University Press, New York, NY, USA, 1999, pp. 52–53, Ch. 6.
[16] P. de Carvalho Gomes, D. Gurov, M. Huisman, 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.
[17] P. Gomes, SyncTAsk VErifier, http://www.csc.kth.se/~pedrodcg/stave, 2015.
[18] N.J. Dingle, W.J. Knottenbelt, T. Suto, PIPE2: a tool for the performance evaluation of generalised stochastic Petri nets, SIGMETRICS 36 (4) (2009) 34–39, https://doi.org/10.1145/1530873.1530881.
[19] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, Model checking programs, Autom. Softw. Eng. J. 10 (2) (2003) 203–232, https://doi.org/10.1023/A:1022920129859.
[20] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, M. Yamamoto, K. Takahashi, Modular software model checking for distributed systems, IEEE Trans. Softw. Eng. 40 (5) (2014) 483–501.
[21] K. Jensen, S. Christensen, L.M. Kristensen, CPN Tools State Space Manual, Tech. rep., Department of Computer Science, University of Aarhus, 2006, http://cpntools.org/_media/documentation/manual.pdf.
[22] P. de Carvalho Gomes, D. Gurov, M. Huisman, Specification and Verification of Synchronization with Condition Variables, Springer International Publishing, Cham, 2017, pp. 3–19.
[23] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, I. Neamtiu, Finding and reproducing Heisenbugs in concurrent programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 267–280, http://dl.acm.org/citation.cfm?id=1855741.1855760.
[24] A. Miné, Static analysis of run-time errors in embedded real-time parallel C programs, Log. Methods Comput. Sci. 8 (1) (2012) 483–501, https://doi.org/10.2168/LMCS-8(1:26)2012.
[25] R. Monat, A. Miné, Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions, in: Verification, Model Checking, and Abstract Interpretation, VMCAI 2017, in: Lecture Notes in Computer Science, vol. 10145, Springer, 2017, pp. 386–404.
[26] C. Artho, K. Havelund, A. Biere, High-level data races, J. Softw. Test. Verif. Reliab. 13 (4) (2003) 220–227.
[27] C. Artho, A. Biere, K. Havelund, Using block-local atomicity to detect stale-value concurrency errors, in: Proc. 2nd Int. Symposium on Automated Technology for Verification and Analysis, ATVA 2004, in: Lecture Notes in Computer Science, vol. 3299, Springer, Taipei, Taiwan, 2004, pp. 150–164.
[28] C. Flanagan, S.N. Freund, Atomizer: a dynamic atomicity checker for multithreaded programs, ACM SIGPLAN Not. 39 (1) (2004) 256–267.
[29] R.J. Dias, V. Pessanha, J.M. Lourenço, Precise detection of atomicity violations, in: Haifa Verification Conference, in: Lecture Notes in Computer Science, vol. 7857, Springer, 2012, pp. 8–23.
[30] J. Mund, R. Huuck, A. Fehnker, C. Artho, The quest for precision: a layered approach for data race detection in static analysis, in: Proc. 11th Int. Symposium on Automated Technology for Verification and Analysis, vol. 8172, ATVA 2013, Hanoi, Vietnam, 2013, pp. 516–525.
[31] C. Wang, K. Hoang, Precisely deciding control state reachability in concurrent traces with limited observability, in: Verification, Model Checking, and Abstract Interpretation, in: Lecture Notes in Computer Science, vol. 8318, Springer, Berlin, Heidelberg, 2014, pp. 376–394.
[32] C. Kaiser, J.-F. Pradat-Peyre, Weak fairness semantic drawbacks in Java multithreading, in: Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies, Springer-Verlag, 2009, pp. 90–104.
[33] K. Kavi, A. Moshtaghi, D.-j. Chen, Modeling multithreaded applications using Petri nets, Int. J. Parallel Program. 30 (5) (2002) 353–371, https://doi.org/10.1023/A:1019917329895.
[34] W. van der Aalst, C. Stahl, M. Westergaard, Strategies for modeling complex processes using colored Petri nets, in: Transactions on Petri Nets and Other Models of Concurrency VII, in: Lecture Notes in Computer Science, vol. 7480, Springer, Berlin, Heidelberg, 2013, pp. 6–55.