Encoding Deadlock-Free Monitors in the VerCors Verification Tool

Matthijs Roelink
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.j.roelink@student.utwente.nl

ABSTRACT

When developing a concurrent program, a deadlock is never the intended result. However, avoiding them is often attributed to the experience of the developer, as a compiler is generally not able to detect them. Recently, a technique has been proposed to verify deadlock-freeness of a program with monitors. The aim of this research is to investigate how this technique can be encoded in the VerCors verification tool to verify deadlock-freeness of Java-like programs. This paper specifies the required annotation syntax and describes the implementation of the technique in VerCors.

Keywords

Deadlock, Monitor, Verification, Concurrency, VerCors

1. INTRODUCTION

In concurrent programs, synchronization of threads is a widely used construct. It allows controlled execution of critical sections, often only accessible by one thread at a time.

One such synchronization construct is a monitor. Monitors enforce mutual exclusion, thus ensuring that only one thread can execute the critical section protected by the monitor. In general, a monitor consists of a resource and a condition variable. Often, this resource is a lock. In addition, a monitor has at least two operations: `wait` and `signal` [15]. The former operation releases the resource and waits until it is signaled, while the latter operation signals a waiting thread. A thread can only call these operations if it has acquired the resource.

One caveat of monitors is the risk of deadlocks. A deadlock occurs when two or more threads have a cyclic locking dependency, thus indefinitely waiting for each other [12]. This may cause (parts of) the program to block. Without any precautions, the only way to recover from this situation is by restarting the program, which is often inconvenient. As threads in concurrent programs can have many different interleavings, it might happen that a deadlock occurs under very specific circumstances and is not detected before going into production. Naturally, deadlock-free assurance of a program is crucial.

33rd Twente Student Conference on IT July 3rd, 2020, Enschede, The Netherlands.

To enable the verification of concurrent programs, the VerCors verification tool was developed [6]. It is able to verify parallel and concurrent features of programs written in Java, C, PVL [26], OpenCL and OpenMP [5]. Programmers can annotate their code, which is then used by VerCors for verification. Although VerCors supports a wide range of verification features, such as data-race freedom and functional correctness of programs written in the aforementioned languages, it does not yet support verification of absence of deadlocks.

Examples of other verification tools are VeriFast [16] and VCC [7]. VerCors distinguishes itself from these other verification tools by first compiling the original program to an intermediate program written in Silver [20] and then verifying this intermediate program using the Viper framework [20]. This allows verification of any program written in a language that can be compiled to Silver [5].

Verification of deadlock-freeness of a program that uses locks is already a solved problem [13, 17]. When using locks, a deadlock occurs when there is not a strict locking order. To mitigate a deadlock in such a scenario, one can impose a specific lock order. This ensures that the locks are always acquired in the same order and will never have a cyclic dependency. Verification of absence of deadlocks then merely consists of verifying if this lock order is enforced.

This approach does not work for programs that use monitors, however, as threads can acquire and release locks dynamically using the `wait` and `signal` operations. In such programs, a deadlock occurs when a thread that is waiting for a condition variable is never notified or, when it is notified, is not able to acquire the associated lock.

There are several approaches to detect and avoid [1, 11, 25] or recover [2] from deadlocks during runtime, but these do not verify that a program is indeed deadlock-free beforehand. Other approaches present methods to verify the absence of deadlocks of programs that use channels [19] and locks [13, 17]. These approaches cannot be applied to monitors, however, since monitors have the property that signals can be missed if no thread is waiting for the condition variable [14]. Gomes et al. [8] describe the verification of synchronization with condition variables, but put restrictions on the programs that can be verified.

A more promising technique to verify the absence of deadlocks in programs that use monitors is described in [14]. The proposed approach is modular and is generalized for all monitor applications. In addition, the authors managed to implement this technique in the VeriFast program verifier [16]. The technique uses separation logic [21] to verify the absence of deadlocks, which is supported by VerCors in an extended form [3, 4].

The aim of this research is to investigate how the technique described in [14] can be encoded in VerCors to verify deadlock-freeness of Java-like programs that use monitors.

This paper is structured as follows. Section 2 gives background information on the approach presented in [14] and on the VerCors verification tool. Section 3 defines the annotation syntax necessary for the verification. Section 4 explains the implementation in VerCors. The validation of the implementation is discussed in Section 5. Section 6 shows an example of an annotated Java program that uses the annotations defined in section 3. Finally, Section 7 concludes the paper and Section 8 proposes future work.

2. BACKGROUND

2.1 Deadlock-Free Monitors

Hamin et al. [14] describe a technique to verify deadlock-freeness, which uses separation logic [21] to reason about deadlock-freeness of a program. Their approach is modular, meaning that separate parts of the program can be verified independently of each other. In addition, the approach does not suffer from long verification time when the state space size increases. Although the authors generalize their technique for other applications of condition variables as well, this research only focuses on the verification of monitors.

The central idea of the technique is the notion of obligations, introduced in [19]. There can be obligations for both locks and condition variables. A thread is assigned an obligation for a lock if it acquires that lock. The thread can discharge this obligation by releasing the lock.

An obligation for a condition variable can be arbitrarily assigned to a thread. Discharging an obligation for a condition variable v, however, is only allowed if either no threads are waiting for v or there is a thread that has an obligation for v. There is not a fixed moment when an obligation is assigned and discharged for a condition variable. It depends on the context of the program and is the programmer's responsibility.

Formally, Wt is defined as the bag of threads that are waiting for a condition variable. This bag is a mapping from a condition variable v to the number of threads that are waiting for v and is stored per monitor. The number of threads waiting for v can be queried with $Wt(v)$.

To keep track of the obligations of all threads, the bag Ot is defined. Ot is a mapping from a condition variable v to the number of threads that have an obligation for v. Similar to Wt, the bag is stored per monitor and the number of threads that have an obligation for v can be queried with $Ot(v)$.

Each lock associated with a condition variable v should have an invariant that implies the invariant $enoughObs(v, Wt, Ot)$ defined as

$$enoughObs(v, Wt, Ot) = (Wt(v) > 0 \implies Ot(v) > 0)$$

meaning that either no threads should be waiting for v or there should exist a thread that has an obligation for v.

Next to the bags Wt and Ot, each thread keeps track of its own obligations using a local bag $obs(O)$. When an obligation for a lock or condition variable is loaded into Ot, it should also be loaded into $obs(O)$ of the thread. Similarly, when an obligation for a lock or condition variable is discharged from Ot, it should also be discharged from $obs(O)$ of the thread.

In addition, each lock and condition variable is assigned a *wait level*. A thread is only allowed to acquire a lock or wait for a condition variable if the wait level of that lock/condition variable is the lowest of all wait levels of the obligations of that thread. This is denoted by $o \prec obs(O)$, where o is the lock or condition variable. The wait levels ensure that there are no cyclic dependencies.

Now that the necessary utilities are defined, we can reason about the absence of deadlocks in a program by following certain rules. The rules are systematically shown in Table 1 and ensure that:

1. When a thread executes `wait()` on a condition variable v, there should exist a thread that has an obligation for v.

2. A thread only discharges an obligation for a condition variable v if either no threads are waiting for v or there exists a thread that has an obligation for v.

3. A thread only executes `wait()` on a condition variable v if the wait level of v is lower than the wait levels of all obligations of that thread.

A program begins with an empty bag and should also end with an empty bag. If in the end the bag is not empty, i.e., there is still a thread that has an obligation (and thus a thread that is still waiting), then the program is not deadlock-free.

When a thread t wants to acquire a lock l, l must have the lowest wait level of all obligations of t: $l \prec obs(O)$. Note that this relation may be relaxed to $l \preceq obs(O)$ if t already acquired l, since the intrinsic lock of a monitor is reentrant. If t has successfully acquired l, l is added to the obligations of t. The obligations of t are now $obs(O \uplus \{l\})$.

A thread t can release a lock l without any preconditions, providing that t has acquired l. When t releases l, l is removed from the obligations of t (i.e., $obs(O - \{l\})$).

A thread t can load n obligations for a condition variable v arbitrarily. These obligations are then added to the local bag of obligations ($obs(O \uplus \{n * v^1\})$) and to the global bag of obligations ($Ot \uplus \{n * v\}$).

A thread t can discharge n obligations for a condition variable v arbitrarily if after the discharge, the rule $enoughObs(v, Wt, Ot)$ holds. Therefore, before discharging the obligations, $enoughObs(v, Wt, Ot - \{n*v\})$ should hold, as n instances of v will be removed from Ot. The n obligations for v are discharged from the local bag of obligations of t ($obs(O - \{n * v\})$) as well as from the global bag of obligations ($Ot - \{n * v\}$).

Before a thread t can wait for a condition variable v that is protected by a lock l, t should have acquired l and l should be present in the local bag of obligations of t ($obs(O \uplus \{l\})$). In addition, the wait levels of v and l should be lower than the wait levels of all obligations of t ($v \prec O$ and $l \prec O$). $l \prec O$ is necessary because when t is notified, it will try to reacquire l. Since l is going to be released by calling `v.wait`, it is not necessary that the wait level of v is lower than the wait level of l. Finally, as the number of threads waiting for v will be incremented, the invariant $enoughObs(v, Wt \uplus \{v\}, Ot)$ should hold. After calling `v.wait()` and before suspension, one instance of v is added to Wt.

When a thread executes `notify()` on a condition variable v, an arbitrary thread that is waiting for v (if there is one)

[1]In this context, $n * v$ means n instances of v.

Table 1. Rules for deadlock-free monitors

Action	**Preconditions**	**Postconditions**
When a thread t acquires a lock l	• $l \prec$ *obs(O)*	• $\mathit{obs}(O \uplus \{l\})$
When a thread t releases a lock l		• $\mathit{obs}(O - \{l\})$
When a thread t loads n obligations for a condition variable v		• $\mathit{obs}(O \uplus \{n * v\})$ • $\mathit{Ot} \uplus \{n * v\}$
When a thread t discharges n obligations for a condition variable v	• $\mathit{enoughObs}(v,\ \mathit{Wt},\ \mathit{Ot} - \{n * v\})$	• $\mathit{obs}(O - \{n * v\})$ • $\mathit{Ot} - \{n * v\}$
When a thread t waits for a condition variable v that is protected by a lock l	• $\mathit{obs}(O \uplus \{l\})$ • $v \prec O$ • $l \prec O$ • $\mathit{enoughObs}(v,\ \mathit{Wt} \uplus \{v\},\ \mathit{Ot})$	• $\mathit{obs}(O \uplus \{l\})$
When a thread t notifies a thread waiting for a condition variable v		• $\mathit{Wt} - \{v\}$
When a thread t notifies all threads waiting for a condition variable v		• $\mathit{Wt}(v) = 0$

is woken up and one instance of v is removed from *Wt*. Executing `v.notifyAll()` results in waking up all threads that are waiting for v (if any) and removing all instances of v from *Wt*.

An example program that can be successfully verified with this technique is shown in Listings 1-3. This program is developed by the author of this paper. For more examples, see [14].

The example program is a concurrent unbounded buffer with two producers and one consumer. The producers run in separate threads and each adds five items to the buffer. The main thread acts as the consumer and tries to read ten items from the buffer.

After creating a buffer in line 6 of Listing 3, a wait level is assigned to the lock and the condition variable. `buffer.l` references the intrinsic lock of the buffer, while `buffer.v` references its condition variable. `R` is a function that maps a lock/condition variable to its wait level.

After creating the two producers, ten obligations for `buffer.v` are loaded into the bag of obligations of the main thread (line 13), since it will try to read ten values from the buffer later in the program. As both `p1` and `p2` will write five values to the buffer, the obligations are divided over the two producers, each getting five obligations (lines 15-17). This leaves the main thread with an empty bag of obligations again (line 18).

After the producers have been instantiated, the main thread will continue by reading values from the buffer (lines 20-22). As the main thread has no obligations, the preconditions of `read()` are satisfied (see Listing 1). If the buffer is empty, $\mathit{enoughObs}(\mathit{buffer.v},\ \mathit{Wt} \uplus \{\mathit{buffer.v}\},\ \mathit{Ot})$ holds, since the producer threads have loaded the obligations for `buffer.v`. Otherwise, the first value from the buffer is returned (lines 12-18).

When a producer executes `buffer.write(i)`, `i` is added to the buffer (line 15 of Listing 2). After signalling a waiting thread (if one), the producer can discharge an obligation for `buffer.v` (lines 28-29 of Listing 1), since either no threads are waiting for `buffer.v` or one of the producers has an obligation for `buffer.v`, satisfying $\mathit{enoughObs}(\mathit{buffer.v},\ \mathit{Wt},\ \mathit{Ot})$. After a producer has written to the buffer five times, the thread has an empty bag of obligations (line 18 of Listing 2).

As can be seen in Listing 3, the program starts with an empty bag of obligations and also ends with an empty bag, proving the absence of deadlocks in this program. If the program would try to read eleven values instead of ten or if the `notify()` call was forgotten in `write(T item)`, the verification would fail and the program would not be deadlock-free.

```
 1 import java.util.LinkedList;
 2
 3 public class UnboundedBuffer<T> {
 4
 5   private final LinkedList<T> buffer = new
       LinkedList<>();
 6
 7   // req: obs(O) ∧ this.l ≺ O ∧ this.v ≺ O
 8   // ens: obs(O)
 9   public T read() {
10     synchronized (this) {
11       // obs(O ⊎ {this.l})
12       while (buffer.size() == 0) {
13         // enoughObs(this.v, Wt ⊎ {this.v}, Ot)
     * this.l ≺ O ∧ this.v ≺ O
14         try { wait(); }
15         catch (InterruptedException e) { }
16         // enoughObs(this.v, Wt, Ot)
17       }
18       return buffer.pop();
19     }
20     // obs(O)
21   }
22
23   // req: obs(O ⊎ {this.v}) ∧ this.l ≺ O ⊎ {this
       .v}
24   // ens: obs(O)
25   public void write(T item) {
26     synchronized (this) {
27       // obs(O ⊎ {this.v, this.l}) * Ot ⊎ {this.
     v}
28       buffer.add(item);
29       notify();
30       // discharge(this.v)
31       // obs(O ⊎ {this.l}) * Ot
32     }
33     // obs(O)
34   }
35 }
```

Listing 1. UnboundedBuffer.java

```
public class Producer extends Thread {

  private final UnboundedBuffer<Integer> buffer;

  public Producer(UnboundedBuffer<Integer> buffer) {
    this.buffer = buffer;
  }

  // req: obs(O ⊎ {5∗buffer.v})
  // ens: obs(O)
  @Override
  public void run() {
    for (int i = 0; i < 5; i++) {
      // obs(O ⊎ {(5−i)∗buffer.v}
      this.buffer.write(i);
      // obs(O ⊎ {5−i−1)∗buffer.v}
    }
    //obs(O)
  }
}
```

Listing 2. Producer.java

```
public class Main {

  // req: obs({})
  // ens: obs({})
  public static void main(String[] args) {
    UnboundedBuffer<Integer> buffer = new UnboundedBuffer<>();
    // R(buffer.l) = 0 ∧ R(buffer.v) = 1

    Producer p1 = new Producer(buffer);
    Producer p2 = new Producer(buffer);

    // obs({})
    // charge(buffer.v, 10)
    // obs({10 ∗ buffer.v})
    p1.start();
    // obs({5 ∗ buffer.v})
    p2.start();
    // obs({})

    for (int i = 0; i < 10; i++) {
      buffer.read();
    }
  }
}
```

Listing 3. Main.java

2.2 VerCors

VerCors is a verification tool developed by the University of Twente to verify parallel and concurrent programs [10]. Next to reasoning about race freedom and memory safety, it is also able to verify functional correctness of a program [5]. Verification of a program in VerCors is modular, meaning that each function in a program is verified both separately (i.e., independent of other functions) and together with other functions.

A wide range of frontend languages are supported, consisting of subsets of Java, C, PVL [26], OpenCL and OpenMP [5]. A program written in any of those languages can thus be verified by VerCors.

Programs can be verified by VerCors with the help of annotations. These annotations are combined with permission-based separation logic [3, 4]. Programmers can annotate their code using predefined keywords. In Java, those annotations should be written as comments and the comment should start with '@'.

The functionality provided by these annotations includes, but is not limited to, specifying invariants, pre- and postconditions of a method and permissions of a heap location. These assertions are then evaluated using symbolic execution.

An example of an annotated program is shown in Listing 4. The program is a simple counter. It features pre- and postconditions (`requires` and `ensures`), heap location permissions (`Perm`) and a loop invariant.

In the example program, the counter can be incremented by one (`increment`) or by any nonnegative number n (`incrementByN`). The precondition of the latter function ensures that only nonnegative numbers can be passed as arguments. If the method would be called with a negative number, the verification would fail.

```
public class Counter {

  public int number;

  /*@
    requires Perm(number, write);
    ensures  Perm(number, write) ** number == \old(number) + 1;
   */
  public void increment() {
    number = number + 1;
  }

  /*@
    requires Perm(number, write) ** n >= 0;
    ensures  Perm(number, write) ** number == \old(number) + n;
   */
  public void incrementByN(int n) {
    int i = n;
    //@ loop_invariant Perm(number, write) ** number + i == \old(number) + n ** i >= 0;
    while (i > 0) {
      number = number + 1;
      i = i - 1;
    }
  }
}
```

Listing 4. Counter.java

The architecture, as described in [5], can be found in Figure 1. Before a program written in one of the supported languages can be verified, a programmer has to annotate its code in a JML-like [18] syntax. Together with the program, these annotations are translated into a verification problem in Silver [20]. The verification is then done on this verification problem using the Viper framework [20], which in turn uses an SMT solver Z3 [9].

VerCors is primarily responsible for parsing the input programs and transforming those into a verification problem that Viper can reason about. Verification of the resulting Viper program is done using external libraries and is thus not part of the development of VerCors.

The first step in the verification process is parsing the source program in one of the supported languages. Both the language-specific statements and annotations are translated into the internal Abstract Syntax Tree (AST) named COL (Common Object Language), after which the COL is type checked.

The next step is applying several rewrites to the COL. During the rewrite phase, the COL is rewritten such that the resulting program has the same result as before the rewrite, but features more verification constructs. In addition, some rewrites are necessary to translate concepts that are not supported by Viper. Each rewrite is responsible for a certain change in the AST. This can be a large rewrite, such as rewriting classes to functions, or a smaller one, such as providing the current thread identifier to all functions.

Finally, the transformed COL is translated into a valid Viper program. The Viper framework is one of the sup-

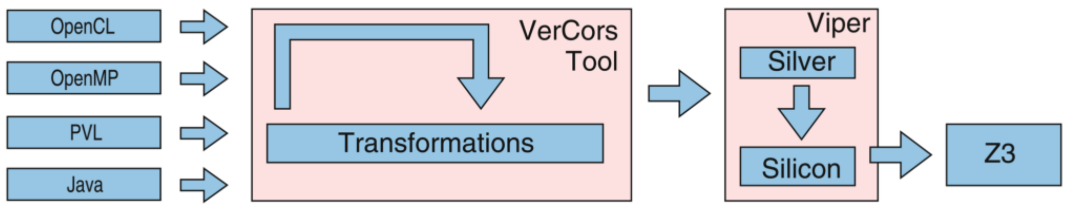

Figure 1. The architecture of VerCors [5]

ported backends and currently the main backend. The Viper framework is responsible of verifying the resulting Viper program. Based on this outcome, VerCors either marks the program as a pass or a fail.

3. ANNOTATION SYNTAX

This section proposes the additional annotation syntax that is required for the verification of absence of deadlocks in programs that use monitors. The annotations provide extra information to VerCors, such that VerCors can successfully verify deadlock-freeness of a program. These annotations are systematically shown in Table 2.

The annotations defined in this paper follow the guidelines of VerCors. This means that statement annotations don't use parentheses for separating the function name with the parameters, while expression annotations do use such parentheses. In addition, expressions start with a backslash.

To charge obligations for a monitor, `charge_obs` can be used. It requires two parameters, namely a monitor instance m and the amount of obligations to be charged n. Note that only the condition variable of a monitor is charged via this function. In the case of $n = 1$, `charge_ob` can be used instead, which takes only a single parameter, namely a monitor instance.

Obligations can be discharged using `discharge_obs`. Just like `charge_obs`, `discharge_obs` requires a monitor instance and the amount of obligations to be charged. In addition, only obligations for the condition variable of the monitor are discharged. Similarly, `discharge_ob` can be used if only one obligation for a monitor is to be discharged.

Transferring obligations to another thread is done via `transfer_obs`. Its parameters are a monitor instance m, the amount of obligations to be transferred n and the thread identifier t to which the obligations should be transferred. Only obligations for condition variables can be transferred between threads. Similar to the previous two statements, the simplified form `transfer_ob m, t` is available for $n = 1$.

To access Wt of a monitor m, `\Wt(m)` can be used. Similarly, `\Ot(m)` is defined to access Ot of m. These annotations are typically used in pre- and postconditions of a method.

Table 2. Annotation syntax

Annotation	Type	Description	Example

`charge_obs m, n`
if n = 1: `charge_ob m`
	Statement	Charge n obligations for the condition variable of a monitor m.	Object o = **new** Object();
			//@ charge_obs o, 3;

`discharge_obs m, n`
if n = 1: `discharge_ob m`
	Statement	Discharge n obligations for the condition variable of a monitor m.	Object o = **new** Object();
			//@ discharge_obs o, 3;

`transfer_obs m, n, t`
if n = 1: `transfer_ob m, t`
	Statement	Transfer n obligations for the condition variable of a monitor m to a thread with identifier t.	Object o = **new** Object();
			//@ charge_obs o, 5;
			Thread th = **new** Thread();
			//@ transfer_obs o, 3, th.getId();

`set_wait_level z, r`	Statement	Set the wait level r of a lock / condition variable z.	Object o = **new** Object();
			//@ set_wait_level \lock(o), 0;
			//@ set_wait_level \cond(o), 1;

`\Wt(m)`	Expression	Access Wt of a monitor m.	\Wt(**this**)
`\Ot(m)`	Expression	Access Ot of a monitor m.	\Ot(**this**)
`\lock(m)`	Expression	Access the lock of a monitor m.	\lock(**this**)
`\cond(m)`	Expression	Access the condition variable of a monitor m.	\cond(**this**)
`\wait_level(z)`	Expression	Access the wait level of a lock / condition variable z.	\wait_level(\cond(**this**))

`\has_obs(z, n)`
if n = 1: `\has_ob(z)`
	Expression	Check that the current thread has at least n obligations for a lock / condition variable z.	\has_obs(\cond(**this**), 3)

`\no_obs`	Expression	Check that the current thread has no obligations.	\no_obs

Since a monitor in Java combines both a lock and a condition variable into a single object, annotations are needed to differentiate between the two. To this end, `\lock(`*m*`)` accesses the lock of a monitor m and `\cond(`*m*`)` gives access to the associated condition variable.

To set the wait level of a lock or condition variable, the `set_wait_level` annotation has been defined. It is a statement that sets the wait level of a lock or condition variable z to an integer r. The parameter z must either be a lock (e.g. $z ==$ `\lock(`*m*`)`) or a condition variable (e.g. $z ==$ `\cond(`*m*`)`). Querying the wait level can be done using `\wait_level(`*z*`)`.

To check if the current thread has a certain number of obligations for a lock or condition variable, `\has_obs` can be used. The function has two parameters: a lock or condition variable z and an amount n. The function returns `true` if the current thread has at least n obligations for z and `false` otherwise. If $n = 1$, an alternative function `\has_ob(`*z*`)` is available, only requiring a lock/condition variable z.

Finally, to indicate that the current thread has an empty bag of obligations, `\no_obs` is defined. This annotation is typically used as both precondition and postcondition of the main method of a program, as a program should start with an empty bag of obligations and also end with an empty bag.

4. IMPLEMENTATION

This section describes how the verification of deadlock-free monitors can be implemented in VerCors. Section 4.1 discusses the the general steps that are taken. Section 4.2 focuses on the utilities that should be defined for the rewrite phase. Finally, section 4.3 explains how each annotation and monitor method is rewritten.

4.1 Overview

The general approach is to transform the input program with the proposed annotations to expressions and statements that are already supported in VerCors. Each class of the input program is transformed into an extended class with ghost fields and specifications over these fields.

The implementation consists mainly of three stages:

1. Parsing the new annotations
2. Type checking the annotations
3. Rewrite the annotations and other constructs to supported VerCors input

First, the new annotations can be parsed by adding them to the specification parser. These should then be mapped to appropriate keywords.

Second is the type checking phase. In this stage, the arguments of the new annotations should be type checked. This is necessary, since the parser may allow invalid types to be passed as arguments to the annotations. The allowed types can be derived from Section 3 and Table 2. For example, an obligation can only be charged for a class instance, not for an integer. Similarly, the argument of `wait_level` expects either the lock or condition variable of a monitor.

Finally, the last step consists of adding a new rewrite pass, which walks through the Abstract Syntax Tree and rewrites the annotations defined in Section 3 and the monitor methods into expressions and statements supported by VerCors. This rewrite pass is were the technique from [14] is applied. A schematic overview of the required rewrites can be found in Appendix A of the Online Appendix [24]. The remainder of this section discusses the implementation of the rewrite pass.

4.2 Preliminaries

4.2.1 Variables per monitor

Each class should be given four new ghost fields. These are `Wt`, `Ot`, `wait_level_lock` and `wait_level_cond`, all of type integer. The former two correspond to *Wt(v)* and *Ot(v)* and should have 0 as initial value and should always be nonnegative. The latter two fields are the wait levels of respectively the intrinsic lock and the condition variable. These need to be manually set by `set_wait_level` and thus do not have an initial value.

4.2.2 Obligations per thread

Next to the variables per monitor, each thread should have its own bag of obligations `obs`. To keep track of a thread's obligations, this bag is injected into every function and is thus always available for a thread. This can be achieved by adding the obligations argument to all methods and method invocations.

4.3 Rewriting annotations and method calls

4.3.1 Expressions

Accessing *Wt* and *Ot* of a monitor m is done by using the expressions `\Wt(m)` and `\Ot(m)` respectively. These expressions should be translated into `m.Wt` and `m.Ot`.

The intrinsic lock and condition variable of a monitor m can be accessed with `\lock(m)` and `\cond(m)`. There are two approaches that can be taken with these expressions. The first approach is ignoring these expressions and use them when rewriting the expression or statement in which they occur. Alternatively, two fields `lock` and `cond` can be added to each monitor. The expressions can then be rewritten to `m.lock` and `m.cond` for a monitor m. This paper uses the first approach.

The `\wait_level` annotation queries the wait level of a lock or condition variable. Depending on whether the argument is a lock or condition variable, the annotation is rewritten to `m.wait_level_lock` or `m.wait_level_cond` for a monitor m.

To check whether the current thread has certain obligations, `\has_ob` or `\has_obs` is used. This expression should be replaced by an expression that checks whether the obligations are present in `obs`.

Finally, `\no_obs` can be rewritten to `obs.length == 0`.

4.3.2 Charging and discharging obligations

To charge one or more obligations for a lock or condition variable, a programmer has to use the `charge_ob` or `charge_obs` annotation. Upon encountering one of these annotations in the rewrite phase, the annotation statement should be replaced by two ghost statements. In the first statement, the obligations are added to `obs`. The second statement adds the amount of obligations to the field `Ot` of the monitor.

Discharging obligations (using `discharge_ob` or `discharge_obs`) has a similar procedure as charging obligations, except that an assertion is needed to check the precondition (see Table 1). The annotation should thus be replaced by three statements. The first statement should assert that the precondition *enoughObs*(v, *Wt*, *Ot*$-\{n{*}v\}$) holds. Since each monitor has been given the fields `Wt` and `Ot`, this is equal to asserting `m.Wt == 0 || m.Ot - n > 0` for a monitor m and the amount of obligations n. The next two statements remove the obligation(s) from `obs` and subtract the amount of obligations from the `Ot` field of the monitor.

4.3.3 Transferring obligations

As discussed in Section 3, obligations can be transferred from one thread to another with `transfer_ob` or `transfer_obs`. Replacing these annotations is done in three parts.

First, an assert statement is necessary to assert that the current thread has the obligations that should be transferred. This avoids negative obligations for the current thread or that obligations are implicitly created.

Secondly, the obligations should be copied to the local bag of obligations of the thread with the indicated thread identifier.

Finally, the obligations should be removed from `obs` of the current thread.

4.3.4 Setting the wait level

The wait level of a lock or condition variable can be set with `set_wait_level`. This annotation can be rewritten to an assignment statement, either assigning the wait level to `m.wait_level_lock` (in case of a lock) or `m.wait_level_cond` (in case of a condition variable), where m is a monitor.

4.3.5 Acquiring and releasing an intrinsic lock

When acquiring the intrinsic lock of a monitor (by using the `synchronized` keyword), there are three required steps.

First, the precondition of acquiring a lock needs to be checked. As shown in Table 1, the lock should have the lowest wait level of the current thread's obligations. This can be checked by comparing each obligation's wait level in `obs` with the wait level of the lock. Note that it is allowed to have multiple obligations for the same lock, since intrinsic locks are reentrant. Therefore, this may require an additional comparison.

Secondly, after the lock is acquired, the obligation for the lock should be added to `obs`.

Finally, having acquired the lock means that the current thread obtains full permissions to read from and write to the fields `Wt` and `Ot` of the monitor. These permissions should be given to the current thread.

When releasing the intrinsic lock of a monitor, one obligation for that lock should be removed from `obs`. In addition, the permissions for `Wt` and `Ot` should be revoked.

4.3.6 Waiting for a monitor

Upon encountering a `wait()` statement, there are five actions that should be taken.

First, a thread can only execute `m.wait()` for a monitor m if it has acquired the intrinsic lock of m. There should thus be at least one obligation for the lock of m in `obs`. This should be asserted.

Secondly, a thread may only wait for a condition variable v if v has the lowest wait level of all wait levels of the obligations of the thread. An assertion is thus needed that compares the wait levels of the obligations in `obs` and asserts that the wait level of v is the lowest.

Thirdly, similar to the previous action, it should be asserted that the intrinsic lock of the monitor has the lowest wait level among the wait levels of `obs`.

Fourthly, as `m.Wt` is going to be incremented, an assertion is required that asserts that `m.Ot > 0` (which is equal to $\mathit{enoughObs}(v,\ \mathit{Wt} \uplus \{v\},\ \mathit{Ot})$, since `m.Wt + 1 > 0`) holds.

Finally, `m.Wt` should be incremented.

4.3.7 Notifying threads

Notifying one thread should result in decrementing the `Wt` field of the monitor m. Note that `m.Wt` should always be nonnegative, so `m.Wt` should only be decremented if it is greater than zero. Otherwise, `m.Wt` stays zero.

Notifying all threads results in assigning 0 to `m.Wt`.

5. VALIDATION

To validate the proposed implementation, part of it has been implemented in VerCors as a proof of concept. All new annotations have been added to the parser and the type checker, but only the functionality of the annotations and monitor methods that involve *Wt* or *Ot* has been implemented, as well as setting and querying the wait level of a lock or condition variable. The implementation can be found on the associated GitHub repository [22].

All annotated examples in this paper have been successfully verified by the implementation in VerCors. Note that it does not support all functionality, in particular annotations and methods that involve the local bag of obligations. The reason for this is a time constraint. The current implementation is thus not able to fully verify absence of deadlock, but nonetheless shows the feasibility of the proposed implementation.

6. EXAMPLE PROGRAM

To illustrate how a program should be annotated, this section shows an example program. More examples can be found on [23].

The example program in Listings 5-7 is an implementation of a barrier. A barrier is a construct that is often used to synchronize threads. A thread that enters a barrier waits until a predefined amount of threads has entered the barrier. If the last thread enters the barrier, all threads are notified and resume their executions.

The program starts with the `main()` method in line 4 of Listing 7. The main thread starts with an empty bag of obligations and should also end with an empty bag, indicated by `context \no_obs` in line 3. After a barrier is created in line 5, the wait levels of the condition variable and the intrinsic lock of `barrier` are set (lines 6-7). Also, three obligations for `barrier` are charged (line 8).

Each `BarrierThread` then takes over one of those obligations (by using `transfer_ob`) and is started (lines 10-18). At the end of `main`, the main thread thus has an empty bag of obligations. All threads are concurrently executing the first for-loop and will then enter the barrier (lines 29-30) of Listing 6. Next to the correct permissions, the `waitForBarrier()` method (line 28 of Listing 5) requires that the calling thread has an obligation for the condition variable and specifies the allowed values for `n` and `\Ot(this)`.

Since `waitForBarrier()` will discharge one obligation, each thread will end with an empty bag of obligations. As there are no obligations in the system anymore, this program is verified successfully.

```
1 public class Barrier {
2
3   private int n;
4
5   /*@
6     requires n > 0;
7     ensures Perm(this.n, read);
```

```
    ensures this.n == n;
  */
  public Barrier(int n) {
    this.n = n;
  }

  /*@
    context Perm(n, read);
    context Perm(\Ot(this), read);
    context Perm(\wait_level(\lock(this)), read)
    ;
    context Perm(\wait_level(\cond(this)), read)
    ;
    context \wait_level(\lock(this)) == 0;
    context \wait_level(\cond(this)) == 1;
    requires n > 0;
    requires \Ot(this) > 0;
    requires n <= \Ot(this);
    requires \has_ob(\cond(this));
    ensures !\has_ob(\cond(this));
    ensures n == \old(n) - 1;
  */
  public synchronized void waitForBarrier() {
    n--;
    if (n == 0) {
      notifyAll();
      //@ discharge_ob this;
    } else {
      //@ discharge_ob this;
      wait();
    }
  }
}
```

Listing 5. Barrier.java

```
public class BarrierThread {

  private final Barrier barrier;

  /*@
    ensures Perm(this.barrier, read);
    ensures this.barrier == barrier;
  */
  public BarrierThread(Barrier barrier) {
    this.barrier = barrier;
  }

  /*@
    context Perm(barrier, read);
    context Perm(\Ot(barrier), read);
    context Perm(\wait_level(\lock(barrier)),
    read);
    context Perm(\wait_level(\cond(barrier)),
    read);
    context Perm(barrier.n, read);
    context \wait_level(\lock(barrier)) == 0;
    context \wait_level(\cond(barrier)) == 1;
    requires barrier.n > 0;
    requires \Ot(barrier) > 0;
    requires barrier.n <= \Ot(barrier);
    requires \has_ob(\cond(barrier));
    ensures \no_obs;
    ensures barrier.n == \old(barrier.n) - 1;
  */
  public void start() {
    for (int i = 0; i < 10; i++) { }
    barrier.waitForBarrier();
    for (int j = 10; j < 20; j++) { }
  }

  public void join();

  //@ ensures \result == \current_thread;
  public int getId();
}
```

Listing 6. BarrierThread.java

```
public class Main {

  //@ context \no_obs;
  public void main() {
    Barrier barrier = new Barrier(3);
    //@ set_wait_level \lock(barrier), 0;
    //@ set_wait_level \cond(barrier), 1;
    //@ charge_obs barrier, 3;

    BarrierThread t1 = new BarrierThread(barrier
    );
    //@ transfer_ob barrier, t1.getId();
    BarrierThread t2 = new BarrierThread(barrier
    );
    //@ transfer_ob barrier, t2.getId();
    BarrierThread t3 = new BarrierThread(barrier
    );
    //@ transfer_ob barrier, t3.getId();
    t1.start();
    t2.start();
    t3.start();

    t1.join();
    t2.join();
    t3.join();
  }
}
```

Listing 7. Main.java

7. CONCLUSION

This paper showed how the technique proposed in [14] can be incorparated in VerCors to verify deadlock-freeness of Java-like programs with monitors. With a proof of concept implementation, the feasibility of the technique is shown. For the implementation, eleven new annotations were defined, consisting of both expressions and statements. The application of those annotations were shown in an example program.

8. DISCUSSION AND FUTURE WORK

Based on the results of this paper, several directions for future work can be identified.

This research only focused on monitors that have exactly one lock and one condition variable, since this is the default monitor in Java. However, Java also has support for condition interfaces, which allows a lock to have multiple condition variables associated with it. Future work would consist of extending the implementation proposed in this paper to support multiple condition variables per lock.

Although the technique introduced in [14] describes verification of condition variables in general, this paper only focused on the application of monitors. For example, Hamin et al. [14] describe a relaxation of their technique, such that a wider range of applications of condition variables can be verified, such as bounded channels. Future work consists of applying this relaxation to the implementation described in this paper.

Finally, the proposed implementation requires the programmer to annotate their code manually. In addition, some insight in the technique is necessary to correctly annotate a program. For example, a precondition or lock invariant using *Wt* or *Ot* might be required, which requires knowledge of the underlying theory. Future work can be aimed at reducing the amount of manual annotations and simplifying the verification for programmers.

9. REFERENCES

[1] A. O. Abd El-Gwad, A. I. Saleh, and M. M. Abd-ElRazik. A novel scheduling strategy for an efficient deadlock detection. In *Proceedings - The 2009 International Conference on Computer Engineering and Systems, ICCES'09*, pages 579–583, 2009.

[2] E. Aldakheel. Deadlock Detector and Solver (DDS). In M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, editors, *Proceedings of the 40th*

International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pages 512–514. ACM, 2018.

[3] A. Amighi, S. Blom, S. Darabi, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Verification of Concurrent Systems with VerCors. In M. Bernardo, F. Damiani, R. Hähnle, E. B. Johnsen, and I. Schaefer, editors, *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, volume 8483 of *Lecture Notes in Computer Science*, pages 172–216. Springer, 2014.

[4] A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-Based Separation Logic for Multithreaded Java Programs. *Logical Methods in Computer Science*, 11(1), feb 2015.

[5] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In N. Polikarpova and S. Schneider, editors, *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings*, volume 10510 of *Lecture Notes in Computer Science*, pages 102–110. Springer, 2017.

[6] S. Blom and M. Huisman. The VerCors Tool for Verification of Concurrent Programs. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 127–131. Springer, 2014.

[7] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[8] P. de Carvalho Gomes, D. Gurov, and M. Huisman. Specification and Verification of Synchronization with Condition Variables. In C. Artho and P. C. Ölveczky, editors, *Formal Techniques for Safety-Critical Systems - 5th International Workshop, FTSCS 2016, Tokyo, Japan, November 14, 2016, Revised Selected Papers*, volume 694 of *Communications in Computer and Information Science*, pages 3–19, 2016.

[9] L. De Moura and N. Bjørner. Z3: An efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems,14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedi*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, Berlin, Heidelberg, 2008.

[10] FMT - University of Twente. The VerCors Verifier. https://vercors.ewi.utwente.nl.

[11] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type and effect system for deadlock avoidance in low-level languages. In S. Weirich and D. Dreyer, editors, *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*, pages 15–28. ACM, 2011.

[12] B. Goetz. *Java Concurrency in Practice.* Addison-Wesley Professional, 1st edition, 2006.

[13] C. S. Gordon, M. D. Ernst, and D. Grossman. Static lock capabilities for deadlock freedom. In B. C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 67–78. ACM, 2012.

[14] J. Hamin and B. Jacobs. Deadlock-Free Monitors. In A. Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 415–441. Springer, 2018.

[15] C. A. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[16] B. Jacobs and F. Piessens. The VeriFast program verifier. *CW Reports*, 2008.

[17] C. Laneve. A Lightweight Deadlock Analysis for Programs with Threads and Reentrant Locks. In K. Havelund, J. Peleska, B. Roscoe, and E. P. de Vink, editors, *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, volume 10951 of *Lecture Notes in Computer Science*, pages 608–624. Springer, 2018.

[18] G. T. Leavens and Y. Cheon. Design by Contract with JML. 2006.

[19] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-Free Channels and Locks. In A. D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 2010.

[20] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

[21] P. O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, jan 2019.

[22] M. J. Roelink. GitHub - Matthiti/vercors. https://github.com/Matthiti/vercors.

[23] M. J. Roelink. Github - Matthiti/vercors-monitor-examples. https://github.com/Matthiti/vercors-monitor-examples.

[24] M. J. Roelink. Online Appendix. https://roelink.eu/files/encoding-deadlock-free-monitors-in-vercors-online-appendix.pdf.

[25] F. Sorrentino. PickLock: A Deadlock Prediction Approach under Nested Locking. In B. Fischer and J. Geldenhuys, editors, *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*, volume 9232 of *Lecture Notes in Computer Science*, pages 179–199. Springer, 2015.

[26] Utwente-fmt. Prototypal Verification Language. https://github.com/utwente-fmt/vercors/wiki/Prototypal-Verification-Language.