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.

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 \( \text{enoughObs}(v, Wt, Ot) \) defined as

\[
\text{enoughObs}(v, Wt, Ot) = (Wt(v) > 0 \implies Ot(v) > 0)
\]

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 \( \text{wait level} \). 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 \( o < obs(O) \), 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 \( \text{wait}() \) 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 \( \text{wait}() \) 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 \( t \) \( l < obs(O) \). Note that this relation may be relaxed to \( l \leq obs(O) \) 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 \( obs(O) \).

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., \( obs(O) \)).

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

A thread \( t \) can discharge \( n \) obligations for a condition variable \( v \) arbitrarily if after the discharge, the rule \( \text{enoughObs}(v, Wt, Ot) \) holds. Therefore, before discharging the obligations, \( \text{enoughObs}(v, Wt, Ot) \setminus \{n \ast v\} \) 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 \( t \) \( obs(O \setminus \{n \ast v\}) \) as well as from the global bag of obligations \( obs(O) \).

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 \( t \) \( obs(O) \).

In addition, the wait levels of \( v \) and \( l \) should be lower than the wait levels of all obligations of \( t \) \( v < obs(O) \) and \( l < obs(O) \). \( l < obs(O) \) 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 \( \text{enoughObs}(v, Wt \cup \{v\}, Ot) \) should hold. After calling \( v \).wait() and before suspension, one instance of \( v \) is added to \( Wt \).

When a thread executes \( \text{notify}() \) on a condition variable \( v \), an arbitrary thread that is waiting for \( v \) (if there is one)
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. After creating a buffer in line 6 of Listing 3, a wait level is acquired for a condition variable that is protected by a lock. This technique is shown in Listings 1-3. This program is successfully verified with the verification would fail and the program would not be deadlock-free.

### Table 1. Rules for deadlock-free monitors

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>When a thread t acquires a lock l</td>
<td>l ≺ obs(O)</td>
<td>obs(O ⊔ {l})</td>
</tr>
<tr>
<td>When a thread t releases a lock l</td>
<td></td>
<td>obs(O − {l})</td>
</tr>
<tr>
<td>When a thread t loads n obligations for a condition variable v</td>
<td></td>
<td>obs(O ⊔ {n * v})</td>
</tr>
<tr>
<td>When a thread t discharges n obligations for a condition variable v</td>
<td>enoughObs(v, Wt, Ot − {n * v})</td>
<td>obs(O − {n * v})</td>
</tr>
<tr>
<td>When a thread t waits for a condition variable v that is protected by a lock l</td>
<td>obs(O ⊔ {l})</td>
<td>obs(O ⊔ {l})</td>
</tr>
<tr>
<td>When a thread t notifies a thread waiting for a condition variable v</td>
<td></td>
<td>Wt − {v}</td>
</tr>
<tr>
<td>When a thread t notifies all threads waiting for a condition variable v</td>
<td></td>
<td>Wt(v) = 0</td>
</tr>
</tbody>
</table>

import java.util.LinkedList;

private final UnboundedBuffer<T> {  
    // obs(O) ∧ this.l ≺ O ∧ this.v ≺ O
    // ens: obs(O)
    public T read() {  
        synchronized (this) {  
            // obs(O ⊔ {this.l})
            while (buffer.size() == 0) {  
                // enoughObs(this.v, Wt ⊔ {this.v}, Ot)
                // this.l ≺ O ∧ this.v ≺ O
                try { wait(); }
                catch (InterruptedException e) { }  
                // enoughObs(this.v, Wt, Ot)
            }
            return buffer.pop();
        }  
        // obs(O)
    }
    // req: obs(O) ∧ this.l ≺ O ⊔ this.v
    // ens: obs(O)
    public void write(T item) {  
        synchronized (this) {  
            // obs(O ⊔ {this.v, this.l}) ⊔ Ot ≺ this.v
            // buffer.add(item);
            notify();
            // discharge(this.v)
            // obs(O ⊔ {this.l}) ⊔ Ot
        }  
        // obs(O)
    }
}

Listing 1. UnboundedBuffer.java
invariant Perm(number, write)

for i = 0; i < 5; i++ {
  obs(O \{ (5 - i) * buffer.v \})
  this.buffer.write(i);
  obs(O \{ (5 - i - 1) * buffer.v \})
}
obs(O)

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++) {
      int n = number + i;
      // req: Perm(number, write)
      // ens: Perm(number, write)
      this.buffer.write(i);
      // obs(O \{ (5 - i) * buffer.v \})
    }
    obs(O)
  }
}

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

public class Counter {
  public int number;
  @Override
  public void increment() {
    int i = n;
    // req: Perm(number, write)
    // ens: Perm(number, write)
    number = number + 1;
    old(number) + n
    obs(O \{ (5 - i) * buffer.v \})
    this.buffer.write(i);
    obs(O \{ (5 - i - 1) * buffer.v \})
  }
}

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.

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

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>charge_obs m, n</td>
<td>Statement</td>
<td>Charge n obligations for the condition variable of a monitor m.</td>
<td>Object o = new Object(); @ charge_obs o, 3.</td>
</tr>
<tr>
<td>discharge_obs m, n</td>
<td>Statement</td>
<td>Discharge n obligations for the condition variable of a monitor m.</td>
<td>Object o = new Object(); @ discharge_obs o, 3.</td>
</tr>
<tr>
<td>transfer_obs m, n, t</td>
<td>Statement</td>
<td>Transfer n obligations for the condition variable of a monitor m to a thread with identifier t.</td>
<td>Object o = new Object(); @ transfer_obs o, 3, t.getId();</td>
</tr>
<tr>
<td>set_wait_level z, r</td>
<td>Statement</td>
<td>Set the wait level r of a lock / condition variable z.</td>
<td>Object o = new Object(); @ set_wait_level (\nabla \text{lock}(o)), 0; @ set_wait_level (\nabla \text{cond}(o)), 1;</td>
</tr>
<tr>
<td>(\nabla Wt(m))</td>
<td>Expression</td>
<td>Access Wt of a monitor m.</td>
<td>(\nabla Wt(\text{this}))</td>
</tr>
<tr>
<td>(\nabla O(m))</td>
<td>Expression</td>
<td>Access Ot of a monitor m.</td>
<td>(\nabla O(\text{this}))</td>
</tr>
<tr>
<td>(\nabla \text{lock}(m))</td>
<td>Expression</td>
<td>Access the lock of a monitor m.</td>
<td>(\nabla \text{lock}(\text{this}))</td>
</tr>
<tr>
<td>(\nabla \text{cond}(m))</td>
<td>Expression</td>
<td>Access the condition variable of a monitor m.</td>
<td>(\nabla \text{cond}(\text{this}))</td>
</tr>
<tr>
<td>(\nabla \text{wait}_\text{level}(z))</td>
<td>Expression</td>
<td>Access the wait level of a lock / condition variable z.</td>
<td>(\nabla \text{wait}_\text{level}(\nabla \text{cond}(\text{this})))</td>
</tr>
<tr>
<td>(\nabla \text{has}_\text{obs}(z, n))</td>
<td>Expression</td>
<td>Check that the current thread has at least n obligations for a lock / condition variable z.</td>
<td>(\nabla \text{has}_\text{obs}(\nabla \text{cond}(\text{this}), 3))</td>
</tr>
<tr>
<td>(\nabla \text{no}_\text{obs})</td>
<td>Expression</td>
<td>Check that the current thread has no obligations.</td>
<td>(\nabla \text{no}_\text{obs})</td>
</tr>
</tbody>
</table>
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, $\text{\textbackslash lock}(m)$ accesses the lock of a monitor $m$ and $\text{\textbackslash cond}(m)$ gives access to the associated condition variable.

To set the wait level of a lock or condition variable, the $\text{set\_wait\_level}$ 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. $z = \text{\textbackslash lock}(m)$) or a condition variable (e.g. $z = \text{\textbackslash cond}(m)$). Querying the wait level can be done using $\text{\textbackslash wait\_level}(z)$.

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

Finally, to indicate that the current thread has an empty bag of obligations, $\text{\textbackslash no\_obs}$ 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 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 $\text{\textbackslash wait\_level}$ 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 where 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 $Wt, Ot, \text{wait\_level\_lock}$ and $\text{wait\_level\_cond}$, 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 $\text{set\_wait\_level}$ 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 $\text{obs}$. 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 $\text{\textbackslash Wt}(m)$ and $\text{\textbackslash Ot}(m)$ 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 $\text{\textbackslash lock}(m)$ and $\text{\textbackslash cond}(m)$. 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 $\text{lock}$ and $\text{cond}$ can be added to each monitor. The expressions can then be rewritten to $m.\text{lock}$ and $m.\text{cond}$ for a monitor $m$. This paper uses the first approach.

The $\text{\textbackslash wait\_level}$ 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 $m.\text{wait\_level\_lock}$ or $m.\text{wait\_level\_cond}$ for a monitor $m$.

To check whether the current thread has certain obligations, $\text{\textbackslash has\_obs}$ or $\text{\textbackslash has\_obs}$ is used. This expression should be replaced by an expression that checks whether the obligations are present in $\text{obs}$.

Finally, $\text{\textbackslash no\_obs}$ can be rewritten to $\text{obs}\_\text{length} == 0$.

#### 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 $\text{\textbackslash charge\_ob}$ or $\text{\textbackslash charge\_obs}$ 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 $\text{obs}$. The second statement adds the amount of obligations to the field $Ot$ of the monitor.

Discharging obligations (using $\text{\textbackslash discharge\_ob}$ or $\text{\textbackslash discharge\_obs}$) 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 $\text{enough\_Obs}(v, Wt, Ot - \{n\_v\})$ holds. Since each monitor has been given the fields $Wt$ and $Ot$, this is equal to asserting $m.Wt == 0 \lor m.Ot - n > 0$ 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 enoughObs(v, Wt ∪ {v}, 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 Wt(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.

```
public class Barrier {
    private int n;
    /**
     * requires n > 0;
     * ensures Perm(this, n, read);
     */

```
8 ensures this.n == n;
9 /
10 public Barrier(int n) {
11 this.n = n;
12 }
13
14 //@ context Perm(n, read);
15 context Perm(Ot(this), read);
16 context Perm(wait_level(cond(this)), read);
17 context Perm(has_ob(cond(this)), read);
18 context wait_level(lock(this)) == 0;
19 context wait_level(lock(this)) == 1;
20 requires n > 0;
21 requires Ot(this) > 0;
22 requires n <= Ot(this);
23 requires has_ob(cond(this));
24 ensures n == old(n) - 1;
25 ensures this.barrier == barrier;
26 public synchronized void waitForBarrier() {
27 n--;
28 if (n == 0) {
29 //@ discharge_ob this;
30 } else {
31 //@ discharge_ob this;
32 wait();
33 }
34 }
35
36 listing 5. Barrier.java

14 private final Barrier barrier;
15 //@ context Perm(barrier, read);
16 context Perm(Ot(barrier), read);
17 context Perm(wait_level(lock(barrier)), read);
18 context Permit(barrier.n, read);
19 context wait_level(lock(barrier)) == 0;
20 context wait_level(lock(barrier)) == 1;
21 requires barrier.n > 0;
22 requires Ot(barrier) > 0;
23 requires barrier.n <= Ot(barrier);
24 requires has_ob(cond(barrier));
25 ensures no_ob;
26 ensures barrier.n == old(barrier.n) - 1;
27 public void start() {
28 for (int i = 0; i < 10; i++) {
29 barrier.waitForBarrier();
30 }
31 }
32 }
33 }
34 //@ ensures result == current_thread;
35 public int getId();
36
37 listing 6. BarrierThread.java

Listing 7. Main.java

7. CONCLUSION

This paper showed how the technique proposed in [14] can be incorporated 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


