Use of Formal Method in Parallel Programming

By
Myo Min Thein
CSC 511
Use of Formal Method in Parallel Programming

Introduction

Over the past few years, software systems inevitable grew in scale and functionality. Because of this increase in complexity, those systems have higher chance of subtle errors and even some of these errors can cause loss of time, energy and even human life. As an approach which could help to create more reliable software, formal methods attempt to provide a specification language which has a firm mathematical semantics and a development notion which has a clear concept of what needs to be proved for a design to satisfy its specification. In the past, the use of formal method in practice seems desperate. The notations were too obscure, technique did not scale and techniques supported were too hard to use. Only too few people were trained to use effectively on the job. But nowadays we began to see the encouraging pictures of formal method. For software specification industry is trying to open out the notations such as Z to document the properties of the systems. In many areas of hardware and software, researchers and practitioners are doing more and more industrialized case studies to gain more benefits of formal method. Use of formal method does not guarantee genuine correctness. However, they provide our understanding of systems by revealing inconsistencies, ambiguities, and incompleteness that we might overlook. Among the advantages of formal methods in programming are its simple declarative and procedural semantics and high expressive power. Many have been claiming that use of formal methods can increase the reliability of software. A lot of case studies have found the use of formal methods to be quite effective.

Even a simple multi-process program can have very complicated behavior when it is executed and it is not possible to try to verify its correctness by exhaustive testing also.
The only way to prove the absence of errors in multi-process program is with a rigorous proof of its correctness. For parallel programming, formal method allows specification to be precisely stated and the conformance of an implementation to be verified by using mathematical and logic techniques.

Specification is the process of describing a system and its desired properties. Specifying parallel programs is much harder than sequential programs. Verification comes after specification. Verification is used to analyze the systems have the desired properties. Verification can be broken down into model checking and theorem proving. Model checking is a technique that relies on building a finite model of system and checking the desired property hold in that model. Theorem proving is technique by which both the system and its desired properties are expressed as formulas in some mathematical logic. Generally, if want to prove the programs correct, we have to start from the correct specification followed by correct verification. Proving parallel programs is much more complex than the sequential programs because of its additional properties that I will discuss in next sections.
Definition and Use of the Language

Parallel programming is different and hard to prove than sequential programming. The languages we study in Gries book [3] are not enough to prove the correctness of parallel program. However, Gries [4] and Owicki [1] introduce some notations that are useful in proving parallel programs. I will discuss to some extent level of those notations to make my paper easy to follow. Owicki [1] introduces three statements for parallel processing. The **cobegin** statement indicate that processes are to be executed in parallel; the **await** and **with** statement provides synchronization and mutual exclusion.

The **await** has the form

```plaintext
await B then S
```

where $B$ is a Boolean expression and $S$ is a statement. Execution of the process is delayed at the **await** until $B$ is true. At this time, $S$ is executed as an indivisible operation – no other process maybe execute while $S$ is executing or during $B$ is found to be true and execution of $S$ is begun since this might falsify $B$. Booleans $B$ come true at the same time, one of them is further delayed while the other executes. The scheduling algorithm for determining which process is allowed to proceed does not concern here. For simplicity, she assume that **awaits** cannot be nested. The formal definition of the **await** statement is:

```plaintext
\{Q\} await B then S \{R\}
```

In extension to the **await** statement, Owicki [4] introduce another add-on. For those parallel program with critical resource section (especially for mutual exclusion), it provides synchronization and protection of shared variables. The statement is in the form:
with \texttt{r when B do S}

has the following interpretation: \texttt{r} is a resource, \texttt{B} is a Boolean expression, and \texttt{S} is a statement which use the variables of \texttt{r}. When a process attempts to execute such a statement it is delayed until the condition \texttt{B} is true and \texttt{r} is not being used by another process. When the process has control of \texttt{r} and \texttt{B} is true, \texttt{S} is executed. Upon termination \texttt{r} is free for further use by other process. When several processes are competing for a particular resource, assumption is based on the scheduling about the order in which they receive. Critical section is the typical need of parallel program and critical section for the same resource cannot be nested. Much of the complexity for the parallel program arise from the interference of processes on the common variables. The critical section statement reduces these problems by guaranteeing that only one process at a time can access to the variables in a resource.

Before introducing the \texttt{cobegin} statement, here is what it means for two parallel process to be \emph{interference-free}.

\textit{Definition:} Given \{\texttt{Q}\} \texttt{S} \{\texttt{R}\}, let \texttt{T} be any assignment or \texttt{await} (not in \texttt{S}) with precondition pre(\texttt{T}). Say \texttt{T} does not interfere with the proof of \{\texttt{Q}\} \texttt{S} \{\texttt{R}\} if (a) \{\texttt{Q} \land \text{pre}(\texttt{T})\} \texttt{T} \{\texttt{R}\} (b) for each statement \texttt{S$'$} of \texttt{S} which is not within an \texttt{await}, \{\text{pre(S$'$)} \land \text{pre(T)}\} \texttt{T} \{\text{pre(S$'$)}\}.

Thus execution of \texttt{T} cannot affect the truth of the preconditions and result conditions used in the proof of \texttt{S}, and hence the proof \{\texttt{Q}\} \texttt{S} \{\texttt{R}\} holds even if \texttt{T} is executed while \texttt{S} is executing.
Definition: \{Q_1\} S_1 \{R_1\} and \{Q_2\} S_2 \{R_2\} are interference-free if each assignment statement of S_2 (which does not occur within an `await`) and each `await` of S_2 does not interfere with the proof of \{Q_1\} S_1 \{R_1\}, and vice versa.

If S_1 and S_2 are interference-free as just defined, then execution of S_2 leaves valid all the arguments used in the proof \{Q_1\} S_1 \{R_1\}, and therefore the proof still holds in the face of parallel execution. This allows to define the `cobegin` statement as follows:

\{Q_1 \land Q_2\} \textbf{cobegin} S_1 || S_2 \textbf{coend} \{R_1 \land R_2\}

Again, for simplicity, assume a program has only one `cobegin` statement.

In any operational model consistent with this and the other axioms, statements S_1 and S_2 can be executed concurrently, and execution of a `cobegin` terminates only when both S_1 and S_2 have terminated. No assumptions about the relative speeds of processes S_1 and S_2 are made.
Properties of Parallel Program

The following properties are the essentials to prove the parallel programs correct. With several processes are executed in parallel, their results can depend on the unpredictable order in which actions from different processes are executed. There are many important correctness properties for parallel programs but they vary on the requirement of the program. The properties discussed here are the basic for all parallel programs. The properties are:

- Resource invariant
- Mutual exclusion
- Blocking
- Termination

Resource Invariant

As we see from the Gries book [3], the notation \( \{Q\} S \{R\} \) express the partial correctness of statement S with respect to assertions Q and R. Since this is the basic of proving sequential programs from the book, I won’t discuss details of \textit{precondition}, Q, and \textit{postcondition}, R. The axioms for parallel programs require a new assertion I(r), the invariant for resource r, which describe the reasonable states of the resource. I(r) must be true when parallel execution begins and remain true at all time outside critical section for r. The axioms that I discussed before cobegin and with-when statements make use of this invariant.
**Mutual Exclusion**

Two statements are mutually exclusive when they cannot execute at the same time. The critical section is meant to discuss the mutual exclusion. Critical section prevent more than one statement operate on shared variables. One important thing about resource invariant, I(r), in critical section is we cannot assume that I(r) is still true after the execution of the critical section since another process may have control of the resource and I(r) may be temporarily false. I will discuss about this more while I am proving the program correct with example.

**Blocking**

Another important property not like the sequential program is the possibility that a program can be blocked before it accomplishes its goal. This can happen in parallel program because of with-when statements. A parallel program is blocked at with-when statement with \( r \) \textit{when} \( B \) \textit{do} \( S \) when resource \( r \) is not available or \( B \) is false. Generally blocking is not unusual case in parallel programs. The process may be blocked and then unblocked many times in the program. However, if an entire program is blocked there is no way to recover.

**Termination**

Program termination is an important property in proving programs correct even though there are correct programs that do not terminate. A sequential program can fail to terminate mainly for two reasons: an infinite loop and illegal operation like dividing by zero. For parallel program there is one more possibility: the program can be blocked forever. So, we can assure the program will terminate by showing that it does not become blocked forever and the rest will be like sequential program.
Problem in Parallel Program

In this paper, to discuss the proof of the parallel program I choose a very simple program which adds two variables with the mutual exclusion as my example. This problem is chosen because it is short and easy to follow, yet quite subtle; and the proof of its correctness is not trivial.

The program code is:

Program add
begin x:=0; y:=0; z:=0;
resource r(x,y,z): cobegin
  with r when true do
    begin x:=x+1; y:=1 end
  //
  with r when true do
    begin x:=x+1; z:=1 end
  coendend

Simply, this program has 3 variables and two parallel processes. In each process it increment the variable x and assigns 1 to y and z respectively. It is just the simple addition problem with parallel processes. Since it has the share variable x, the two process cannot run at the same time which what we called mutual exclusion. One process need to get the resource r to increment the variable x.

Numerous proof methods for programs have been proposed. The traditional and most widely used method and also the method we used in the class is the inductive assertion approach developed by Floyd and Hoare and extended to parallel programs by Owicki and
Gries [1] and others. For my purposes here, this is the approach that will be considered since it illustrates the main concept and is directly related to the topics that I learnt in the class.

A parallel program consists of two or more interacting processes. A process is a sequential program that execute concurrently with other process, delaying only when it need to interact with other processes or share variables. A proof of sequential program with statement list $S$ consists of a formal statement $\{P\} S \{R\}$, informally $P$ is true before execution of $S$ and $S$ terminates, then $R$ is true after execution of $S$. As I discussed in the previous section, concurrent execution is denoted by the $\text{cobegin}$ statement [4] and a parallel program has the form:

\[
\text{cobegin } S_1 \parallel S_2 \parallel S_3 \text{ coend}
\]

In Owicki and Gries [1], the effect of $\text{cobegin}$ is formally defined by the proof rule:

\[
\{P_i\} S_i \{R_i\} \quad 1 \leq i \leq n
\]

are interference-free

\[
\{P_1 \ldots n \ P_n\} \text{cobegin} S_1 \parallel \ldots \parallel S_n \text{ coend} \quad \{Q_1 \ldots n \ Q_n\}
\]

This rule says that given proofs of each of the processes, the combined effect of concurrently executing the processes is the conjunction of their individual effects as long as the proofs are interference-free.

Based on those rules, Andrew [7] developed a set of method that need to prove the parallel program:

1. Develop invariant assertions that state the desired relationships between shared variables.

2. Develop proofs of each process and show that each process maintains the invariant assertion.
3. Show that the proofs are interference-free (mutually exclusive).

Based on inductive assertion method, here is the program rewrite with pre and post conditions. The pre and post conditions are set off by braces {} and interspersed with the program statements.

Program add
{x=0}

begin x:=0; y:=0; z:=0;
    P:{y=0 n z=0 n I(r)}

resource r(x,y,z): cobegin

    P1: {y=0}
        with r when true do
        {y=0 n I(r)}
        begin x:=x+1; y:=1 end
        P3: {y=1 n I(r)}

    R1: {y=1}

    //
    P2: {z=0}
        with r when true do
        {z=0 n I(r)}
        begin x:=x+1; z:=1 end
        P4: {z=1 n I(r)}

    R2: {z=1}

coend

{y=1 n z=1 n I(r)}

end
{x=2}

I(r) = {x = y + z}

To prove the program is correct, we have to follow the 3 step method from Andrew [7].
Develop Resource Invariant

This is the key part of the method. The invariant assertions must be strong enough so that when it combined with the processes, they allow whatever property is of interest to be proved. However, they must be weak enough so the proofs of the processes do not interfere. In the above example, this is pretty simple enough. Since, this is the addition problem, it can get with the good guess $I(r) = \{x = y + z\}$.

Develop Proof of each Process

Proof of each process can be showed from the assertion from the program. Since, this program does not include the alternative command and iterative commands; it is very simple to prove as a sequential program.

As we can see from the program, obviously

$\{P1\} S1 \{R1\}$ and $\{P2\} S2 \{R2\}$ is correct.

Show processes are interference-free

Now the big time arrives. Essence of parallel program correctness is concurrent processes need to be mutually exclusive when they use the share variables. I would like to use the assertions from the program to prove that mutual exclusion is promised between those two processes. It means both processes do not increment the $x$ at the same time. I will prove that by assuming two processes increment $x$ at the same time and deriving a contradiction.

Assuming two processes execute at the same time:

$(P1 \lor P2 \lor I(r))$
\[ (\wp (S1, P3) \land \wp (S2, P4) \land I(r)) \]
\[ (\wp(\text{"x:=x+1; y:=1"}, P3) \land \wp(\text{"x:=x+1; z:=1"}, P4) \land x = y + z) \]
\[ (y:=1; x+1 = y + z \land z:=1; x+1 = y + z \land x = y + z) \]
\[ \text{False} \]

Unfortunately, I(r) can never be true when you execute both processes at the same time.

Gries and Owicki [1] prove that \((P1 \land P2 \land I(r)) \implies \text{false}\) is the sufficient condition for the P1 and P2 are mutually exclusive where \(\text{pre } (S1) \implies P1\) and \(\text{pre } (S2) \implies P2\).

Based on the theorem from Gries and Owicki [1], and \((P3 \land P4 \land I(r)) \implies \text{False}\), mutual exclusion is guaranteed.

Now, this program is satisfied all 3 methods of prove and it can be say that the program is correct. I chose the program with no alternative statements and iterative statements since my main objective for this paper is to go through the proof process for parallel program.

As we already seen, the proof of parallel program is an extension to the sequential programs and we did a lot of proof for sequential program in the class. Moreover, we need to make sure the program will terminate.

As I discuss in the previous sections, there are other important properties that is important in the correctness of the parallel programs. Those are free of forever blocking and termination of the program. Blocking is a necessary property of parallel program unless it will be blocked forever. In this program, the blocking statement needs to wait only for the resource, \(r\), to be available not the Boolean variable to be true (since there is no Boolean checking). By inductive assertion, we can see from the program that it can assure to be
unblocked at some time. Program termination is an important property for the correctness of both sequential and parallel program, although there are correct parallel programs which do not terminate. But this example program needs to terminate to be correct. A parallel program can fail to terminate for infinite loop, illegal operation and blocking. This program does not contain loop, and does not have illegal operation (as seen from assertion) and we just prove that there will be no blocking. So, termination can also be guaranteed.
Conclusion

This report has identified a few principles which have emerged on the proof of parallel programs. It discusses the particular importance of these methods in parallel programming where testing is very complex. These methods can be modified to prove more complex programs which use other synchronization operations instead of with-when. There are many important correctness properties for parallel programs beside the ones discussed here. However, this report can be taken as the discussion of basic properties of proving parallel programs.
Reference


5. L. Lamport, A New Approach to Proving the Correctness of Multiprocess Programs. ACM Transactions on Programming Languages and Systems, 1:84-97, 1979
