IT 11 064

Examensarbete 15 hp
September 2011

Model checking WOOL parallel library using SPIN model checker

Majid Khorsandi Aghai

Institutionen för informationsteknologi
Department of Information Technology

Abstract

Model checking WOOL parallel library using SPIN model checker

Majid Khorsandi Aghai

Verification and validation is the process of checking a product, service, or system to make sure it meets its specifications.
Model checking is an automatic technique for verifying finite-state systems, such as sequential circuit designs and communication protocols. The technique was originally developed in 1981 by Clarke and Emerson[8]. This technique has several advantages over theorem provers or proof checkers. The most important is that the procedure is highly automatic.

Wool is a C-library which is designed to be a really low overhead user-level task scheduler in concurrent programming environments. In order to achieve low overhead, Wool uses a novel algorithm for scheduling tasks between threads which is based on the work-stealing algorithms. This algorithm is called “Direct Task Stack”. In this project we have first studied the source code of library and created a high level model of it with our focus mainly on the “Direct Task Stack” algorithm. Then we tried to verify three important correctness properties of this algorithm in Spin model checker.

Spin was able to verify the three properties exhaustively when the test harnesses are small enough. For bigger test harnesses Spin was only able to verify the properties when the compression options were activated.

Handledare: Ahmed Rezine
Ämnesgranskare: Bengt Jonsson
Examinator: Anders Jansson
IT 11 064
Tryckt av: Reprocentralen ITC

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Verification and validation is the process of checking a product, service, or system to make sure it meets its specifications and fulfills its intended purpose. Model-based techniques for verification and validation of reactive systems, such as model checking, have witnessed drastic advances in the last decades. These techniques require a formal model which describes the system or component, and a property for the intended behavior that can be generated during the design phase in the software life cycle.

Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. The technique was originally developed in 1981 by Clarke and Emerson [8]. Quielle and Sifakis [18] independently discovered a similar verification technique shortly thereafter. This technique has several advantages over theorem provers or proof checkers for the verification of circuits and protocols. The most important is that the procedure is highly automatic. Typically, the user provides a high level representation of the model and of the specification to be checked. The model checker will either terminate with the answer true indicating that the model satisfies the specification, or give a counter example execution that shows why the formula is not satisfied [7].

Model checking is particularly well suited to verify requirements and correctness poroperties of parallel systems and also has the advantage of modeling parallel systems and algorithms. It automatically checks whether the system satisfies the properties. A large number of model checking tools have been proposed such as CHESS [15], UPPAAL [5] and SLAM [3].

One of the most successful tools for the automatic verification is Spin. This model checker [4, 13] is a general tool for verifying the correctness of distributed software models in a rigorous and mostly automated fashion. Unlike many model-checkers, Spin actually does model checking by generating C code. This technique saves memory and improves performance, while also allowing the direct insertion of chunks of C code into the model. Spin also offers a large number of options to further speed up the model-checking process and save memory, such as partial order reduction, state compression, bitstate hashing and weak fairness enforcement.

In this project we have focused on modeling the Wool parallel library in Spin model checker to verify it against some correctness properties. Wool is a C-library supporting fine grained independent task parallelism [9].

1.1 Background

Software may have bugs and it is the common term used to describe an error, flaw, mistake, failure or fault in a computer program or system that produces an incorrect or unexpected result or causes it to behave in unintended ways. Most bugs arise from mistakes and errors made by people in either a program's source code or its design, and a few are caused by compilers producing incorrect code. Better development process, enhancing people knowledge, effective collaboration and verification techniques may reduce the amount of mistakes and increase the reliability of software.

Verification techniques aim to find bugs in software systems and to check if they are built right or not. They are very helpful to find if a system conforms to its correctness properties. One of the most known verification techniques is model checking which in this project we apply in order to verify the Wool library. Wool is a library written in C programming language supporting fine grained independent task parallelism. In the term "fine grained independent task parallelism", a task is a unit of parallel executions which is able to create other tasks. In task parallelism each task has its own control flow in contrast to, for instance, data parallelism where the same operation is always applied to each data item in a collection [1]. Wool takes advantage of the independence of tasks in order to run them in parallel. Two tasks are independent if none of them writes a location that the other task reads or writes. Independent tasks can be executed in arbitrary order or in parallel with no difference in observable behaviour. Fine grained tasks are tasks that are relatively small in terms of code size and execution time. The Wool library takes care of the scheduling of the execution of these fine grained tasks.

The verification technique which we will use in this project is model checking. When the software itself cannot be verified exhaustively, we can build a simplified model of the underlying design that preserves its essential characteristics but that avoids known sources of complexity [13]. In the approach adopted here, the first step is to understand the program or algorithm we want to model and after that, to derive a high level model of it. The model should be as simple as possible yet it should preserve most of the behaviors of the original program or algorithm. Then the next step is to decide what properties of the model should be verified, how important they are in practice and to what extent can these properties be verified automatically with the model checker at hand.

1.2 Goal

The main goal of this project is to first derive a high level model from the source code of the Wool parallel library using the Promela modeling language for the Spin model checker; then to verify this model against some correctness properties.

1.3 Motivation

Like in other work stealing schedulers, in the Wool scheduler a number of threads(typically one) is assigned to each processor and each thread or worker executes tasks. A worker steals and executes tasks from other workers if it does not have any task itself. In most work stealing algorithms, each worker has a task stack with two pointers namely bot and top and synchronization between thief and victim workers is based on the values of those pointers. Unlike these algorithms, the “Direct Task Stack” algorithm introduced in Wool only uses a “state” field in each task for the purpose of synchronization which confers several benefits, like eliminating the overhead of task creation for tasks that are never stolen. Please refer to [9, 10] for further information about Wool’s synchronization mehotd and its benefits. Having these interesting properties, Wool and specially its underlying "Direct Task Stack" algorithm becomes a very interesting subject for model checking. There are different model checking tools one can use in this respect. Spin is a model checker developed by Gerard J. Holzmann [4, 13] for verifying communication protocols. It has since become widely used in verifying parallel algorithms and concurrent programs. It can verify and find bugs in models which are written in the Promela language. In this project we will generate Promela models from the Wool code and then analyze them using Spin.

1.4 Structure of the project

This section outlines the general structure of this report and gives a brief summary of its chapters.

Chapter 2 gives some background about this work, verification and modeling. The background is needed for understanding the rest of the report. Readers who have general knowledge about these areas can skip this chapter. Chapter 3 will introduce the Wool library and its underlying concepts. In Chapter 4 we will explain which parts of the Wool we modeled and how we did it. Chapter 5 discusses simulation of the model and its verification against a number of correctness properties. And finally, in chapter 6 we present conclusions and future work.

Chapter 2

Background

In this chapter we give a short background of our work and the tools and techniques we have used. Our work is verification of the Wool parallel library. Among the different availabe verification techniques like code and design review, static program analysis, testing and model checking, we used the model checking technique because it is an automated technique and it can check different kinds of properties. The tool we used for model checking is the Spin [13] model checker which performs verification of models written in the Promela modeling language [13].

The "Direct Task Stack" algorithm of Wool [9] which is responsible for fine-grained independent task parallelism is the interesting part of the Wool library for verification. We modeled and verified this part of the library using Spin model checker. In comming chapters we will completely describe how we modeled and verified the library but before that in this chapter we will give short introductions about the concepts the reader will see from here on.

2.1 Verification and Verification Techniques

The correctness of a software system is being checked by two processes. One of the processes is to check if the software is what the customer wants which is called validation. The other process is to check if the software is bug free and it matches the specification of the software which is called verification. One approach to verification is to manually inspect the code of the software. But this approach is becoming more difficult and time consuming as more and more complex systems are being developed. Testing and formal verification are two alternative methods for verifying a system. Both methods assume access to a so called specification of the software, i.e., a description of the correct behavior of the system. The methods compare the specification to the actual behavior of the system [19]. There are several verification methods, but some have more advantages. Simulation is one of these techniques that can be used on design level but it is difficult to make exhaustive. In addition, manual selection of test cases and input needs lots of work. Code and design reviews is another one. It is good at finding (some classes of) problems but needs organization and people. Static program analysis is one approach to analyze the source code by tools. It is completely automatic but can verify a limited set of properties (type-correctness, absence of some run-time errors) and unfortunately tools are available only for some languages and properties. Testing and model checking are among the most used and famous methods for verification.

2.1.1 Testing

Testing is the process of sampling the executions of a system according to some given criterion. Each execution is compared with the specification, and any mismatch is reported as an error [17]. Software testing is the process of evaluating an attribute of a program to see whether it satisfies its required results. An input is fed to the software and if the output is as it is expected we say the system has passed the test, otherwise it has failed. Feeding a set of these test cases to a software, different properties of the specification can be tested. Testing is the most "practical" technique that can verify a wide range of properties but can only be used on implementation. It is difficult to make it exhaustive and hard to make it reproducible for concurrent/distributed programs. Also manual selection of test cases and input needs work.

2.1.2 Model Checking

In model checking, a model of a system which describes the system behavior is algorithmically checked against the specification of system. The model is usually expressed as a directed graph consisting of nodes and edges(Kripke structure). The nodes represent the state of the program and the edges are representing the possible execution which changes the program state. Usually, a set of properties is associated with each node.

The properties represent the condition that should hold in a particular state of the program. Model checking is important for both validation and verification of a software system. The model of system can be compared to costumers needs for validating the system. It is also one of the techniques for model-based verification. By checking the formal model of system against its specification of the system, the correctness of system would be specified. Model checking can be done early in the design cycle, e.g., on design level. It is automated and can check different kinds of properties. Further more counter examples are another important feature of model checking which show a trace of model's execution that leads to a possible bug or error. In negative aspect model checking does not scale to very large models. A model must be constructed (at a suitable level of abstraction), and it must be maintained as system evolves.

2.1.2.1 The Model Checker Spin

Spin is a generic verification system that supports the design and verification of asynchronous process systems. It focuses on proving the correctness of process interactions [4, 13] and we use it as model checking tool for verification of concurrent algorithms in this project. Spin translates each process template into a finite automaton. Global behavior of the concurrent system is obtained by computing an asynchronous interleaving product of automata, one automaton per asynchronous process behavior.

The Spin verification procedure is based on an optimized depth-first graph traversal method. In order to reduce the number of states that must be explored to complete a verification, Spin uses a partial order reduction method presented in paper [16] by D. Peled. If size of a verification problem is too big to be verified exhaustively, Spin can perform a verification with approximately the same results of an exhaustive run in relatively small memory. For this purpose Spin uses bit-state hashing or supertrace technique [11, 12].

To check correctness properties of an algorithm in Spin, a model must be written in Promela that describes the behavior of the system, then correctness properties that express requirements on the systems behavior are specified; the absence of deadlocks, run time errors, memory leaks. Finally, the model checker runs to check if the correctness properties hold for the model and if not, to provide a counterexample(a computation that does not satisfy a correctness property).

2.2 Fine-grained Independent Task Parallelism

In a multiprocessor system, task parallelism is achieved when processors execute a different threads (or processes) on the same or different data. The threads may execute the same or different code. In the general case, different execution threads communicate with one another as they work. Communication takes place usually to pass data from one thread to the next as part of a workflow. As a simple example, if we are running code on a 2-processor system (CPUs "a" and "b") in a parallel environment and we wish to perform tasks "A" and "B" , it is possible to tell CPU "a" to preform task "A" and CPU "b" to perform task "B" simultaneously, thereby reducing the runtime of the execution. The tasks can be assigned using conditional statements. Task parallelism emphasizes the distributed (parallelized) nature of the processing (i.e. using threads), as opposed to the data (data parallelism). Most real programs fall somewhere on a continuum between task parallelism and data parallelism [22]. Two tasks are independent if none of them writes a location that the other task reads or writes. Two independent tasks can be executed in arbitrary order or in parallel with no difference in observable behavior (for instance memory content when both tasks have completed). In independent task parallelism, only independent tasks may be executed in parallel, obviating the need for any synchronization between concurrent activities. This is in contrast to the situation in more general multi-threading which allows dependencies between threads that need to be manged using explicit synchronization [9]. In parallel computing, granularity means the amount of computation in relation to communication, i.e., the ratio of computation to the amount of communication. So fine-grained parallelism means individual tasks are relatively small in terms of code size and execution time.Further more The data is transferred among processors frequently in amounts of one or a few memory words.

Chapter 3

The Wool Library

The Wool library involves around four thousand lines of C code, and includes many preprocessor branches. For example it uses different synchronization techniques based on the architecture of the machine it is being run on or to increase its performance Wool uses different optimizations which can be enabled or disabled by the user. On the other hand, Wool creates an arbitrary number of states during execution which means it is not feasible to verify the whole library exhaustively. Our solution for overcoming this problem was to build a simplified model of the underlying design of Wool which not only preserves its essential characteristics but that avoids known sources of complexity and hopefully can be verified exhaustively. In order to model Wool, the first task was to know the library itself and data structures and algorithms it uses to perform independent task parallelism.

In order to perform parallelism, Wool is using work-stealing scheduling to divide the work between working threads. Two scheduling paradigms have arisen to address the problem of scheduling multi-threaded computations: work sharing and work stealing. In work sharing, whenever a processor generates new tasks, the scheduler attempts to migrate some of them to other processors in hopes of distributing the work to underutilized processors. In work stealing, however, underutilized processors take the initiative: they attempt to "steal" tasks from other processors [6].

The Wool's scheduler is using a novel work stealing algorithm called "The Direct Task Stack" [9] with operations like `spawn`,`sync` and `steal`. In this chapter we will introduce this algorithm, its operations and the data structures it uses. We also describe the different synchronization techniques utilized in this algorithm.

3.1 The Direct Task Stack Algorithm

In Wool, each process is called a worker and each worker has a small data structure called a "worker descriptor" which contains the data structures for that worker. In particular it contains:

- An array of task descriptors for tasks spawned by this worker .
- A pointer to the next task to `steal`, used by thieves,called `bot`.
- A pointer to the latest spawned task, used by the owner to synchronize with its own tasks, called `top`.
- Various other things, including statistics counters.

The following piece of C code shows a short definition of worker data structure as we are going to model:

```
typedef struct _Worker {
  task      *top,
            *bot;
  long int  is_thief;
  Task   p[];
}
```

The `is_thief` specifies whether the current worker is a thief or not. We will later see how this field is used by the library. `p` is the array of task descriptors. A task descriptor is a data structure for holding information about tasks. It contains several data fields. The following piece of C code shows the definition of task descriptor in Wool library. Please note that the actual code both for worker and task data structures are longer and more complex but they essentially boil down to what we present here.

```
typedef struct task{
  void *f;
  balarm_t balarm;

}
```

In this code the `f` field is a pointer to the task function. This is the function that each task will run upon execution. The `balarm` field is used for different synchronization purposes which we will cover later in this chapter and next chapter.

The array of task descriptors works like a double ended queue(dequeue). Each worker process can `spawn` new tasks in its dequeue and each new task will be placed on `top` of other tasks. Then it will synchronize with these tasks in a Last In First Out(LIFO) order, it removes them from the dequeue and executes them. Here the `top` pointer is responsible to keep track of the last spawned task. Other workers, in case they are idle and don't have any ready task in their dequeue, can `steal` and execute tasks randomly from the bottom side of other processes' dequeue(where `bot` pointer points to). Among all the operations a worker process can execute, four of them are particularly important: `spawn`, `sync`, `wool_sync` and `steal`. Figure 3.1 shows pseudo code for these operations. In the actual Wool system, the implementation is slightly more complex due to additional optimizations and tests for task pool extension.

`spawn`: This operation takes the `top` pointer(this is where the worker will spawn the new task) and arguments of the task function . Then a new task is generated and its arguments are assigned. Finally the `top` pointer will be incremented. The spawned task either will be executed by the owner after performing `sync` operation or it will be stolen and executed by another worker.

`sync`: By executing this, a worker thread will try to remove the latest spawned task from its dequeue and execute it. Either the task is there and the operation will be performed successfully or the task is already stolen by some other worker in which case the owner will execute `wool_sync`.

`wool_sync`: This operation gets a task and its state as input arguments. First it will keep reading the state of the task as long as its value is equal to `EMPTY`. That is because we want to wait until the owner of that task write the new value of task's state. This is done in line 3 of `wool_sync` operation in figure 3.1. After this the `wool_sync` operation will try to get exclusive access to the task and checks its state. If the task is executable, it executes it, otherwise if it is stolen then it will start leap-frogging with the thief.

```
spawn(task *top, a_1,..., a_n ) {
top->a_1 = a_1;
...
top->a_n = a_n;
top->state = TASK;
top++;
}
```

```
steal( worker *victim ) {
task *t = victim->bot;
state s1 = t->state;
state s2;
if( is_task(s1) ) {
  s2 = cas_val(&(t->state),s1, EMPTY);
  if( s1 != s2 || victim->bot != t) {
    if( s1 == s2 ) t->state = s1;
  } else {
    t->state = STOLEN( self_idx );
    victim->bot = t+1;
    get_wrapper(s1)(t);
    t->state = DONE;
  }
}
}
```

```
sync(task *top) {
state s;
top--;
s = swap( &(top->state), EMPTY );
if(s == TASK)
  return top.f( top->a_1, ..., top->a_n
      );
else {
  wool_sync( top, s );
  return top->result;
}
}
```

```
wool_sync( task *t, state s ) {

while( s == EMPTY ) s = t->state;
if( is_task(s) )
  s = swap( &(t->state), EMPTY );
if( is_task(s) )
get_wrapper(s)(t);
else if( is_stolen(s) )
while( t->state != DONE )
steal( get_thief(s) );
bot--;
}
```

FIGURE 3.1: Wool operations

`steal`: This operation, as argument, takes the victim worker. It will try stealing a task from the bottom of the victim dequeue and then examines it's state. If `steal` was not successful, for example because another thief steals that task before this one then nothing happens and `steal` operation ends. Otherwise if thief succeeds in stealing the the task but `bot` pointer is changed, the thief can not continue stealing and backs off. But if thief succeeds and `bot` is not changed then it changes the state of stolen task to `STOLEN`, executes the task and then changes back the state of finished task to `DONE`. Backing off from stealing when the `bot` pointer is changing is to avoid violating the "NoMiss" property. This is completely explained in 4.2.4 and 5.2.1.

Figure 3.2 shows two consequent steps of the algorithm execution and status of different workers' dequeues in each step. At first step worker 1 has two tasks to execute, worker p has three tasks, worker q is idle and worker n has one task. At second step worker 1 synchronizes with last task, executes it and removes it from the stack. Worker q, who has no task to execute, steals one task from bottom of worker p and dequeue of worker n remains unchanged.

The operations `spawn`, `wrap` and `sync`, and `f` (the function that actually does the work of the task, not shown in the figure) are task specific, while `wool_sync` and `steal` are part of the run time system and used by all tasks. One advantage of having task

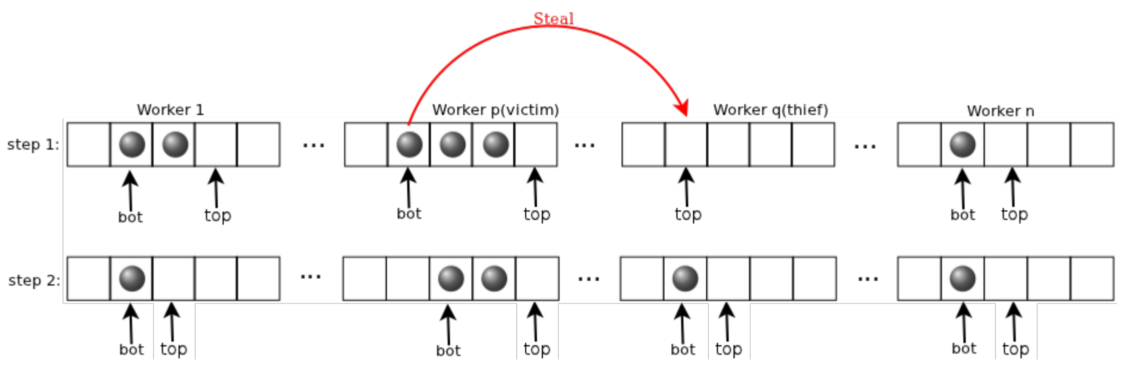

FIGURE 3.2: The Direct Task Stack

specific functions is that for example in `sync` a worker thread when synchronizing with an already spawned task, it can do a direct call to `f` which is the executable function of the task and execute it inline. This is visible to compiler optimization. A task descriptor has one field for each argument the function `f` of the task accepts. It also has an additional "state" field with the following values:

- EMPTY, signifying either a transient state during stealing where the task is not inside the dequeue because it is stolen or that there is no task stored in that task descriptor.
- TASK, which means that there is a task with a `f` function that can be stolen or inlined.
- STOLEN(i), denoting a task that has been stolen by worker i (knowing the thief is necessary for leap frogging [20]. Leap-frogging is stated later in this chapter).
- DONE, for a task that has been stolen and where the thief has completed its execution.

3.1.1 Synchronization Techniques

A worker thread may call two different operations to synchronize with the spawned tasks. Normally each worker after spawning a task calls `sync` operation to synchronize. Here two scenarios may happen: either the task is available in the dequeue and worker will execute it, or the task is already stolen by another worker process in which case the owner will call the public `wool_sync` operation. Normally when there are more than one task inside the dequeue, worker can execute tasks by removing them from top end and thieves can `steal` tasks from the bottom end with no need for synchronization. But when there is only one task inside the dequeue there is possibility that the owner and a thief want to get that task at the same time. Here the algorithm needs to use a synchronization technique. Most workstealing algorithms synchronize thief and victim based on values of `top` and `bot` pointers [2], but the direct task stack algorithm uses the `f` field in the task descriptor instead. At a `sync`, the owner acquires the task using an atomic exchange on the `f` field. This way thieves are decoupled from the frequently changing `top` pointer and the pointer can be kept private to the owner.

Wool uses two different synchronization techniques based on hardware architectures it is run on. On intel itanium architecture(ia64) it uses THE_SYNC method and on other architectures it uses EXCHANGE method. Since in this report we are concentrating on EXCHANGE method here we give a short description of what this method do and how it performs synchronization.

3.1.1.1 EXCHANGE

The EXCHANGE version that we are concentrating on in this report is using an atomic exchange primitive for synchronization. So different threads can access one thread's task stack concurrently. As already explained each task in Wool library has a function field which we call it `f`. In the EXCHANGE version Wool performs the synchronization on `f` field; If `f` is pointing to a valid function definition in memory then the task is ready inside the task stack, i.e not stolen. Otherwise the task is not present in the stack because it is stolen. In a work stealing algorithm when thief wants to `steal` a task, it should make sure no one else has already access to that task, i.e it should gain exclusive access to that task first. As stated already, a thief will `steal` the task that victim's `bot` pointer points to. This is normally the first task at the bottom end of victims dequeue. We call the task thief tries to `steal` as the "victim task". In EXCHANGE version thief first exchanges the `f` of victim task with the predefined value "T_BUSY". T_BUSY is used to show that the task is currently owned by another process and not stealable. So if another thief comes to `steal` that task it will notice that `f` of the task is T_BUSY and it backs off. When the thief finishes executing the stolen task, it will change the value of `f` to "STOLEN_DONE" which is another predefined value showing the task is fully executed and not available any more.

3.1.1.2 Leap-frogging

When a `sync` operation finds the task is stolen but not yet completed, it will be blocked until the joined task completes. In order for the scheduler to be greedy (which amounts to ensuring that there is not simultaneously stealable work and idle processors [6]), it needs to find other work to do for that processor in the meantime. One alternative is to randomly `steal` work from the other workers. But this may lead to loss of parallelism if the task the `sync` waits for completes but the stolen work is not completed while there are idle processors and no other worker in the system. The solution that Wool uses is leap frogging [20] where the worker executing the `sync` is constrained to stealing only from the worker that stole the task. Then it can not be the case that the stolen code has not completed when the `sync` has become unblocked [9]. Intuitively that is because the victim steals back a smaller part of its own task from the thief. So this smaller part should be executed faster than the bigger part which is being stolen.

Chapter 4

Building the Model

In this section we will mainly focus on describing how we have translated (part of) the Wool code written in C to Promela and how we have created the model. We will try to show a direct correspondence between each part of the C code and its translated Promela code. Since Promela is a modeling language, rather than a conventional programming language, it lacks several run time mechanism, such as dynamic memory management and function call-return. In respect to this the modeling power is bounded by some restrictions imposed by the state explosion problem and by some specific limitation of Promela [13]. Therefor we need to imitate such mechanisms inside Spin.

We will first explain how we have modeled Wool's worker threads in Spin. Each worker thread may perform any of four operations `spawn`, `sync`, `wool_sync` and `steal`, so in this chapter we have also shown how these operations are modeled and we have also drawn a direct comparison between the model code in Promela and its corresponding code in C.

Finally, in section 4.3, we will show how we have modeled the task functions in Promela as it was needed for the model to work properly.

4.1 Worker threads

As mentioned in section 3.1, in Wool each worker thread has its own array of task descriptors(task stack) and we need to model it. Furthermore, each worker has two pointers namely top and bot which are pointing to nodes of its task stack. Each worker's bot pointer should be accessible by the other workers but it's top pointer is private to itself and no one else will access it. In our model we have NoOfWorkers number of processes with general name of "thread". NoOfWorkers is a constant defined at the beginning of model. So for example if NoOfWorkers is 5 we will have 5 "thread" processes. Programs in Promela are composed of a set of processes. Each process includes a set of instructions which will execute when it is called. Each process in Promela has a read-only identifier which automatically gives a unique number to each process as it is instantiated. This identifier is accessible using _pid variable and using it we can access and work with different thread processes. We will use _pid in our model to access different processes. Each process in Spin is executed in two ways. One way is to start them as soon as we start executing the whole model. These processes are called active processes. To define this kind of processes one should add the keyword active at the beginning of definition line. Other group of processes will not start execution automatically and need a second process to call them. Please refer to [4] for further information about processes in Spin. Here is how we define our worker threads.

```
active [NoOfWorkers] proctype thread()
```

Then we define the worker and task data structures similar to how they are defined in the Wool library. Please note that these data-structures are already discussed in section 3.1. Figure 4.1 shows the C code for defining the worker and task data structures and their corresponding definition in Promela.

The astute reader will notice that the Promela version of task structure has one extra field "s". This field is later used for verification purposes and will be explained in Chapter 5. Now we are going to introduce the operations each worker process can perform and the way they affect the data-structures.

```
typedef struct _worker {
  Task      *top,
            *bot;
  long int  is_thief;
  Task    p[];
}


typedef struct task{
  void *f;
  balarm_t balarm;

}

```

Data-structures definition in C

```
typedef worker{
pointer   bot;
pointer   top;
bit    is_thief;
task     tStack[StackSize]
}


typedef task {
function  f;
byte     s;
b_alarm   alarm
}

```

Data-structures definition in Promela

FIGURE 4.1: Wool Data Structures and their definition in C and Promela.

4.2 Operations

4.2.1 Spawn

Spawning a task makes it available for parallel execution, but does not guarantee that it will actually be executed in parallel. If a processor becomes idle it tries to `steal` spawned but not yet executed tasks from some other processor, and it is only such stolen tasks that execute in parallel with other tasks spawned on the same processor [9].

In the Wool library each worker thread has its own task stack. At a `spawn` operation, a new task will be generated and its data fields like `f`, `balarm`, etc will be set and then this new task will be placed at the cell where `top` pointer currently points to. Then the `top` pointer will be incremented by one. But the `bot` pointer will remain unchanged. Figure 4.2 shows the C code of `spawn` operation and its translation in Promela. Please note that the codes we are presenting from now on are short version of the actual code. That is for better understanding and preserving the space. The complete code of the library is available at appendix A. The figure also shows the status of a worker's task stack before and after `spawn` operation.

```
inline void spawn(Task *top){
  TD *p = (TD *) top;
  p->balarm = NOT_STOLEN;
  //get the owner of task
  Worker *self = get_self(p);
  //next line sets the f function of the task
  STORE_PTR_REL(&(p->f));
  top++ ;
}
```

spawn code in C

```
inline spawn(top , func){
workers[myid].tStack[top].alarm = NOT_STOLEN;
//the s field is added by us and
//will be used for verification purpose
workers[myid].tStack[top].s = 1;
workers[myid].tStack[top].f = func;
top++;
}
```

spawn code in Promela

FIGURE 4.2: A task stack before and after `spawn` operation. The big ball shows a new spawned task.

4.2.2 Sync

There are two different synchronization operations in Wool library. First is the `sync` which is private to each task. It is executed by each worker after a `spawn` operation. By executing `sync`, the worker first decrements its `top` pointer to point to the top most task at its stack. Here two scenarios may happen: either the task is still in the stack so worker thread does a direct call to the `f` function of the task and runs it. This way the task is considered done and it will be removed from the stack. Another scenario that may happen here is that worker thread comes to `sync` with the task, but the task is not there, i.e stolen. Then it will call the other synchronization operation which is public to all worker threads and is part of the run time system. In the Wool source code this second synchronization function is called `wool_sync`. Figure 4.3 shows the source code of `sync` in C and its translation in Promela.

```
inline RTYPE SYNC(Task *top)
{
Task *q = top;
void (*f) = T_BUSY;
top-- ;
balarm_t a = NOT_STOLEN;
EXCHANGE( f, q->f );
if(f > T_LAST)
{
  // execute the task
  TD *t = (TD *) q;
  return CALL(top, t->d.a.ARG_1)
    ;
}
else
{
  //the task is stolen
  wool_sync(top, a);
  return ((TD *)q)->d.res;
}
}

```

sync code in C

```
inline sync(top , channel){

top--;
f = T_BUSY;
a = NOT_STOLEN;

EXCHANGE( f, workers[myid].
    tStack[top].f );

if
::f > T_LAST ->
  workers[myid].tStack[top].s++;
  run do_f(myid , f , channel,
    top);
::else ->
  wool_sync(top,a,channel);
  run do_f(myid , T_DONE ,
    channel , top);
  fi
}

```

sync code in Promela

FIGURE 4.3: `sync` operation's code in C and Promela

4.2.3 Wool_sync

As mentioned in the previous section, when a worker thread comes to `sync` on a task at its `top` pointer, the task may be stolen by another in which case the victim will call the `wool_sync` operation. By executing this operation, worker first will check to see if the task is already done by the thief or not. If yes it will do nothing and simply exit the operation. But if the task is still in hands of thief, then the leap frogging process will begin; The victim will start stealing tasks from thief worker and execute them. It will continue stealing and executing tasks from thief while its original task is being executed by the thief. Figure 4.4 shows a comparison between `wool_sync` code in C and its translation in Promela.

4.2.4 Steal

Stealing happens when there are idle worker threads and available job, i.e stealabe tasks in one or more other threads. When a worker thread spawns a task, this task will either be executed by the owner or it will be stolen and executed by another worker. When a thread wants to `steal` a task from a victim, it sets its `tp` pointer to `bot` pointer of the victim. As we already know `bot` pointer is always pointing to the next stealable task, if

```c
void wool_sync(Task *t,balarm_t a )
{
Worker * self = get_self ( t );
// the thief is done executing the task
if( a == STOLEN_DONE ) {}
// the thief steal ownes the task
else if( a > B_LAST ) {
int done = 0;
int thief_idx = a;
do {  // leap-frogging
  int s_out = SO_NO_WORK ;
  s_out = steal(self->idx, thief_idx,(Task *) t + 1, 0 );
  if( t-> balarm == STOLEN_DONE ) {
    done = 1;
  }
} while ( ! done );
} else {
     // the task is niether done, nor owned by thief.
     // error situation.
    exit ( 1 );
}
if( self->bot > t ) {
  // the stolen task is now executed so we need
  // to update the bot pointer
  self->bot = (Task *) t;
}


```

wool_sync code in C

```
inline wool_sync(top,a,chl){


if
:: ((a == STOLEN_DONE))->
  skip;
:: (a > B_LAST) ->
  do  //leap-frogging
  ::steal(a,top + 1,chl);
    IF (workers[myid].tStack[top].alarm == STOLEN_DONE) ->
      break;
    FI
  od
:: else -> assert (false);
fi;

IF (workers[myid].bot > top) ->
  workers[myid].bot = top
FI;

}

```

`wool_sync` code in Promela

FIGURE 4.4: `wool_sync` operation's code in C and Promela

any. Then using the atomic exchange operation, one thief will get exclusive access to the stolen task. It will then call the `f` function of the task i.e execute it and finally the thief will inform the victim its task is finished. figure 4.6 shows the source code for `Steal` operation in C and its translation in Promela. One important point in `steal` operation is that thieves should always check if the task they are stealing(which is the task `bot` points to) is actually the bottom most task in the stack and there is no other task below the `bot` pointer. Because each thief after performing the `steal` operation will increment the `bot` pointer. So if there was a task below the `bot`, it will remain hidden from the next `steal` operation. Eventually there is the risk that this task remains hidden from all `steal` operations and never get executed. So thieves always make sure there are no unstolen tasks below the task they are trying to `steal` and in case there is such task, they will back off and abort `steal` operation. But if there is no unstolen task remained under `bot` pointer, thief will complete `steal` operation and execute the task. This is visible in the source code of figure 4.6 between lines 18 through 25. figure 4.5 also shows a scenario which may lead to the described problem. In fact this problem is one

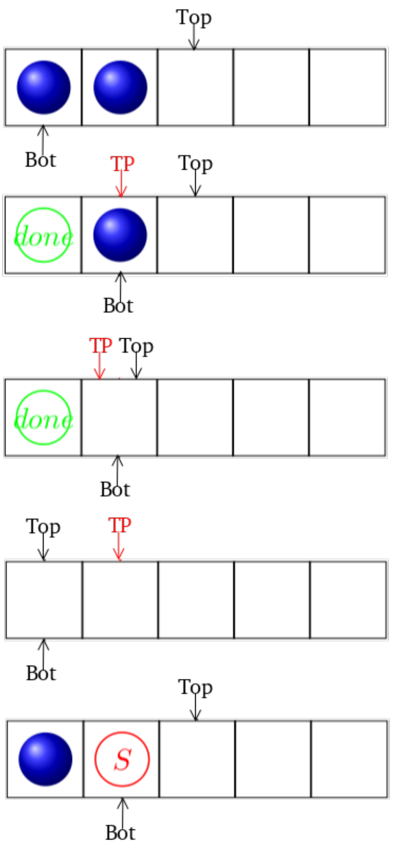

FIGURE 4.5: A scenario which violates “NoMiss” Property

important correctness property of Wool library that we are going to verify in our model which is described fully in next chapter.

4.3 Simulation of Task Functions

As we mentioned earlier in 3.1, each task has a function pointer which is pointing to the start of the function definition that the task upon synchronization calls. In our verification we needed to model the functions because otherwise we wouldn’t be able to simulate the situation that a victim steals a task back from the thief(leap-frogging). Since the function each task calls is independent from the behavior of direct task stack algorithm we decided to abstract it away in our model by simply replacing the `f` field of the task structure in C code with an integer field in our Promela code. Furthermore in a real run of a wool program, the function of a task may contain new `spawn` operations which will generate new tasks. In order to simulate this behavior and since we are not aware of the body of functions in a task, we needed a way to simulate the situation of

```c
static int steal( int self_id, int victim_id)
{
 volatile Task *tp, *ep;
 void (*f) = T_BUSY;
 Worker *self = workers[self_idx];
 volatile Worker *victim = workers[victim_idx];
 tp = victim->bot;
 if( tp->balarm != NOT_STOLEN || tp->f <= T_LAST ) {
   return SO_NO_WORK; //no task to steal ...
 }
 if(tp->balarm == NOT_STOLEN && tp->f > T_LAST) {
     EXCHANGE( f, tp->f );
     if( f > T_LAST ) {
      int all_stolen = 1;
      tp->balarm = self_idx;  // keep the id of thief for leap-frogging
      // in following for loop we check if there
      // are unstolen tasks below bot pointer
      for( ep = victim->bot; ep < tp; ep++ ) {
  if( ep->f > T_LAST ) {
    all_stolen = 0;
        }
      }
      if( all_stolen ) {
  // no task left below bot, so update bot
  victim->bot = (Task *) tp+1;
      }
  //back off
      else{
  tp = NULL;
      }
  }
  if( tp != NULL ) {
     //execute function of stolen task
     f(top, (Task *) tp );
     STORE_INT_REL( &(tp->balarm), STOLEN_DONE );
    return SO_STOLE;
  }
  return SO_NO_WORK;
}
```

Listing 4.1: steal code in C

```
inline steal(v_id, top, channel){
f = T_BUSY;
i = 0;  /* FOR loop counter*/
tp = workers[v_id].bot;
all_stolen = 1;

if
:: (!workers[v_id].is_thief) ->
 if
 :: (workers[v_id].tStack[tp].alarm == NOT_STOLEN) &&
    (workers[v_id].tStack[tp].f > T_LAST ) ->
 EXCHANGE(f,workers[v_id].tStack[tp].f);
 if
 :: (f > T_LAST) ->
   workers[v_id].tStack[tp].s ++;
   workers[v_id].tStack[tp].alarm = myid;
   FOR(i,workers[v_id].bot,tp)
     IF (workers[v_id].tStack[i].f > T_LAST) ->
       all_stolen = 0;
     FI
   ROF(i,workers[v_id].bot,tp) ;
   IF(all_stolen) ->
     workers[v_id].bot = tp + 1;
   FI;
 :: else ->
   tp = Undef;
 fi;
 :: else ->  tp = Undef;
 fi;
 IF (tp != Undef) ->
   workers[myid].is_thief =0;
   run do_f(myid , f , channel , top );
   channel ? _ ; // wait for the task function to complete
   workers[myid].is_thief =1;
   workers[v_id].tStack[tp].alarm = STOLEN_DONE;
 FI;
:: else -> skip;
fi
}
```

Listing 4.2: steal code in Promela

Figure 4.6: `steal` code in C and its translation in Promela

spawning new tasks during execution of the current task in our model. We offered a solution based on the famous Fibonacci series [21]. We say each value of `f` in our model has an implicit meaning with itself; if current value of `f` in a task is 5, it means we are going to calculate Fibonacci sequence from 1 to 5 in a recursive way. So we need to first calculate Fib of 4 and Fib of 3. But for calculating Fib of 4 we again need to have Fib

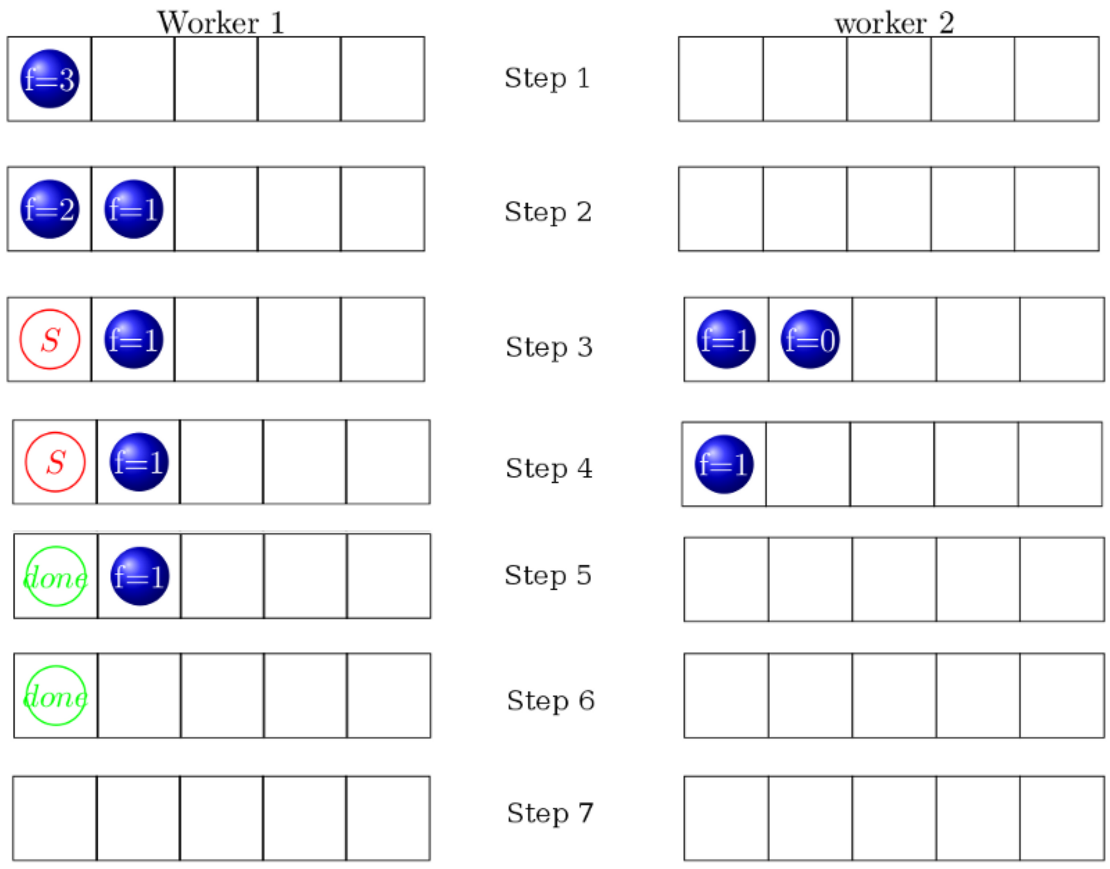

FIGURE 4.7: Two worker threads executing a task concurrently

of 3 and Fib of 2 before hand and so on. So what we do is that we start a worker thread and this worker thread spawns a task with `f` initialized to for example 5. At the same time we start another worker thread with an empty task stack. Since this worker has no task to execute, it will `steal` tasks from the first worker. Each worker when it wants to `sync` with a spawned task will look at its `f` value(if the task is present and not stolen). If the value is bigger than 1, it will `spawn` two new tasks with `f=f-1` and `f=f-2` but if the value is 1 or 0, the task will be executed which here means simply removed from the stack. Since Spin does not support recursive calls, we used the method presented in [14] to simulate recursive calls in our model. Briefly in this method we utilize Spin processes as functions. But since Spin processes do not return any value we needed a way to simulate return values in processes. This is done using channels. The caller process and the called process use a common channel between themselves to transfer values. This channel is defined by caller process. For further information about this method please refer to referenced paper by K. Jiang. Figure 4.7 shows a simple run of the model.

In this figure the following sequence of events happen:

*Step*1 : Worker1 spawns a new task with `f` value 3.

*Step*2 : Worker1 performs `sync` on that task and since the task is ready and no other worker has stolen it, it will execute it which means remove that task from the stack and `spawn` two new tasks with `f` values 2 and 1 respectively. Now worker2 starts working and since it has no ready task in its task stack it starts with stealing a task from worker1.

*Step*3 : Worker2 steals a task from bottom of worker1's task stack and executes it. Since this task has `f` value 2, after execution, worker2 will `spawn` two new tasks 1 and 0.

*Step*4 : Worker2 `sync`s with task 0 and executes it and removes it from the stack. It continues executing its tasks.

*Step*5 : Worker2 will execute 1 and removes it from the stack as well. Then it will inform the worker 1 that its stolen task is now executed by writing `done` on the place of stolen task.

*Step*6 : Worker1 after a delay continues to `sync` with its top most task which is task with `f` value 1. It will execute and remove it from the stack and continue to `sync` with the stolen task and since this task is already stolen and executed it will just remove it from the stack.

*Step*7 : Execution of the original task at this step is considered done.

In this chapter we explained how we modeled a part of Wool parallel library in Spin model checker. In next chapter we will show how we used this model to verify Wool against some correctness properties.

Chapter 5

Verification and Simulation

The Spin model checker is mainly used for two purposes: Simulation and Verification. Simulation means running one possible trace of the model from the beginning. It is usually used to check the constructed model and watch its behavior. Verification is the process of checking all possible traces of the model against a given correctness property. If any of the traces violates the given property then Spin will record the trace up to the location where violation happens and present this recorded trace to the user.

In this chapter we will first describe how we have used Spin's simulation to check our model and make sure it behaves as the original Wool library. After that we will define some correctness properties and describe the techniques we have used to help Spin to verify these properties. Finally we will present the results of verifications.

5.1 Simulation

The Spin model checker has the ability to run a model by performing a simulation. It means to choose one trace among all possible traces a model generates and execute it. Spin can choose this trace in three different ways:

- Random
- Interactive and
- Guided

In random simulation Spin will start running the model. Each time it reaches to a non-deterministic choice in the model it will choose one of the branches randomly until all processes are stopped. Another type of simulation is Interactive simulation. In this method, Spin starts running the model until it reaches a non-deterministic choice. Then Spin will present different choices to the user and asks the user to select his desired choice. The user can choose this method to reach to his desired state in the model. And finally the last method of simulation is guided simulation. In order to perform guided simulation, Spin needs an input file which contains the whole trace for simulations. This file can be generated as a result of a previous verification [13].

We use Spin's random and interactive simulation options to run our model and to check if it works the way intended and it behaves exactly as Wool behaves.

5.2 Verification

After building the model and running different simulations using Spin to make sure that our model preserves the behavior of the original Wool library, we are going to verify our model against some correctness properties. The correctness properties we have checked in this report are: "no unstolen task below `bot` pointer", "no task stolen twice" and "no task both stolen and inlined". We call these properties "NoMiss", "StealOnce" and "StealOrInline" respectively. For the selected size of our model(described below), Spin is able to verify exhaustively and does not report any error which means the model preserves the properties. For each property we will show a scenario which leads to a bug if certain conservations in the model are removed.

5.2.1 NoMiss corretness property

As we mentioned earlier in section 4.2.4, Since `steal` operation updates the `bot` pointer, because of the absence of explicit synchronization in EXCHANGE synchronization method, this may cause the situation where a stealable task remains below the `bot` pointer which will invalidate Wool's implicit synchronization protocol. So making sure no stealabe task is left below the `bot` pointer is the first and most important correctness property of Wool we want to verify with help of our model. Figure 5.4 shows a possible trace of the program that leads to the situation. The figure shows a worker thread who first spawns two new tasks. The bottom task is stolen by another worker and the bot pointer gets incremented. Then the thread comes to synchronize with its tasks but gets delayed. Another thief comes to steal the second task from the owner so it points its tp pointer to where victim's bot points to. Then the thief gets delayed and the owner continues with synchronization. It synchornizes with both tasks and then spawns two new tasks. Now the thief thread is activated and wants to continue its steal. Since the tp pointer is pointing to a valid task(although this is not the previous task thief tried to steal) thief will succeed and steal the task and increment the victim's bot pointer. As a result there remains one stealable task below the bot pointer of owner thread which is not visible to other steal operations.

In order to overcome this problem Wool uses a simple solution. Whenever thief wants to update victim's bot pointer it first checks if there are any unstolen tasks left below it and if yes, thief will not increment the bot pointer. The following lines of code shows how we modeled this solution in Spin.

```
 FOR(i,workers[victim_id].bot,tp)
      IF (workers[victim_id].taskStack[i].f > T_LAST) ->
        all_stolen = 0;
      FI
ROF(i,workers[victim_id].bot,tp)  ;
```

The code above checks all the tasks that are currently below the bot pointer in victims task stack. If any of them is not stolen yet, then the variable all_stolen will be assigned 0. This variable has initial value of 1. After this step program will check all_stolen. If it is 1 it indicates there are no unstolen tasks left below bot and it can be updated:

```
IF(all_stolen) ->
  workers[victim_id].bot = tp + 1;
FI;
```

So Wool uses the variable all_stolen as a flag. If all the tasks below the bot pointer are already stolen then this flag is true and the steal operation can proceed and the bot pointer can be updated. But if this flag is false then the steal operation can not succeed. So obviously Wool is using a very simple method to preserve the "NoMiss" property. Furthermore in the above piece of code no mechanism is protecting the IF command, So

```
active proctype NoMiss_monitor(){
int i;
atomic{
if
:: (workers[0].bot > 0) ->
  FOR(i,0,workers[0].bot-1)
  IF (workers[0].tStack[i].f > T_LAST) ->
    assert(false);
  FI
  ROF(i,0,workers[0].bot-1)
:: else -> skip
fi;
}
}
```

FIGURE 5.1: A monitor process for verifying the "NoMiss" property

there is the risk that between checking the `all_stolen` and updating the `bot` pointer, the thread gets delayed and then another thread changes the `bot` location in such a way that a task is left below the `bot` and the "NoMiss" property is violated. Considering these we wanted to make sure that Wool's simple protection mechanism works and preserves the correctness property. In order to achieve that, we added a monitor process which will check this correctness property during all steps of running model and we call it `NoMiss_monitor`. A monitor is an active process that always runs and checks a specific property during execution of the model. If at any time during execution, the correctness property is violated the monitor process will report the violation. Figure 5.1 shows the body of monitor process.

In order for the model to be able to generate a trace that violates this correctness property we also needed to provide a suitable input(test harness), proper number of worker processes(i.e as few as possible while at the same time able to produce the buggy state) and a good initial value for `f`(i.e as low as possible) . Figure 5.2 shows our test harness for this purpose. For this test we tried our verification with two worker processes and `f` initialized to one.

The code in figure 5.2 checks the `_pid` which is Spins internal variable for keeping the id of current running process [13]. If it is 0 it means currently the worker 0 is running and it will spawn two tasks repeatedly and it does this twice. The other worker(ie. the process with `_pid`=1) will try to `steal` from worker 0 twice.

```
if
:: (_pid == 0) ->
  f = initialF;
  spawn(workers[_pid].top , f);
  spawn(workers[_pid].top , f-1);
  sync(workers[_pid].top , return_channel);
  return_channel ? _ ;
  sync(workers[_pid].top , return_channel);
  return_channel ? _ ;

  f = initialF;
  spawn(workers[_pid].top , f);
  spawn(workers[_pid].top , f-1);
  sync(workers[_pid].top , return_channel);
  return_channel ? _ ;spawn(workers[_pid].top , f);
  sync(workers[_pid].top , return_channel);
  return_channel ? _ ;

:: else ->

  steal(0,workers[_pid].top,return_channel);
  steal(0,workers[_pid].top,return_channel);
fi;

}
```

FIGURE 5.2: A test harness we used along with `NoMiss_monitor` for verification of model against the "NoMiss" property

5.2.2 StealOnce and StealOrInline corretness properties

The second correctness property we want to verify using our model is "StealOnce". We want to verify that a task can not be stolen twice. Another similar property we can verify in our model is "StealOrInline" which states that no task should be both stolen and inlined. In this part we add codes to our Promela model to be able to verify these two properties together. In order for the model to be able to verify these properties, we add a new field `s` to task data type. This is a counter, which at every linearizion point of `Steal` and `sync` we increment its value by one. `S` has initial value of one, So if a task is stolen more than once or both inlined and stolen then `s` will have a value grater than 2. Therefore we can verify these two properties by always checking the value of `s` is equal or smaller than 2. For this purpose we add another monitor process responsible for checking the value of `s`. Figure 5.3 shows the Promela code of this monitor. The monitor will always make sure that `s` can never reach a value bigger than 2. Since this monitor is checking the property that a task should only be stolen or inlined once we call it `Once_monitor`.

In order to prevent violation of these properties the EXCHANGE synchronization mechanism in Wool is good enough. In this mechanism whenever a thief or a an owner

```
active proctype Once_monitor(){
int i;
atomic{
FOR(i,0,StackSize-1)
  IF (workers[0].tStack[i].s > 2) ->
    assert(false);
  FI
ROF(i,0,StackSize-1)
}
}
```

Figure 5.3: Monitor for "StealOnce" and "StealOrInline" properties

of a task wants to `steal` or inline a task, it first exchanges its `f` with an invalid value atomically which will prevent others to interleave during the exchange process. After the exchange is done, the task will have an invalid value in its `f` field so when other workers or the owner come to run another exchange on the task, they will note that the task has an invalid `f` value meaning that the task is not available(i.e already stolen or inlined). This way each task at any time can be stolen or inlined only by one worker process and that process is the one that can successfully execute the atomic exchange operation on the task before others. If this exchange operation was not performed atomically then there was the possibility these correctness properties are violated.

5.3 Verification Results

5.3.1 Verification results for NoMiss property

All verifications were conducted on a 3.00 GHz Pentium 4 processor with 2.2 GB of usable RAM using Spin Version 5.2.5. We used most of default XSpin settings for all verification attempts, except when we increased the memory limit from 128 MB to 2 GB to allow the search to complete. In cases where verification did not complete with default parameters within physical memory limits, verification with compression was performed.

As we mentioned earlier in this chapter, wool uses the variable `all_stolen` to prevent the violation of NoMiss property. Wool does this by checking the value of `all_stolen` at every `steal` operation and if it is 0, which means all the tasks below `bot` are not stolen yet, then Wool stops stealing otherwise finishes it.

In order for us to make sure that in our model, the test harness and provided inputs are able to generate the trace which violates this property, we first removed the check that Wool does on `all_stolen` from the `steal` operation. Then we verified the model in Spin. The following table shows the result of verification in this case. Spin was able to find the bug and report an error trace. Figure 5.4 shows a scenario based on verification results that causes the violation of "NoMiss" property . What happens here is that a worker thread first spawns two tasks consequently. Then another worker thread steals one task from its bottom, executes it and marks it as done. It also updates victim's `bot` pointer and increments it by one. Then another thief comes to `steal` from this worker and points its `tp` pointer to where `bot` points to but it gets delayed for any reason. The worker thread `sync` with top most task in its stack and then with the task which was stolen and done by the other thief worker. When synchronizing with a stolen task, owner will decrement the `bot` pointer so `bot` will now point to where `top` points. Note that a thief is still interested in stealing and its `tp` pointer is still pointing to a cell but it is waiting to be preempted. Finally the owner spawns two new tasks and then the thief eventually succeeds in stealing and then it increments victim's `bot` pointer. Now, as obvious in the picture, there is one task left under the `bot` pointer which is hidden from thief threads.

Time(s)	Used memory(MB)	Vector size(Byte)	State stored+Matched	Search Depth
0.04	4.258	263	14191	173

Table 5.1: Verification results when model violates "NoMiss"

After this step we added the check on `all_stolen` to the `steal` operation and using the same inputs and test harness we run the verification again. Spin verifies the model without any error. Then we increased the size of input parameters to have a more realistic verification. Spin was able to verify the model with initial `f` set to 2 also using default verification parameters. It was also able to verify the model with `f` set to 3 but we needed to activate compression option. Table 5.2 shows results for `f`=3 and having compression enabled.

Time(s)	Used memory(MB)	Vector size(Byte)	State stored+Matched	Search Depth
23.9	95.169	519	3083339	571

Table 5.2: Verification results when model preserves "NoMiss"

for `f` values bigger than 3 Spin was not able to verify the model exhaustively. So using our model we showed that for the small test harness we fed the model with, small

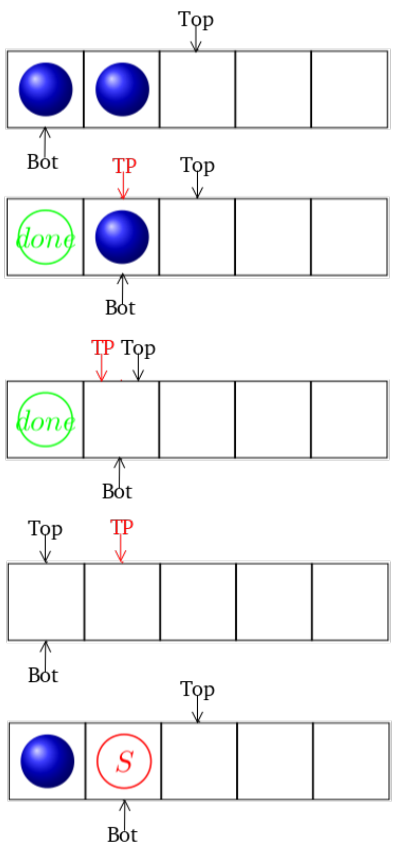

FIGURE 5.4: A scenario that will violate NoMiss property

initial values of `f` and two worker threads the simple protection mechanism that Wool uses to preserve the correctness property "NoMiss" is actually working.

5.3.2 Verification Results for StolenOnce and StolenOrInlined properties

As explained in 3.2.1.1, in EXCHANGE synchronization method, Wool is synchronizing different working threads on the `f` field of the task they access concurrently. All threads who want to own a task will perform an atomic exchange on the `f` field and exchange its value with a predefined value(invalidates it). The first successful thread will own the task and other threads will notice that the `f` value is not valid any more so they will back off. If this synchronization mechanism does not work properly, then the properties "StolenOnce" and "StolenOrInlined" may be violated because for example two threads may `steal` one task at the same time or a thread `steal` a task while the owner is executing it. Therefore in order to make sure that this synchronization mechanism is enough to preserve the two correctness properties, we first used Spin verification to observe what happens if we alter the mechanism so that it does not perform the exchange atomically. So we removed the `atomic` keyword from the beginning of our exchange inline function and run the verification. In this verification we used two worker threads, `f` initialized to 3 and task stacks of size 5. Spin was able to find a trace that violated the "StolenOrInlined" property. The table 5.3 shows Spins's verification results. Intuitively what happens here is the owner thread reads the `f` of a task in its stack before synchronizing with it. But before invalidating task's `f`, it gets delayed. A thief thread reads the same value of `f` and since it is still valid it continues with first invalidating the `f` and then stealing the task. Finally the owner resumes and although the `f` is invalidated by the thief and the task is not there anymore, since it has a valid copy of `f`, it continues with synchronization which causes violation of property.

Time(s)	Used memory(MB)	Vector size(Byte)	State stored+Matched	Search Depth
32	16.53	663	14765044	782

TABLE 5.3: Verification results when model violates "StolenOrInlined"

We changed back the exchange inline function to its previous state by adding the `atomic` keyword and then we verified the model against the two correctness properties. Spin was able to verify the model with two worker processes and initial value `f` up to 3. But for `f` value of 4 Spin was not able to verify exhaustively. Table 5.3 shows output of spin verification for `f`=3.

Time(s)	Used memory(MB)	Vector size(Byte)	State stored+Matched	Search Depth
31.8	111.087	543	3717192	615

TABLE 5.4: Verification results when model preserves "StolenOrInlined" and "StolenOnce"

Chapter 6

Conclusion and Future Works

Wool is a parallel library which uses a novel work-stealing algorithm [9] and achieves low spawning overhead and reasonable stealing cost. Our work in this project was a first attempt towards verification of the Wool against a number of correctness properties. We mainly focused on the verification of its underlying work-stealing algorithm. The first property was "NoMiss" which checks there is no unstolen tasks left below the `bot` pointer of the worker thread. As we already explained in chapter 5, the pointer is always pointing to the oldest stealable task. If a task is stealable but it is remained below the `bot` pointer it means that task is the oldest stealable task while the `bot` is not pointing to it and hence the synchronization mechanism of Wool becomes invalidated. Our verification showed that for a small and simple test harness that is capable of violating the property, two worker threads and task's function(`f`) initiated to 3, Wool's simple protection mechanism works fine and avoids violation. For the "StolenOnce" and "StolenOrInlined" correctness properties which are stating that a task should not be stolen twice or should not be both stolen and inlined, we added a counter field `s` to task data structure and then using this extra field and a monitor process we were able to verify the model against these properties and we didn't get any error report from Spin.

Wool is a library written in C code with different options and alternative branches implemented inside its code to give programmers and users the power to choose different optimizations based on hardware architectures they are using or other factors. In this work we mainly focused on synchronization methods that Wool uses based on hardware architectures. Among the two methods Wool offer namely THE and EXCHANGE, we decided to model the latter one because it not using any locking mechanism for synchronizing between the working threads. Therefor it was interesting to verify the method to make sure that it is still synchronizing properly despite it is not using any lock. There were other optimization options in the Wool code that depending on whether they are active or not, the library would act differently. We assumed default values for these options. So what we did in our project is the verification of the Wool library with respect to the EXCHANGE synchronization and its default options. There are possibilities to extend the current model so that it involves the other synchronization method(THE) as well as the other optimization options it utilizes. Since Wool is using a work-stealing deque as the basic data-structure, there is also room for automatic verification of this library against general functional properties of a work-stealing deque. Furthermore our verification is using a small model with only two number of worker threads, but one can extend this verification to an unbounded number of threads. Finally in our model we use a task array of maximum size 6, but one can extend the verification to verify task arrays with unbounded size.

Bibliography

[1] D. Andrade, B.B. Fraguela, J. Brodman, and D. Padua. Task-parallel versus data-parallel library-based programming in multicore systems. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 101 –110, feb. 2009.

[2] N.S. Arora, R.D. Blumofe, and C.G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129. ACM, 1998.

[3] T. Ball, B. Cook, V. Levin, and S.K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods*, pages 1–20. Springer, 2004.

[4] M. Ben-Ari. *Principles of the Spin model checker*. Springer-Verlag New York Inc, 2008.

[5] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, B. Berard, M. Bidoit, and A. Finkel. *Systems and software verification*, volume 204. Springer, 2001.

[6] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. 1994.

[7] E. Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.

[8] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131, 1981.

[9] Karl-Filip Faxén. Wool-a work stealing library. *SIGARCH Comput. Archit. News*, 36:93–100, June 2009.

[10] K.F. Faxén. Wool-a work stealing library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, 2009.

[11] G.J. Holzmann. An improved protocol reachability analysis technique. *Software: Practice and Experience*, 18(2):137–161, 1988.

[12] G.J. Holzmann. An analysis of bitstate hashing. *Formal methods in system design*, 13(3):289–307, 1998.

[13] G.J. Holzmann. *The SPIN model checker: Primer and reference manual.* Addison Wesley Publishing Company, 2004.

[14] K. Jiang. Model Checking C Programs by Translating C to Promela. 2009.

[15] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 267–280. USENIX Association, 2008.

[16] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer Aided Verification*, pages 377–390. Springer, 1994.

[17] D. Peled. *Software reliability methods.* Springer Verlag, 2001.

[18] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[19] S. Soleimanifard. Generating a Model of a Communication Protocol from Test Data. 2009.

[20] D.B. Wagner and B.G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *ACM SIGPLAN Notices*, volume 28, pages 208–217. ACM, 1993.

[21] Wikipedia. Fibonacci number — wikipedia, the free encyclopedia, 2011. [Online; accessed 19-May-2011].

[22] Wikipedia. Task parallelism — wikipedia, the free encyclopedia, 2011. [Online; accessed 18-April-2011].