The Use of Model-Checking for the Verification of Concurrent Algorithms

Joseph Cordina

Department of Computer Science and AI,
University of Malta

Abstract. The design of concurrent algorithms tends to be a long and difficult process. Increasing the number of concurrent entities to realistic numbers makes manual verification of these algorithms almost impossible. Designers normally resort to running these algorithms exhaustively yet can never be guaranteed of their correctness. In this report, we propose the use of a model-checker (SMV) as a machine-automated tool for the verification of these algorithms. We present methods how this tool can be used to encode algorithms and allow properties to be guaranteed for uni-processor machines running a scheduler or SMP machines. We also present a language-generator allowing the designer to use a description language that is then automatically converted to the model-checker's native language. We show how this approach was successful in encoding a concurrent algorithm and is able to verify the desired properties.

1 Introduction

Designing any new algorithm tends to be a strenuous and tough mental exercise. This is especially more difficult when designing concurrent algorithms due to their complex interaction. Commonly any such algorithm is painstakingly designed and is then dry-run to test for errors. Such errors normally tested for are deadlocks, memory value inconsistencies, etc. Dry-runs for concurrent algorithms involve testing each line of code when run in parallel with any other line of code in its concurrent counterpart. This ensures that no race-conditions could arise. Understandably this is a very long and difficult task especially since the problem grows exponentially with the number of concurrent tasks present within the system. These algorithms are normally tested in their assembly format, ensuring the atomicity of each instruction but making it more difficult for the designer to reason with. Often algorithms more than few lines long and running on more than a handful of concurrent tasks quickly become impossible to verify manually and thus designers tend to exhaustively execute them on hardware. While testing on hardware is seen as sufficient by most system software designers, the lack of certainty of the absence of errors makes such algorithms unsuitable for critical software. While novel concurrent algorithms can enhance the potential of computer hardware and software, the lack of proper execution guarantees makes work on such algorithms futile.

In this paper we propose the use of model-checking to facilitate and provide guarantees for the construction of concurrent algorithms. Such concurrent algorithms are not limited to be used on single-processor systems (making use of an un-deterministic scheduler) but also on synchronous and symmetric multi-processor algorithms. We also envisage this research to encompass un-synchronous multi-processor systems. In addition, we propose a meta-language, allowing the designer of the algorithms to concentrate on the construction of the algorithms itself and also widen the scope of this research to users that are not familiar with model-checking system.

2 Model-Checking

The term *Model-Checking* is very adequately described by Clarke and Emerson as '... an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for that model' [2]. In fact most modern model-checking systems are able to build a finite-state model of the system given a description language. In addition, they are then able to verify certain properties of the model using the same or some other description language. A brute-force model-checker merely traverses the state table for the possibility of ending up in a state that contradicts the property in hand. Yet the number of states quickly become too large for any hardware to handle. Thus research in model checkers aims to provide means and ways to verify properties while keeping the number of states small.

Research of the application of model-checking systems for concurrent algorithms is common. One interesting example is the work by Bauer et al. [6] that makes use of PVS [9] for the verification of concurrent algorithms when applied to dynamic memory. There is also a large number of model-checkers each suited for a particular verification domain. Very popular systems include Lustre [10] and Esterel [1], synchronous declarative languages for programming reactive systems; SPIN [5], a tool best suited for analysing data communication protocols and many others.

One of the reasons behind our work is the application of such model-checkers for the verification of algorithms that were designed within the Department of Computer Science and A.I. namely the wait-free algorithms proposed by Debattista [4] and Vella [13]. We would like to extend this research to be applied for CSP communication algorithms proposed by Vella [14, 12] yet this would require a different model-check system than we are concentrating upon here. For these reasons we have made use of SMV [7], a tool for checking finite state systems against specifications in the temporal logic CTL. It makes use of the OBDD-based symbolic model checking algorithm to control the state-explosion problem [8]. The SMV language was found ideal to verify concurrent algorithms since its very similar to imperative programming languages. In addition, the use of CTL temporal logic lends itself naturally to the specification of properties of these algorithms. SMV also has freely-available visual tools for both Windows and Linux systems making is easy to use. After parsing a specification, SMV will verify certain properties on demand and is able to present a counter-example to negated properties.

In our implementation we assume that the user will approach the system with a concurrent algorithm that has been reduced to its assembly equivalent[1]. The algorithm can be made up of different parts that can be executed separately. We have built two specifications in SMV that are able to verify concurrent algorithms. One of these is able to verify concurrent algorithms that are executed on a single processor system whereby a scheduler schedules instances of the algorithm in an undeterministic format. The second system specifies a multi-processor system that executes instances of the algorithm on each processor. Note that running the same algorithm on these systems is not guaranteed to have the same behaviour. One can envisage a scenario whereby two instances of one algorithm are run, whereby the algorithm takes a shared resource and releases it. On a fair multi-processor system, deadlock will never happen while in a single processor system, the first instance might never be de-scheduled when it does not have control of this resource. In such a case the second instance will be starved of the resource completely. We envisage these two systems to be integrated together thus allowing schedulable tasks on multi-processor systems.

[1] Most concurrent algorithms are normally designed in their assembly format anyway, otherwise one can simple use the assembler output of the original code.

2.1 Single Processor System

In this system, we have defined an SMV specification, such that each instance of a concurrent algorithm is executed within a task. Multiple tasks are then scheduled on the same CPU using a scheduler. In SMV, we have created the definition of a *task* which is made up of a task control block containing the CPU registers, the program counter and other CPU flags. In addition, we specified the list of tasks as an array of control blocks. We specified an input to the system that specifies which task to start executing. Every time the input changes, the previous state is stored within the appropriate task control block and the next one is started. To emulate the execution of instructions, we have hard-coded the end result of each instruction within the algorithm. Thus basically the current instruction that is executed (specified by the value of the current instruction pointer) will affect the next state in terms of Program Counter, registers, flags and other external memory locations.

Using such a system, since the input can change un-deterministically, SMV will verify all combinations of execution of each instruction. We also assumed that no more than one instruction can be executed at any one time, thus allowing a deterministic value for shared variables in the next state. A user of the system can encode concurrent algorithms that contain different independent parts as separate tasks.

2.2 SMP Systems

Using SMV, we have specified a symmetric multi-processor system that is able to execute instructions concurrently on each CPU. Unlike the single-processor system, we have not included a scheduling entity but we assume that once an algorithm starts, it will continue executing to termination (if it terminates). In this system, we have implemented the multi-processor system by specifying a series of registers and flags for each CPU that are independent of each other and that can change concurrently. To simulate the execution of the algorithm, we have specified an input trigger for each CPU and for each independent code section of the algorithm. This trigger will start a specific algorithm on one of the CPUs being used in the verification[2]. Using this system, we ensure that the start of the algorithm on a particular CPU is un-deterministic with respect to other CPUs. Thus SMV will verify all permutations of execution of the algorithm. Likewise to the uni-processor system, we have hard-coded each instruction so that when a particular CPU's program counter reaches a certain value, the next state of the registers and memory locations is determined. In our implementation we have assumed that all processors are synchronised in their execution cycles. In other words, one instruction on one CPU will take the same amount of time as another instruction on another CPU and not more than one instruction can execute within one clock cycle. While this is not strictly true in SMP systems, we felt confident in applying this assumption since deadlocks and memory consistency problems only occur when algorithms share data and on SMP systems access to shared variables is always synchronised (normally through the use of a *lock* instruction). Creating a system that is extensible to non-symmetric multi-processor systems would require removing this assumption, something that would not require major changes in our implementation.

The SMP specification has presented certain difficulties in implementing shared variables since one needs to guarantee the value of a shared memory location in the next clock cycle. We have tackled this situation by checking which instruction each CPU is executing and verifying what these instructions are. If only one instruction is modifying a shared memory value, then the value is deterministic. Otherwise we specify an un-deterministic value (from a range) to the shared

[2] If the trigger is enabled while the CPU is executing some code, the trigger is ignored.

memory location, allowing SMV to verify the algorithm for all possible values. Note that concurrent algorithms should ensure that values are always deterministic, and SMV clearly shows where these restrictions have to be applied.

3 Pre-Processing

While SMV presents a language that is easily-understandable, it can quickly become a daunting task to re-write complex algorithms using this language. To aid in the process of verification, we have created a description language that would enable the user to specify a concurrent algorithm which is then used to create an SMV specification.

Implementing several common concurrent algorithms in SMV, we have pinpointed several elements that such a description should contain. Apart from the algorithm itself, written in assembly format, the user should be able to provide general assumptions for the model in addition to properties that need to be proven. These assumptions and properties should be stated in temporal logic, a simple yet powerful construct. Additionally, we have provided the facility to create certain logical assertions depending on the algorithm's line number, thus enabling logical properties to be based on these assertions. We have made use of FLEX [11] to generate a parser and scanner for the algorithm description that is then able to generate a specification in SMV. Figure 1 shows the data flow in the process of verification. We envisage to create a parser for the SMV counter example output, allowing visual representation of the execution trace leading to the occurrence of the false assertion of a particular property.

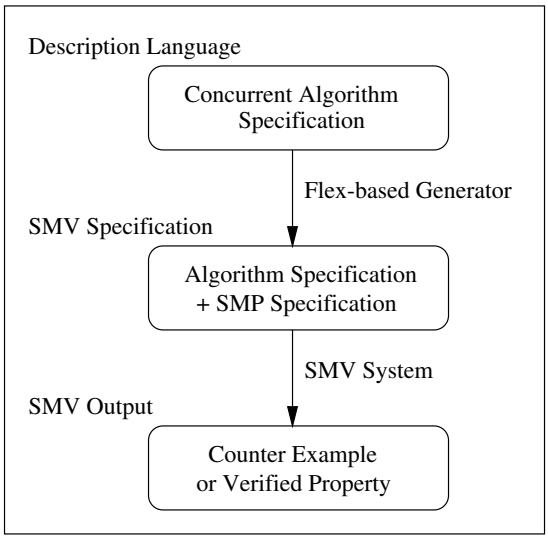

Fig. 1. Data Flow in the Verification Process

3.1 Results

What follows is a sample description file showing all characteristics making up an algorithm description together with all properties to be proven. This example is targeted to be used for SMP

systems. In this description we have specified a trivial yet valid concurrent algorithm, namely the *basic spin-lock algorithm with cache-invalidation* [3].

```
CPU 2

REGISTERS
carry:boolean;

MEMORY
lock:boolean;

STATEVARS
lockacquired:boolean;

LABELS
LABEL1 1
LABEL2 4
LABEL3 5

CODE
btsl lock
jne LABEL2
jmp LABEL1
nop

ASSERTIONS
4:lockacquired:=1;

TRIGGER
getlock:boolean;

CODE
btrl lock
nop

ASSERTIONS
2:lockacquired:=0;

TRIGGER
releaselock:boolean;

START
lock:=0;
lockacquired:=0;

PERCPUASSUMPTIONS
norelease: G(releaselock->lockacquired)

EXCLUSIVEINSTRUCTIONS
LABEL1
LABEL3
```

```
PROVE
mutex: G ~(&lockacquired[x]);
```

In the above description one can see how internal registers are specified (*carry*), external memory locations that might be shared (*lock*) and also the two code sections marking the acquire and the release stage of the shared lock. In addition labels used within the code are specified using a global line numbering starting from line number 1. One can also create state variables (*lockacquired*) stating those properties one wants to assert or check upon. For each code segment, one also needs to specify the variable that is used to trigger each particular code segment (*getlock* and *releaselock*). These variables are used as the input to the SMV module, thus enabling a non-deterministic execution start of the algorithm. In addition one can specify assertions for each code segment making use of the state variables specified previously. These assertions are executed upon reaching the given line number.

After specifying the algorithm itself, one can specify any initial values to variables using the `START` tag. There are also two forms of assumptions one can provide to the model. There are assumptions that are specific to each CPU, in other words, they are assumptions that are specific to the code itself (in our case we specify that the code corresponding to releasing the lock is only executed if one owns the lock[3]). In addition, one can specify interactions between different CPUs (using the `EXCLUSIVEINSTRUCTIONS` tag). Since the only interaction between the different CPUs on standard SMP systems is specifying those instructions that cannot access memory concurrently, we have opted to allow the user to specify labels indicating instruction position within the code. Consequently, SMV will ensure that instructions specified here will not be allowed to executed concurrently, but will introduce a one time step delay when this happens. Finally one can specify the properties that needs to be proven. We have introduced a special syntax allowing proofs to vary over particular CPUs or all of them. In our example we state that *lockacquired[x]* is expanded to the AND of all *lockacquired[x]* where x varies over all CPU indices.

In our example it turns out that *mutex* can be verified in SMV using 14 state variables with 4 combinatorial variables for 2 CPUs. This rises to 21 state variables with 6 combinatorial variables for 3 CPUs. While the number of state variables rapidly increases with the number of CPUs making verification difficult for a large number of CPUs, at least we do have some guarantee properties to work with for a small number of CPUs. To have complete confidence that the verification applies to n CPUs one can then apply some other technique such as proof by induction to arrive to a satisfactory verification.

While the example given is quite trivial, much more complex algorithms can be verified using our tool. All that is required to verify wait-free algorithms is to encode within the FLEX specification the novel instructions that are used within these algorithms. Then any algorithm can be specified and quickly verified given that the state space is not huge[4].

4 Conclusion

In this report we have highlighted the difficulties normally encountered by designers of concurrent algorithms to ensure the correctness of these algorithms. We have pinpointed a tool (SMV) that is

[3] While it is normally assumed that one never calls release unless one has the lock, SMV quickly picks up on the gross negligence in this assumption and shows how this simple algorithm fails when this is not assumed.

[4] Normally concurrent algorithms are quite short in the number of instructions, thus ensuring a limited state space and thus verification can be completed within a reasonable amount of time within SMV.

able to verify these algorithms and we have constructed an implementation that would satisfactorily verify these algorithms for both uni-processor machines running a non-deterministic scheduler and a multi-processor machine. In addition we have presented a description language using which a designer can specify his algorithms without having to resort to the use of the SMV language. Using our parser this description is automatically translated to its SMV counterpart which in its turn is directly given to the SMV system.

While this work is still in its infancy and many more improvements can be applied, we are confident that using these ideas, many complex concurrent algorithms can be verified and guarantees reasonably given on these algorithms.

References

1. Gérard Berry. The foundations of esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte*. MIT Press, 1998.
2. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In *Logic of Programs, Volume 131 of Lecture Notes in Conputer Science*, pages 52–71. Springer-Verlag, 1981.
3. J. Cordina. Fast multithreading on shared memory multiprocessors. B.Sc. I.T. Final Year Project, Department of Computer Science and Artificial Intelligence, University of Malta, June 2000.
4. K. Debattista. High performance thread scheduling on shared memory multiprocessors. Master's thesis, University of Malta, February 2001.
5. Gerald J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison Wesley Publishing Company, Sep 2003.
6. R. Nickson K. Bauer, L. Groves and M. Moir. Machine-assisted reasoning about concurrency and dynamic memory. Victoria University of Wellington, School of Mathematical and Computing Sciences, December 2001.
7. K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem*. PhD thesis, SCS, Carnegie Mellon University, 1992.
8. K.L. McMillan. The SMV system. Cernegie Mellon University, November 2000.
9. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, Jun 1992. Springer-Verlag.
10. N. Halbwachs P. Caspi, D. Pilaud and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Conf. on Principles of Programming Languages.*, Munich, January 1987.
11. Vern Paxson. Flex, version 2.5. URL http://www.gnu.org/software/flex/.
12. K. Vella. CSP/occam on networks of workstations. In C.R. Jesshope and A.V. Shafarenko, editors, *Proceedings of UK Parallel '96: The BCS Parallel Processing Specialist Group Annual Conference*, pages 70–91. Springer-Verlag, July 1996.
13. K. Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, University of Kent at Canterbury, December 1998.
14. K. Vella and P.H. Welch. CSP/occam on shared memory multiprocessor workstations. In B.M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering*, pages 87–120. IOS Press, April 1999.