A Practical approach on Model checking with Modex and Spin

Muhammad Iqbal Hossain #1, Nahida Sultana Chowdhury *2

# Computer Science and Engineering, Kyungpook National University
Daegu, South Korea
1milon_mi7@yahoo.com

* Computer Science and Engineering, Kyungpook National University
Daegu, South Korea
2nahida_uap@yahoo.com

Abstract— Software verification methods are used widely in commercial and industrial development. To formally verify a large software application we can extract a model from the source code and verify that model with any verification mechanism or human contribution. So there are two things extraction and verification. There are two main problems in this approach. The first problem is verification process by human is no longer reliable because in manual verification the error rate can be higher and that can decrease the functionalities of the whole process. The second problem is the effort to extract or verify the model can take a considerable investment of time and expertise. For a large system manual verification can take a great effort and the performance of humans.

In this paper we have tried to represent the whole process in an automatic way. And also we have tried to ignore the manual contribution of human to make the whole procedure faster and reliable. For this purpose we choose an automated model extractor method: Modex and an automated verifier to extrac model: Spin. Firstly we have applied MODEX to automated the model extractor method and then we used SPIN to automatically verify the extracted model.

Index Term — MODEX, SPIN, Model Extraction, Model Verification

I. INTRODUCTION

There are many techniques for evaluating code quality and for reducing the expected number of residual bugs, manual code walkthrough, and peer review and of course plain old unit and integration testing. There is no question that these methods work, they catch bugs but they are not perfect. Conventional verification technique has some restriction that limit the number of bugs that can be caught in this way. This is sometimes called the ceiling that is imposed by standard verifying technique. To verify a piece of code either in isolation or in a given context, a verifier usually creates a test harness, special instrument that is used to efficiently and reproducibly administer a series of tests and evaluate the result[1].

In this paper we divide the whole procedure in two steps: model extractor and model verification. Models are widely used in most engineering disciplines. A model can be a mathematical theory, a physical entity an image or shape, a mere guiding mental image in the mind of a designer. The purpose of a model is to facilitate analysis, either explicitly or implicitly. Models are abstractions of real world artifacts. A designer only takes the trouble to build a model when it is easier, cheaper, or faster to analyse the model than it is to analyse the real world artifact itself. A model that is more complex than the artifact that it describes would be comparable to the summary of a book that is longer than the book. The effective use of abstraction is the key to the successful construction of models. The construction of an abstract model, in turn, is the key to successful analysis. A model should not only be simpler than the artifact being modeled, it should also be simpler to construct than that artifact. If the construction of a model is perceived to be too complex, it will be all too tempting to forgo the construction of the model, and make do with more basic types of analysis [2].
everything is meant to work. The documentation gives a good head-start. Experience with the application of the tools, building test harnesses, and verifying applications, will do the rest. Experience with the effective use of these tools comes slowly but surely, so be patient! After these somewhat discouraging notes, it is also good to add that the reward for finally mastering the use of these tools can be very substantial. Our own experience to date indicates that these tools, if used right, allow us to verify software applications, or portions thereof, with a thoroughness that cannot be achieved by any other testing or verification method [3, 5]. The procedure to run modex has shown below:

Spin is a model checker [2] that has been developed at Bell Labs in the original UNIX group of the computer Science research centre. It is originally designed for verifying communications protocols and later it become most widely used verification tools. Spin is particularly suited for modeling concurrent and distributed systems that are based upon interleaving of atomic instruction. Models, written in simple language called Promela, can be simulated randomly or interactively [6]. Spin can generate efficient verifiers that search for a counter example to correctness specification applied to a model. Unlike many model-checkers, SPIN does not actually perform model checking itself; instead, it generates a highly optimized model checking program in C for each model and each correctness claim to be verified. The user need not look at this C code, although to use Spin you need to have a C compiler installed. A verification in Spin can be run with a three-line script (generate the verifier, compile it and run it); alternatively, a development environment can be used. We can generate different kind of model to better understand the flow of datas, procedures, error

Spin can be run in four modes [6]:

1. Random simulation mode uses a random number generator to resolve the non-determinism inherent in a concurrent program as well as the possible non-determinism in the guarded commands of a single process.
2. Interactive simulation mode enables the user to choose the next instruction to be executed. Interactive simulation is also supported in concurrency simulators and is essential for demonstrating scenarios (such as those for starvation or fairness) that are very unlikely to occur randomly.
3. In verification mode, Spin systematically searches the entire state space looking for a counter example, a computation that violates a correctness specification.
4. If a counterexample is found, a trail of the incorrect computation can be used in guided simulation mode to recreate the computation to the user to examine. The verification process with SPIN has shown below

III. MODEL VERIFICATION: SPIN

Spin is written in C and is distributed as a single executable file for Windows and Linux. One way that Spin achieves efficiency is that it does not perform the model checking itself; instead, it generates a highly optimized model checking program in C for each model and each correctness claim to be verified. The user need not look at this C code, although to use Spin you need to have a C compiler installed. A verification in Spin can be run with a three-line script (generate the verifier, compile it and run it); alternatively, a development environment can be used. We can generate different kind of model to better understand the flow of datas, procedures, error

Spin can be run in four modes [6]:

1. Random simulation mode uses a random number generator to resolve the non-determinism inherent in a concurrent program as well as the possible non-determinism in the guarded commands of a single process.
2. Interactive simulation mode enables the user to choose the next instruction to be executed. Interactive simulation is also supported in concurrency simulators and is essential for demonstrating scenarios (such as those for starvation or fairness) that are very unlikely to occur randomly.
3. In verification mode, Spin systematically searches the entire state space looking for a counter example, a computation that violates a correctness specification.
4. If a counterexample is found, a trail of the incorrect computation can be used in guided simulation mode to recreate the computation to the user to examine. The verification process with SPIN has shown below

IV. IMPLEMENTATION AND RESULT

To implement our procedure, we have to write a C code and that will be verified through modex and spin. At the first step we have to extract a model from the existing or newly created C code. The basic idea of model extraction is converting ANSI C code to promela code. The branching
instructions are somewhat equivalent to promela. In this extraction procedures are not really supported. We can use inlining or link to external procedure, by the conversion table we can generate the basic statements. A test harness[7] is created associated with the C code. Test harness is used to specify the details of system verification. All the test harness specifications are stored in a single file, with the suffix .prx, grouped into different commands, each dealing with different aspect to the verification setup.

Figure 4 represents the C code of the program. This is a simple mutual exclusion code. We attempt to extract a model from this code by the help of modex

```c
int x, y, z;
void
lock(int Pid)
{
    busywait:
    x = Pid;
    if (x != Pid)
        goto busywait;
    y = Pid;
    if (y != Pid)
        goto busywait;
    z = Pid;
    if (z != Pid)
        goto busywait;
}
void unlock()
{
    x = 0;
    y = 0;
    z = 0;
}
```

Figure 4. C code of mutual exclusion

Figure 5 represented the extracted model from the C code using modex. The test harness file also executed automatically during the model extraction. We will not discuss about the test harness file (.prx) to generate the model there are some specific command:

```bash
modex mutex.c

cpp -E -P _modex_.run > model

cat model
```

Now we will use spin for the verification of the model extracted by mutex. Figure 6, 7 and 8 show different type of
verification model generated by Spin. When a PROMELA model contains embedded C code, SPIN cannot simulate its execution in the normal way because it cannot directly interpret the embedded code fragments. If we try to run a simulation anyway, SPIN will make a best effort to comply, but it will only print the text of the c_expr and c_code fragments that it encounters, without actually executing them. To faithfully execute all embedded C code fragments, we must first generate the pan files and compile them. We now rely on the standard C compiler to interpret the contents of all embedded code as part of the normal compilation process.

$ spin -a model
$ cc -o pan pan.c
$ ./pan

hint: this search is more efficient if pan.c is compiled

(Spin Version 6.1.0 -- 4 May 2011)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim    - (none specified)
assertion violations +
acceptance cycles - (not selected)
invalid end states +

State-vector 64 byte, depth reached 34, errors: 1
58 states, stored
19 states, matched
77 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.004 equivalent memory usage for states
(stored*(State-vector + overhead))
0.289 actual memory usage for states
(unsuccessful compression: 6530.00%) state-vector as stored = 5208 byte + 16 byte overhead
2.000 memory used for hash table (-w19)
0.343 memory used for DFS stack (-m10000)
2.539 total actual memory usage

pan: elapsed time 0 seconds

Fig. 6. Creating pan through spin

$ ./pan -r

1: proc 1 (user) model:34 (state 1) [run lock(_pid,q)]
2: proc 0 (user) model:34 (state 1) [run lock(_pid,q)]
3: proc 3 (lock) model:11 (state 1) [now.x=Lock->Pid; ]
4: proc 3 (lock) model:13 (state 5) [else]
5: proc 3 (lock) model:17 (state 7) [now.z=Plock->Pid; ]
6: proc 3 (lock) model:19 (state 11) [else]
7: proc 3 (lock) model:23 (state 13) [now.y=Plock->Pid; ]
8: proc 3 (lock) model:25 (state 17) [else]
9: proc 3 (lock) model:30 (state 19) [q0]
10: proc 0 (user) model:34 (state 2) [q0]
11: proc 3 (lock) :0 (state 0) [ -end- ]
12: proc 2 (lock) model:11 (state 1) [now.x=Plock->Pid; ]
13: proc 2 (lock) model:13 (state 5) [else]
14: proc 2 (lock) model:17 (state 7) [now.z=Plock->Pid; ]
15: proc 2 (lock) model:19 (state 11) [else]
16: proc 2 (lock) model:23 (state 13) [now.y=Plock->Pid; ]
17: proc 2 (lock) model:25 (state 17) [else]
18: proc 2 (lock) model:30 (state 19) [q0]
19: proc 1 (user) model:34 (state 2) [q0]
20: proc 2 (lock) :0 (state 0) [ -end- ]
21: proc 1 (user) model:35 (state 3) [cnt = (cnt+1)]
22: proc 1 (user) model:36 (state 4) [assert((cnt==1))]
23: proc 0 (user) model:35 (state 3) [cnt = (cnt+1)]
24: proc 1 (user) model:37 (state 5) (invalid end state) [assert((cnt==1))]

code1: { now.x=Plock->Pid; }

code3: { now.y=Lock->Pid; }

code5: { now.y=Lock->Pid; }

Fig. 7. output generated by ./pan -r

$ spin -p model
Starting lock with pid 2
1: proc 1 (user) model:34 (state 1) [run lock(_pid,q)]
Starting lock with pid 3
2: proc 0 (user) model:34 (state 1) [run lock(_pid,q)]

code1: { now.x=Plock->Pid; }

code3: { now.y=Lock->Pid; }

code5: { now.y=Lock->Pid; }

110105-8989 IJECS-IJENS © October 2011 IJENS
10: proc 0 (user) model:34 (state 2) [q’0]
11: proc 3 terminates
c_code1: { now.x=Plock->Pid; }
12: proc 2 (lock) model:11 (state 1) [[c_code1]]
13: proc 2 (lock) model:15 (state 4) [else]
c_code3: { now.y=Plock->Pid; }
14: proc 2 (lock) model:17 (state 7) [[c_code3]]
15: proc 2 (lock) model:21 (state 10) [else]
c_code5: { now.z=Plock->Pid; }
16: proc 2 (lock) model:23 (state 13) [[c_code5]]
17: proc 2 (lock) model:27 (state 16) [else]
18: proc 2 (lock) model:30 (state 19) [q’0]
19: proc 1 (user) model:34 (state 2) [q’0]
20: proc 2 terminates
21: proc 1 (user) model:35 (state 3) [cnt = (cnt+1)]
22: proc 1 (user) model:36 (state 4) [assert((cnt==1))]
23: proc 0 (user) model:35 (state 3) [cnt = (cnt+1)]
24: proc 0 (user) model:36 (state 4) [assert((cnt==1))]
spin: text of failed assertion: assert((cnt==1))
24: spin: trail ends after 24 steps
#processes: 2
x = 0
y = 0
z = 0
_w64 = 0
__builtin_va_list = 0
cnt = 2
24: proc 1 (user) model:37 (state 5)
24: proc 0 (user) model:37 (state 5)
4 processes created.

In figure 8, the assertion is violated at the end, but this is merely because it was hardwired to fail. None of the C data objects referenced were ever created during this run, and thus none of them had any values that were effectively assigned to them at the end. Note also that the text of the c_code fragment that is numbered c_code5 here is printed out, but that the print statement that it contains is not itself executed, or else the values printed would have shown up in the output near this line.

V. CONCLUSIONS

The method we have pronounced for verifying reactive software applications combines data abstraction with implementation level verification. The user provides a test harness, written in the language of the model checker, that non-deterministically selects inputs for the application that can drive it through all its relevant states. Correctness properties can be verified in the usual way: by making logical statements about reachable and unreachable states, or about feasible or infeasible executions. The state tracking capability allows us to perform full temporal logic verification on implementation level code.

Model checking is a formal method that can facilitate learning important CS concepts like concurrency, verification, and nodeterminism. Just as importantly, model checking can motivate the study of discrete mathematics and theoretical computer science by showing how they are used in a real-world application: automata theory, logic, graph theory, hashing functions, and data compression. Although Spin is a professional software tool widely used in industry, the simple and clean syntax and semantics of Promela, and the ease of running simulations and verifications make it ideal as a teaching tool, and there are some other tools such as isp in, jSpin and Erigone to simplify the use of model checking even further.

In MODEX, still there have some limitations present in the translation process from C to Promela code. So, we need to consider this issue more carefully to make it more reliable and efficient for SPIN model checker.

REFERENCES

[7] An automated verification method for distributed systems software based of model extraction by Gerard J. Holzmann and Margaret H. Smith, IEEE transaction on software Engineering Vol. 28, No. 4 April 2002