University of Toronto
CSC2108S – Automated Verification

A Case Study of Simple Mine System With PROMELA/SPIN

Su Rong
surong@control.toronto.edu

Yuwei Wang
wangyw@control.toronto.edu

Abstract

In this project we developed a model for a simple mine system. The main objective is to model the Controller of the mine system and then check the properties by using the PROMELA/SPIN framework. We first introduce the mine system , then give the corresponding Controller which can guarantee the pre-specified requirements and discuss the final results we've got. We also describe some experiences and draw our conclusions.

Contents

1. **Introduction……..3**

2. **Description of Simple Mine System(SMS)……3**
   2.1 General Description……...3
   2.2 Explanation of Physical Structure in SMS……..……..…6

3. **Modeling of SMS…….9**
   3.1 Basic Modules of SMS……..9
   3.2 Modeling of Sub-Modules of SMS by Using SPIN……10

4. **Specifications……..19**

5. **Properties and Results……...25**

6. **Discussion …….29**

7. **References……..32**

8. **Appendix**
   8.1 Source Code.
   8.2 Checking Results by SPIN.

1. Introduction.

In the course CSC2108 Automatic Verification, we have learned several formal methods such as SMV, PROMELA/SPIN, CWB and LP. These formal methods can help us to check the validity of our Controllers, algorithms, or proofs, etc.

Note that usually when we design some kind of a system, we use modular (decentralized) methods or hierarchical methods to construct local small models first, then combine them together to get the final centralized system. Due to current combining methods, the final centralized structure may be quite large. If we still expect to use manual methods for checking whether the centralized system satisfies the specifications or not, sometimes it is tedious or totally infeasible. So automatic verification methods are powerful tools to help us finish that tedious checking task.

In this paper, we use SPIN to verify a simple mine system[1]. SPIN is developed at Bell Labs. PROMELA, its input language, is a non-deterministic language, loosely based on Dijkstra's guarded command notation and hoare's language CSP. It allows to specify synchronous and asynchronous message passing via channels.

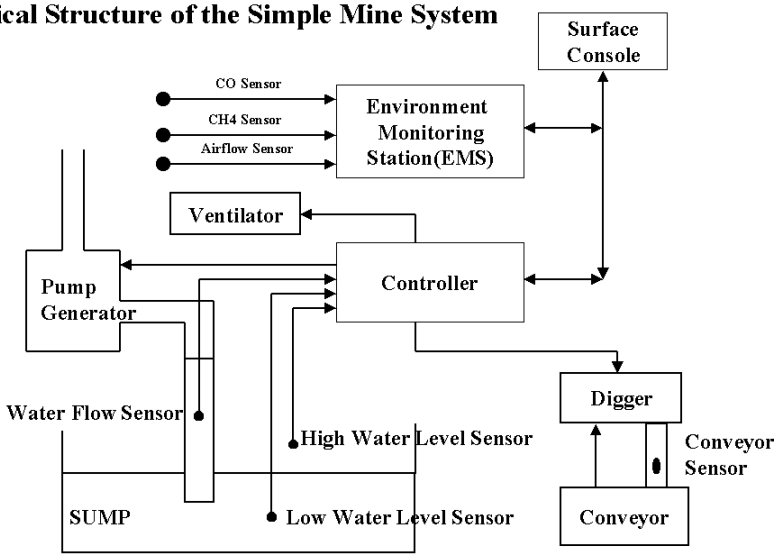
2. Description of Simple Mine System(SMS).

2.1 General Description.

Simple Mine System(SMS) is a simplified version of the real mine system. In this system, there are three levels of physical structures: Environment Monitoring station (EMS), System Controller, and Production Level.

The following is the system physical structure which indicates components of this system and their configurations and connections:

Physical Structure of the Simple Mine System



(1) EMS is used to monitor the quality of the atmosphere in the working environment. In the working environment, there are two dangerous gases: CH_4 and CO. Usually, CH_4 is much more dangerous than the concentration of CO, so in order to prevent fire, CH_4 concentration must be monitored and once it surpasses the normal value, some protection procedures should be taken. When the concentration of CO is not low, it may also be quite dangerous to the mine workers, so once its concentration surpasses a normal value, the system should sound an alarm. The air in the mine should always be kept fresh, which means that once the airflow is too small, the system should sound an alarm and take some recovery action, for example open the Ventilator. In the mine, there is another fatal factor – water. If the water level is too high, the mine will be drawn. This will be a disaster. So in order to prevent this problem, EMS should monitor the water level and send the corresponding information to the Controller and the console in time so that some protection actions will be taken. In this simple system, the Pump will be started to draw the water out.

(2) System Controller is used to analyze the information sent from EMS and take corresponding actions. Here we only use a centralized Controller, which handles all the information from EMS and the console and control all physical devices at the production level. It is the most important part in this simplified system.

(3) Production Level consists of three instruments: a Pump, a Digger, and a Ventilator. The Pump is used to draw water out of mine. The Digger is used to dig coal. The Ventilator is used to improve the quality of the atmosphere in the mine. Here in order to monitor the status of the Pump, a water-flow sensor is used. If no water flow is detected, then the system should sound an alarm to show that something wrong with the Pump. In order to move the ore out of the mine, a Conveyor is needed. Note that sometimes the actions of the Digger and the Conveyor may not synchronize well, so the coal may pile up on the Conveyor and maybe will stop the production line. In order to prevent this situation, a sensor is used to detect the piling of the coal on the Conveyor. If the coal is piled too high on the Conveyor, the Digger should be stopped for a while so that the coal can be moved from the Conveyor. This will make the Digger and the Conveyor work well simultaneously.

(4) Sometimes, in special circumstances, the operator's intervention is necessary, so a console is included in this system. At the console, an operator can monitor the whole system and send out a direct command to control the action of the Pump.

2.2 Explanation of Physical Structure in SMS.

- The Digger: the main working machine
- The Conveyor: conveying the ore

- The Pump generator: keeping the water level below the high level so that the Digger can work properly

- Environment monitoring station (EMS): monitoring the level of CO, CH_4 and airflow and reporting the value to the Controller and surface console.

- The Ventilator: when the level of CO and CH_4 is beyond the safe level, the Ventilator can restore the level of CO and CH4 to safe values quickly.

- Surface console: monitoring the whole system and sending commands to the pump generator and the digger directly when it is necessary.

- Sensors:

  CO sensor: reporting the value of CO

  CH_4 sensor: reporting the value of CH_4

  Air flow sensor: reporting the status of air flow

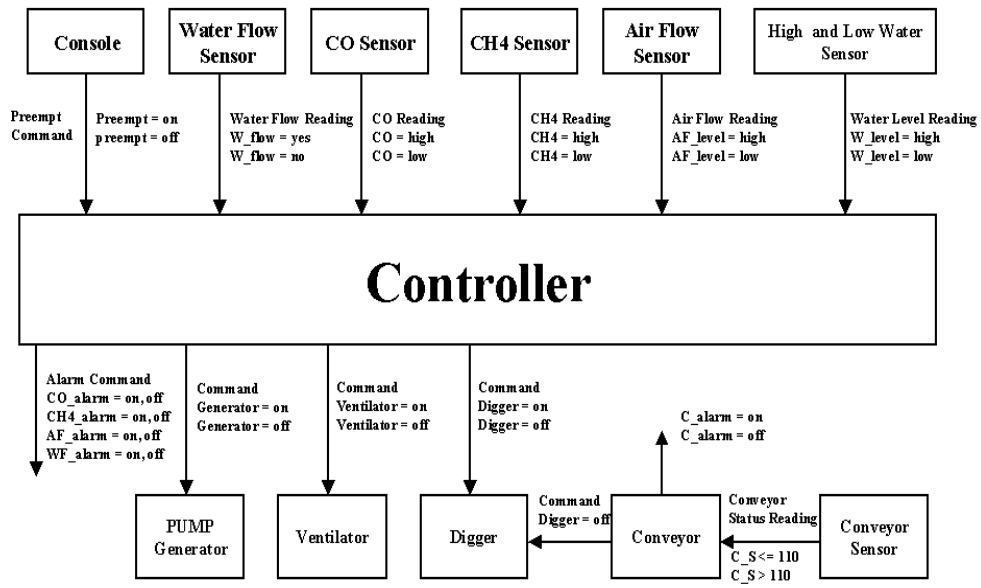
  The Conveyor sensor: reporting the value of the load on the Conveyor.

The main considerations for the mining system are as follows:

1. The Digger should keep working when the environment is safe. This will guarantee the efficiency of the whole production line.

2. The Pump generator and the Digger can only work when the environment is safe. This is the safety requirement.

Before we begin our modeling of this system, we use the diagram to show the signal flow in the system and this will make the following modeling much easier to be understood.

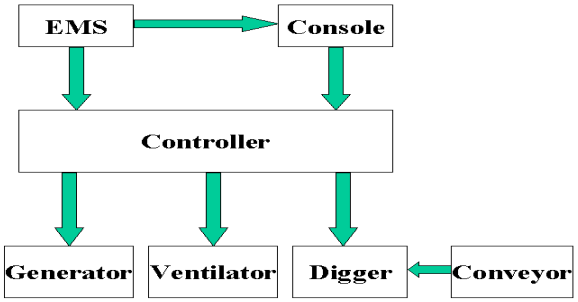


Signal Flow Chart of the Simple Mine System

3. Modeling of SMS.

3.1 Basic Modules of SMS.

From above descriptions, we model the mining system as follows:



In the above diagram, we assume that the Ventilator and the Digger will not break down, so once they receive a command, they will act immediately. The Digger accepts the command sent by the Controller and the Conveyor to synchronize their operations. The Pump may break down. We assume that once it does, the Controller will only sound the corresponding alarm, and wait for the operator to fix the problem. In the reality, once the operator receives this alarm, he will repair the Pump immediately if the

water level is not very high. Otherwise he will replace the broken Pump by a spare one. But in this paper, we only consider a simplified system. Since the Pump, the Ventilator and the Digger act only when required, we mainly focus on modeling the EMS, the console, the Conveyor and the Controller. And the Controller is the most important part.

3.2 Modeling the Sub-Modules of SMS in PROMELA.

(1) **Modeling EMS.**

In this part, since EMS only monitors the environment and sends out the corresponding detected information, we need to create random input signals which represent the environment information and use this random information to decide the output of EMS (or the output of sensors). From the CSC2108 course, we have learned how to create random signals.
Consider the following example:

```
proctype Water_level()
{
 do
 :: (W_level == 0) ->
     if
      :: 1 -> W_level = 1
      :: 1 -> W_level = 0
      fi
 :: ( W_level == 1) ->
      if
      :: (generator == 1) -> W_level = 0
```

```
      :: else -> skip
      fi
 od
}
```

This proctype is used to simulate the water-level sensor. When SPIN executes this module, it randomly selects a condition to execute, either W_level = 0, which means the water level is low, or W_level = 1, which means the water level is high. In this module, if the water level is low, then next time the water level can still be either low or high. So we use :

```
      if
      :: 1 -> W_level = 1
      :: 1 -> W_level = 0
      fi
```

to express this kind of random selection. But if the water level is high, then it will not go back to the low level automatically. Only when the Pump starts and draws the water out, can the water level go back to low again. Otherwise, the water level will remain high. We can use

```
      if
      :: (generator == 1) -> W_level = 0
      :: else -> skip
      fi
```

to express it. Generator = 1 means that the Pump has been started.

Here we assume that once the Pump starts, the water level will drop back immediately, without any time delay. This assumption will make the final code much simpler. Of course it is different from the reality. But it helps us to understand the role of the controller in the whole system.

Simulations of other sensors have similar structures except for the Conveyor Sensor. In the previous description, we have said that we use this sensor to detect the piling of ore on the Conveyor. Note that under the normal working condition, the Digger and the Conveyor should match quite well and no extra ore should pile on the Conveyor. But sometimes due to random factors, the Digger and the Conveyor may not match well, and the Digger may send more ore to the Conveyor than it can handle. We assume that under normal conditions, the piling of ore on the Conveyor is 100. And due to random factors, the piling may increase by 1 randomly. Once this extra piling occurs, the system will not recover automatically from it. We can use:

```
C_S = 100;
proctype Conveyor_sensor()
{
  do
  :: C_S = C_S + 1
  :: C_S = C_S + 0
  od
}
```

to express it. Here C_S = 100 means that under normal conditions, the output of the Conveyor_Sensor is 100. If we want to recover the Conveyor_Sensor value, we should stop the Digger first, so that the Conveyor will have time to move the extra ore out of this mine. Here, we still assume that no time delay is needed to move the extra ore out of the mine, similar to the assumption about the Pump.

(2) Modeling the Conveyor.

We model the Conveyor as follows:

```
proctype The Conveyor()
{
  do
  :: (C_S >= 110) -> Digger = 0;
                     C_S = 100;
                     C_alarm = 1
  :: (C_S < 110) ->  C_alarm = 0
  od
}
```

If the extra piling surpasses the upper bound, then the Conveyor will send command to the Digger to stop it, and sound an alarm. Note that once the Digger stops, the extra ore will be moved out immediately, so C_S will be reset immediately. If the extra piling has not surpassed the upper bound, then the Conveyor will do nothing except shutting off the alarm. Note that if no alarm, this

shut-off operation will still be done. Of course this will have no effect on the system status, except make modeling much simpler.

(3) Modeling the Console.

In the real system, the console will do two things: monitor the whole system, and control the operation of the Pump. The operator can send a command "preempt" to the Pump. This command has a higher priority than the command sent by the Controller caused by the water level. So that means that even if the water level is high, the operator still can stop the Pump from running. But for the safety reasons, the command "preempt" can not preempt the command caused by the higher CH_4 level because fire is much more dangerous than water. Thinking that the monitoring system will only read the current system status and write necessary information into database for the future retrieve without affecting the system behavior, so in our model, we don't think about this function of the Console, only think about the command function. We model the console as follows:

```
number = 1;
proctype Console()
{
  do
  ::  (number == 5) ->
                  if
                  :: 1 -> preempt =0; number =0
                  :: 1 -> preempt =0; number =0
                  :: 1 -> preempt =1; number =0
```

```
                    fi;
                    number = number +1
  ::   (number < 5) -> preempt =2; number = number +1
  od
}
```

Due to the rare preemption command sent by the operator, we assume that most of time the operator will not intervene with the operation of the Pump. We use a counter to count the time, and every fifth unit of time, the operator has a chance to intervene with the Pump. He/She may use this chance or not. This is a random event. So we use a random structure to realize it.

(4) Modeling of the Controller.

Now we will model the most important part – the Controller. Before we do it, we need to know the control objectives in this system and the relationship between the commands and the conditions.

(a) The control objectives are:
   (I) The system should always work in the safe environment.
   (II) The Digger should work at any time permitted by the environment to obtain the maximum productivity.

(b) The relation between the command and the condition:
   (I) The Pump is started only when the water level is high.
   (II) When CH4 level is high, no instrument can be started.
   (III) The Ventilator can be started only when the quality of the

atmosphere in the mine is below normal.

(IV) An alarm can sound only when the corresponding index is below the normal value.

The priority order of all commands is:

CH_4 > preempt > water level

So the command caused by CH_4 level has the highest priority. The preempting command sent by operator has the second highest priority. The command triggered by the water level has the lowest priority.

From the above descriptions, we can construct the following table to decide the relationship between the environment condition and the control command for the Pump, the Digger and the Ventilator.

TABLE FOR THE PUMP

THE PUMP CURRENT STATUS	CH_4 LEVEL	WATER LEVEL	PREEMPT COMMAND	THE PUMP NEXT STATUS
Off (0)	Low (0)	Low (0)	Off (0)	Off (0)
Off (0)	Low (0)	Low (0)	On (1)	On (1)
Off (0)	Low (0)	Low (0)	No command (2)	Off (0)
Off (0)	Low (0)	High (1)	Off (0)	Off (0)
Off (0)	Low (0)	High (1)	On (1)	On (1)
Off (0)	Low (0)	High (1)	No command (2)	On (1)

Off (0)	High (1)	*	*	Off (0)
On (1)	Low (0)	Low (0)	Off (0)	Off (0)
On (1)	Low (0)	Low (0)	On (1)	On (1)
On (1)	Low (0)	Low (0)	No command (2)	Off (0)
On (1)	Low (0)	High (1)	Off (0)	Off (0)
On (1)	Low (0)	High (1)	On (1)	On (1)
On (1)	Low (0)	High (1)	No command (2)	On (1)
On (1)	High (1)	*	*	Off (0)

Here * means any value.

From this table, it is quite easy to get the control rule for the Pump. We can express the control rule as given below:

```
(generator == 0) ->
       if
       :: ((CH4 == 0)&&(preempt == 1 || (W_level == 1
          && preempt == 2))) -> generator = 1
       :: else -> generator = 0
       fi;
(generator == 1) ->
       if
       :: ((CH4 == 1)||((CH4 == 0) && ((W_level == 0
          && preempt == 2) ||(preempt == 0)))) ->
          generator = 0
       :: else -> generator = 1
       fi
```

Here generator indicates the states of the Pump; CH_4 indicates the CH_4 level, and W_level indicates the water level. As for the control rule for the Digger, it is quite easy to get:

```
if
:: ((CH4 == 0) && (W_level == 0) && (C_S < 110)) ->
   digger = 1
:: else -> digger = 0
fi
```

This rule means: if the CH_4 level is low, the water level is low and no extra ore piled on the Conveyor, then the Digger should run to guarantee the maximum productivity. Otherwise, it should stop to guarantee the safety requirement.

The control rule for the Ventilator can be modeled as shown below:

```
if
:: ((CH4 == 1) || (AF_level == 0) || (CO == 1)) ->
    ventilator = 1
:: else -> ventilator = 0
fi
```

This rule means: if the CH_4 level is high or the air flow rate is too low or the CO level is high, then the Ventilator should start to improve the quality of the atmosphere in the mine.

If some index becomes worse, the Controller should sound an alarm. For example: if CH_4 is high, then the CH_4_alarm will be set which means the alarm has been sent out. Since this is quite easy to model, so we don't describe it in detail here. More detail is given in the Appendix.

4. Specifications.

The properties can be classified into two types:

- **Liveness properties**
- **Safety properties**

Before discussing specifications, we give the explicit explanation of the meaning of Boolean values.

	0	1
Generator	Pump generator is Off	Pump generator is On
Digger	Digger is Off	Digger is On
W_level	Water level is Low	Water level is High
CH4	CH_4 level is Low	CH_4 level is High
CO	CO level is Low	CO level is High

For generator, we use "current_generator" to record the current value of the generator and "next_generator" to record the value of the generator at the next moment .

The variable preempt has three possible values: 0,1,2:

Value	0	1	2
Meaning	Command: turn Pump generator off	Command: turn the Pump generator on	No command sent

Now we discuss the properties:

(1). Liveness properties

The liveness properties require that when a specific event happens, it will be responded to eventually. For example, when the water level is high, the pump generator should be eventually turned on. This is one of the basic and most important properties.

1.

When the water level is high, the pump generator should be turned on eventually.

[] ((W_level == 1) -> <> (generator == 1))

And the counterpart when the water level is low is similar:

2.

When the water level is low, the pump generator should be turned off eventually.

[] ((W_level == 0) -> <> (generator == 0))

3.

If the weight of the load on the conveyor is more than 110, then eventually the weight should be less than 110 again, that is, the weight of load should not increase without limit.

[] ((C_S > 110) -> <> (C_S <= 110))

This property assures that the weight of load can not increase infinitely. Actually when C_S is greater than 110, the digger will be shut down so that the weight of load can decrease.

(2). Safety Properties

In the mining system, when the environment is safe, the digger should keep working so that the highest efficiency is assured.

4.

When the level of CH_4 is Low, the water level is Low and the weight of load is less than 110, the digger should keep working.

[] ((CH4 == 0) /\ (W_level == 0) /\ (C_S < 110))

--> (Digger == 1))

The mining system is a safety-critical environment. If a machine is operated when the environment is unsafe, it could cause an explosion. Among the factors mentioned above, the level of CH_4 is the most critical one. The following two properties deal with the level of CH_4:

5.

When the level of CH_4 is high and the pump generator is working, it should be shut down.

[] (current_generator ==1) /\ (CH4 ==1) →
(next_generator == 0))

6.

When the level of CH_4 is high and the pump generator is not working, it should not be started.

[] ((current_generator == 0) /\ (CH4 == 1) →
(next_generator == 0))

Only when the above two are satisfied can we start the pump generator.

There are several factors which can affect the behavior of the Pump machine and the Digger, for example, the water level, the level of CH_4, and the value of preempt. So, the action of the Pump is influenced not only by the environment, but also by the operator's direct command.

The priority of these factors is important to the controller. It must decide a proper action when more than one of these factors are present.

7.

When the water level is high and at the same time, the level of CH_4 is high, the pump generator should be shut down.

$$[]\ ((CH4 == 1) \wedge (W\_level == 1) \rightarrow (next\_generator == 0))$$

8.

When the water level is low and the level of CH_4 is low and the value of preempt is 1, then the pump generator should be started.

$$[]\ ((CH4 == 0) \wedge (preempt == 1) \wedge (W\_level == 0) \rightarrow (next\_generator == 1))$$

In this case, the water level is low and the Pump generator should not be started if there is no command from Console. But when the user sends the command to the Pump, it should be turned on.

9.

When the water level is high, and the level of CH_4 is low and the value of preempt is 0, the pump generator should not be started.

$$[] ((CH4 == 0) \wedge (preempt == 0) \wedge (W\_level == 1) \rightarrow (next\_generator == 0))$$

In this case, if there is no command from Console, the Pump generator should be started. But when the operator sends a command to keep the pump generator shut down, it will not be started.

10.

When the level of CH_4 is high and the value of preempt is 1, the pump generator should not be started.

$$[] ((CH4 == 1) \wedge (preempt == 1) \rightarrow (next\_generator == 0))$$

This shows that the command will not change the status of the machine if the environment is not safe. So a wrong command will not cause an accident.

From the above four properties, we can see that the priority order of the three factors water level, operator's command and CH4 level is:

CH_4 level > operator's command > water level

just as described in the previous section of the controller design.

In our control logic, the digger can only work when the water level is low. The pump generator, on the other hand, should start working when the water level is high and should stop working when the water level is low. Thus the pump generator and the digger do not work at the same time.

11.

The Pump generator and the Digger should not work at the same time.

[] (! ((generator ==1) /\ (digger == 1)))

5. Properties and Results

- **Properties**

(1). Liveness properties

1. *[] ((W_level == 1) -> <> (generator == 1))*
2. *[] ((W_level == 0) -> <> (generator == 0))*
3. *[] ((C_S > 110) -> <> (C_S <= 110))*

(2). Safety Properties

4. *[] ((CH4 == 0) /\ (W_level == 0) /\ (C_S < 110)) --> (Digger == 1))*
5. *[] (current_generator ==1) /\ (CH4 ==1) → (next_generator == 0))*
6. *[] ((current_generator == 0) /\ (CH4 == 1) → (next_generator == 0))*
7. *[] ((CH4 == 1) /\ (W_level == 1) → (next_generator == 0))*
8. *[] ((CH4 == 0) /\ (preempt == 1) /\ (W_level == 0) → (next_generator ==1))*

9. *[] ((CH4 == 0) /\ (preempt ==0) /\ (W_level == 1) → (next_generator ==0))*

10. *[] ((CH4 == 1) /\ (preempt == 1) → (next_generator == 0))*

11. *[] (! ((generator ==1) /\ (digger == 1)))*

The above eleven properties were proved to be true.

We tried to prove a property:

[] (C_S <=110)

which means that the weight of the load on the conveyor is kept less than or equal to 110 all the time. But it turned out that it is not true. So it is possible that the weight of the load is more than 110. But we proved the following property:

[] ((C_S > 110) -> <> (C_S <= 110))

which shows that although the weight of the load could be more than 110, it will not keep increasing forever.

- **Results**

The results of checking some of the properties show that the checking process can be quite complex. For example, the result for checking the first liveness property is:

```
State-vector 56 byte,depth reached 1238013, errors: 0
  619007 states, stored
     616 states, matched
  619623 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 828 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
39.616    equivalent    memory    usage    for    states
(stored*(State-vecto  + overhead))
      compressed  State-vector  =  36  byte  +  8  byte
overhead
27.238      actual    memory    usage    for    states
(compression: 68.76%)
1.049       +memory used for hash-table (-w18)
240.000     +memory used for DFS stack (-m10000000)
0.193       +memory used for other data structures
268.389     =total actual memory usage
```

Note the number of states and transitions in the above example. In fact, during the checking process, we tried several depth numbers

in order to get the final correct result. At first we used the default search depth (10,000), but it was insufficient. We had to increase the search depth to 10,000,000 before getting the correct result. Depth reached was `123,8013`.

6. Discussion

In the simple mining system, there are several concurrent processes. The value of the sensors for the water level, CH_4, CO and the weight of load on the Conveyor change randomly. And the operator may send commands to the machine directly now and then. When we designed the system, we used modular method by which we designed each module first, and synchronized them. Although each module is not very complex by itself, but after synchronization, the number of states is quite large. In fact this "state explosion" problem is quite common in automata theory. The biggest problem in modular design is that the actions in different modules may conflict with each other so that although each single module will not deadlock, after the synchronization, the whole system will deadlock. So checking these specifications in centralized model is necessary. But as we know, due to the huge number of states, it is almost impossible to check those specifications manually. Fortunately we have SPIN, a quite powerful verification tool. With the tool of PROMELA/ SPIN, we can check all the specifications easily and completely.

Alternatively, we could have used SMV to check our model. We did not do that, feeling that Promela is a more natural modeling language for our taste.

From the above example, we can also see a shortcoming of model checking tools like SPIN - the verification tool simply traverses the whole transition structure of the centralized model. Note that if we have 20 modules with 20 states each, then when we synchronize them, by using current technology, at the worst case, the final centralized model will have 20^{20} states. Although these verification tools are quite fast, handling such huge model is still a very hard work and may cost a lot of time. Fortunately, the size of our model is relatively small. So in the future, maybe we should improve the searching and decision algorithm of SPIN so that it has some kind of artificial intelligence and in each verification, it does not need to traverse the whole transition structure of the model. This will make SPIN much more powerful. Of course, the problem of finding a more efficient way to model the synchronized process is also an important research topic and its result can be used to improve the efficiency of SPIN.

Generally speaking, after we spent about two weeks on modeling and proving properties about the model, we find that PROMELA/SPIN is a quite useful tool for verification of specifications. After using this tool, some fuzzy ideas about the real system became clear and some hidden logic errors were found easily. For example, when we tried to model the system with the

liveness property that if the water level is high, then eventually the pump will start, there was an error in our first model. In that model, we assume that the operator could send the preempting command at any time. Then if events "the CH_4 level is high" and "the operator issuing "off" command" occurred in alternation, the pump could not start forever (or we can say: the pump is "starved" by the CH_4 level and the preempting command of the operator). After we used SPIN, we found this hidden error and modified the behaviors of the operator so that only every fifth unit of time, the operator has a chance to issue a preempting command. By using this new model, we can guarantee that if the water level is high, then eventually the pump will start to work. This experience leaves us with a deep impression about the power of model checking tools. We think that they should be widely used to guarantee efficiency and correctness of models.

References

1. Additional material of the notes of *Discrete-Event Systems*, W.M. Wonham 1998.

2. Nielson K. and Shumate K. Designing *Large Real-Time Systems with Ada*. New York: MaGraw-Hill

3. Gerard J. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, 279-295, Vol. 23, No. 5, May 1997.

4. Thierry Cattel. Process Control Design Using SPIN, Spinworkshop, Montreal, Canada 1995.

5. Gerard J. Holzmann. Design and Validation of Computer Protocols, Prentice Hall, New Jersey, 1991, ISBN 0-13-539925-4.

6. Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. Proc. First IEEE Symp. on Logic in Computer Science, 1986, pp. 322-331.

7. Gerth, Doron Peled, Moshe Vardi Rob, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. Proc. PSTV 1995 Conference, Warsaw, Poland.