International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 21 (2017) pp. 11213-11217

Modelling a Safety-Critical System through CCS

Ram Chandra Bhushan
Research Scholar, Department of Computer Science and Engineering,
Motilal Nehru National Institute of technology, Teliyargang, Allahabad-211004, Uttar Pradesh, India.

Orcid Id: 0000-0003-3094-6182

Dharmendra K. Yadav
Associate Professor, Department of Computer Science and Engineering,
Motilal Nehru National Institute of technology, Teliyargang, Allahabad-211004, Uttar Pradesh, India.

Abstract

Model checking is a technique for automatically verifying the correctness of properties of finite-state systems in computer science. This paper deals with the modeling of a safety critical system, named "A level-crossing control system". The specifications of the model are expressed as processes in the calculus of communicating systems (CCS). The requirements of the system are written in the form of properties or formula in temporal logic and Mu-calculus, which has been verified then with the help of Concurrency Workbench (CWB-NC) tool. Specifications in CCS have been verified against the requirements of the system using Mu-Calculus. It is motivated by a temporal logic specification. The main safety requirement of the crossing system is that there should never be a train and a car inside the crossing at the same time, which has been achieved in this paper by counting the number of cars entered into the crossing and number of cars left from the crossing through modeling. Before lowering the gate, the number of cars entered must be equal to the number of cars left. This main requirement of the crossing has been verified with the help of CWB-NC tool.

Keywords: CCS, CWB-NC, Model checking, Modal calculus, Mu-calculus

INTRODUCTION TO MODEL CHECKING

Model checking or property checking in computer science refers to a problem for a given model of a system, exhaustively and automatically checking, whether this model meets a given set of specifications and requirements. Typically, one can think of hardware or software systems, whereas the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash.

Software testing is different form model checking because it shows the presence of errors and talks about the statement, branch and path coverage of a program [9] whereas model checking approach is to check system exhaustively.

Model checking is also a technique, which automatically verifies through process logic [1] the correctness properties and safety requirements of finite-state systems. In order to obtain a solution for such a problem algorithmically, both the model of the system and the specification has to be formulated in some precise mathematical language like process algebra [2]. To this end, the problem is formulated as a task in logic, namely to check whether a given structure satisfies a given logical formula. This general concept can be applied to several types of logics and suitable structures. A simple model checking problem is to verify that, whether a given formula in the propositional logic is satisfied by a given structure or not.

UNDERSTANDING OF FORMAL VERIFICATION

Formal verification is the process of checking whether the design of a model satisfies the defined required properties. We are concerned with the formal verification of designs that may be specified hierarchically. In order to formally verify a design, it must first be converted into a simpler 'verifiable' format. The design is specified as a set of interacting systems, each has a finite number of configurations, called states. States and transitions between states constitute finite state machines (FSM). The first step in verification consists of obtaining a complete FSM description of the system. Given a present state, the next state of an FSM can be written as a function of its present state and inputs. Next step is to build a unique formula in process algebra [2] for each requirement, which can be verified against FSM as true or false.

INFORMAL DESCRIPTION OF THE SYSTEM

The layout of the level crossing system [3] is shown in Figure 1 [3]. There are two sensors on the approach side of the line, TA (Train Approach) and TI (Train In), which are located after the appropriate lights ALight and ILight respectively. One more sensor TO (Train Out) is there on the exit side of the crossing. The track lights can be either red or green. Initially ALight (approach light) is green and ILight (in light) is red. These change on receiving the signal from control.

An approaching train when it observes the ALight as red it stops. The train waits for the ALight to become green until then it cannot proceed towards TA signal. The same behavior is carried out at the ILight. When this is green, the train crosses the TI sensor. It sends an observable signal train_in to the environment. When the train leaves the crossing, it sends an observable signal train_out just before crossing TO.

The behavior required of the train driver is that he/she stops the train at a red light and proceeds when the light is green. While proceeding, the train necessarily crosses the sensors on the line. The expected behavior of car drivers is that they must stop their cars when the road light is red and do not proceed until the light changes to green. This is not specified explicitly. Driver attempts to use the crossing after the light has changed to red are taking calculated risks and should be aware that no account of this is taken in specifying the safety features of the system.

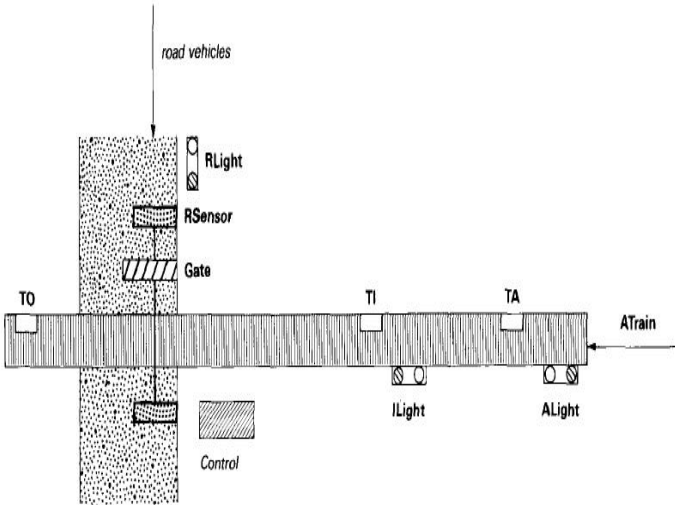

Figure 1: A Level Crossing Control System [3]

There is one more sensor in the system RSensor (Road Sensor) which works in two ways. First, it responds to the (RLight) road light. When RLight is red, the sensor RSensor simply waits for it to turn into green. Once it turns into green, the cars waiting are permitted to enter into the crossing and are counted in and out. If there are no cars waiting, the sensor will just wait for the lights to turn back to red, or for a car to arrive, whichever is first. Every car entering and leaving the crossing is being observed by the control system. Once all cars have left the crossing, i.e. when the number of cars entered is equal to the number of cars left, the sensor checks the lights again.

The main safety concern is threatened when the lights might have changed to red in the meantime. Modeling the situation in which cars may choose to ignore red lights and hence prevent the train from using the crossing. The sequential nature of the control makes it safe for them to do this. It is important to note here, however, that when the sensor is busy in reading the red light, at the same instance of time it will not sense the cars on the crossing. Having checked once that the crossing is clear, it sends the acknowledgment to control system, which in reply begins to close the gate. But if at this point of time a car enters into the crossing, it will not be sensed and an accident may take place. That is why the responsible behavior of car driver is assumed. A car using the crossing legitimately and breaking down before it left will be sensed, and will, therefore, be safe.

SAFETY REQUIREMENTS OF THE SYSTEM

The highest priority safety requirement [8], is that there should never be a train and a car inside the crossing at the same time. This main safety requirement can be achieved by the following lower level constraints [8] on the application domain.

1) For every train, if it is outside the crossing and the ALight (approach light) is red, train remains outside unless approach light turns back to green.
2) For every train, if it is before TI and ILight (in light) is red, the train does not cross unless in light turns back to green.
3) Once the RLight (road light) has been switched to red, cars in the crossing will be allowed to leave before the barrier is lowered.
4) Once the RLight (road light) has been switched to red, cars outside the crossing will not be allowed to enter into the crossing.
5) Once the RLight (road light) has been switched to green, cars outside the crossing will be allowed to enter into the crossing.
6) If the crossing is open for cars, the RLight must be green and the ALight must be red and there must be no train in the crossing.
7) If the gates are closed, the RLight must be red.

THE CCS SPECIFICATIONS OF THE SYSTEM

The calculus of communicating systems (CCS) [5] is a process calculus. Its actions, models indivisible communications between exactly two processes or participants. CCS [6] is a formal language which includes primitives for describing parallel [7] composition, choices between actions and scope restriction. CCS can also be timed [6], which is useful for evaluating the qualitative correctness of properties of a system such as a deadlock or livelock. The CCS for the main model is presented in Table 1.

Two processes SafeCrossing without car and SafeCrossing with car are defined in it. If no car has entered into the crossing then two signals train_in and train_out will be received by the control signal. Whereas, if cars got entered into the crossing then before lowering the gate it is being checked, that the number of cars entered is equal to the number of cars left or not.

Table 1: Main Model

SafeCrossing = $train_{in}$’ . $train_{out}$' . SafeCrossing + car_{in}’ . SafeCrossing_cars(1)
SafeCrossing_cars(n) = car_{in}’ . SafeCrossing_cars(n+1) + car_{out}’. if n=1 Safecrossing
else Safecrossing_cars(n-1)

The set of processes involved in building the system has been divided into two groups namely road control group and train control group. In Figure 2 it can be seen that the processes above environment falls under the road control group, whereas the processes below environment comes under train control group.

CCS for Train Control Group

In this section the CCS [9] of all the processes involved in the train control group has shown. Three sensors, TAS (Train Approach Sensor), TIS (Train In Sensor) and TOS(Train Out Sensor) are triggered by the signals t_a, t_i, and t_o respectively, sent by the processes TA(Train Approach), TRAIN_IN and TRAIN_OUT respectively. Once these sensors triggers, it sends signals to the control (a, i, 0) which tells about the position of the trains. Track lights TAL_RED and TAL_GREEN are two processes which represents the two colors of track approach light. Similarly, TIL_RED and TIL_GREEN represent the two colors of track in light. Initially, the train approach light is green and the train in Light is red. These processes change their states from red to green and vice versa on receiving the signal from control. Table 2 shows the CCS for train control group processes.

Table 2: Train Control Group

```
proc TAS = t_a . 'a . TAS
proc TIS = t_i . 'i . TIS
proc TOS = t_o . 'o . TOS

proc TAL_RED = 'send_a_red . change_a . TAL_GREEN
proc TAL_GREEN = 'send_a_green . change_a . TAL_RED
proc TIL_RED = 'send_i_red . change_i . TIL_GREEN
proc TIL_GREEN = 'send_i_green . change_i . TIL_RED

proc TA = send_a_red . TA + send_a_green . 't_a . TRAIN_IN
proc TRAIN_IN = send_i_red . TRAIN_IN + send_i_green . 't_i . TRAIN_OUT
proc TRAIN_OUT = 'train_in . 'train_out . 't_o . TA
```

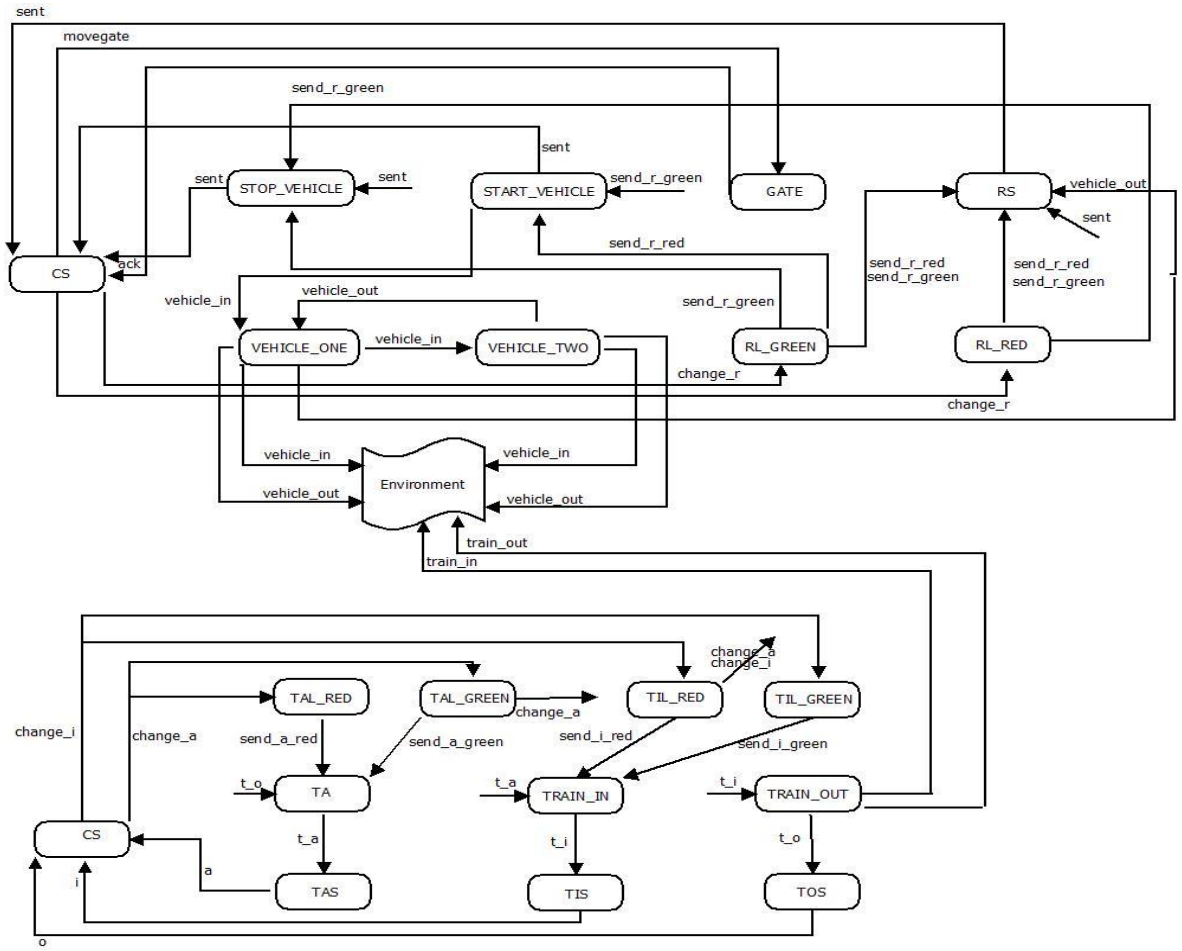

Figure 2: State Diagram for Complete System

CCS for Road Control Group

In this section, the CCS of all the processes involved in the Road control group has shown. Two processes RL_RED and RL_GREEN are the two states of road light. Process RS is a road sensor which allows all the waiting cars to move into the crossing if it sees the road light as red. RS after allowing the cars into the crossing, it goes into START_CAR state. In this case study for the sake of simplicity, we have restricted the maximum number of cars as two. So, it can be understood clearly that if the system is in CARS_ONE state, it goes to CARS_TWO state if it sees an car entering into the crossing and it goes to the state RS if it sees a car going out of the crossing. Similarly, if the system is in CARS_TWO state, it goes to the state CARS_ONE state, if a car goes out of the crossing and there is no transition defined if another car enters into the crossing. Because here maximum only two cars are allowed, so we have not defined the transition for CARS_TWO on car_in action. Table 3 shows the CCS for road control group processes.

Table 3: Road Control Group

```
proc RS = send_r_red . 'sent . STOP_VEHICLE + send_r_green . START_VEHICLE

proc STOP_VEHICLE = send_r_green . 'sent . RS

proc START_VEHICLE = 'vehicle_in . VEHICLES_ONE + send_r_red . 'sent . STOP_VEHICLE


proc VEHICLES_ONE = 'vehicle_in . VEHICLES_TWO + 'vehicle_out . RS

proc VEHICLES_TWO = 'vehicle_out . VEHICLES_ONE


proc RL_RED = 'send_r_red . RL_RED + change_r . 'send_r_green . RL_GREEN

proc RL_GREEN = 'send_r_green . RL_GREEN + change_r . 'send_r_red . RL_RED


proc GATE = movegate . 'ack . GATE
```

CCS for Control System

In this section, the CCS of the main unit of the system [2] i.e. control system (CS) and the entire system has been shown as process CROSSING. All communications take place through control system only. The actions defined in CS are entirely sequential, and so it can be observed that, in which order the actions are performed. CS, when it senses an approaching train, first it changes the approach light to red i.e. it goes to TAL_RED state so that no other trains can enter into the section. It then changes the road light to red i.e. it goes to RL_RED state and, after getting an acknowledgment it closes the gate. Once up to this is done and has been acknowledged, the CS turns the in light to green i.e. it goes to TIL_GREEN state so that the train will be allowed to proceed through the crossing. Once it has been sensed that, the train is in the crossing, the in light is turned back to red i.e. it goes to TIL_RED state. And once the train has left the crossing, the gate will be opened, and once again approach light and road light will be turned back to green and in light will be turned back to red. Also, cars will be allowed to enter into the crossing. Table 4 shows the CCS for control system and Crossing.

Table 4: Control System and Crossing

```
proc CS = a . 'change_a . 'change_r . sent . 'movegate . ack . 'change_i . i . 'change_i . o . 'movegate . ack . 'change_r . sent . 'change_a . CS


proc CROSSING = TA | TAS | TIS | TOS | TAL_GREEN | TIL_RED | CS | TIL_GREEN | GATE | RS \ {a, i, o, t_a, t_i, t_o, change_a, change_i, change_r, send_a_green, send_a_red, send_i_green, send_i_red, send_r_red, send_r_green, sent, movegate, ack}
```

Temporal Logic Based Verification using CWB-NC

The CWB-NC [4] includes facilities for determining whether or not a system satisfies properties expressed in temporal logic. Mu-Calculus is an enriched form of the modal that includes the operators of Computation Tree Logic (CTL). To use CWB-NC model checkers, one first creates a file containing definitions of properties required. The same system is modeled and verified by the authors of this paper through mCRL2 [10], a modeling language and toolset. The property after execution gives one of the two Boolean values True or False. Recursive formulas are given with the least (min) and greatest (max) fixpoint operators. In this section, property for all the safety requirements is presented one by one.

1) For every train, if it is outside the crossing and the ALight (approach light) is red, train remains outside unless approach light turns back to green.

```
prop   Can_Send_ta = min Y = <t_a> tt V <-> Y
prop Approach_Light_red = AG(([send_a_red](not Can_Send_ta)) V ([send_a_green] (Can_Send_ta)))
```

2) For every train, if it is before TI and ILight (in light) is red, train does not crosses unless in light turns back to green.

```
prop Can_Send_ti = min Y = <t_i> tt V <-> Y
prop In_Light_red = AG(([send_i_red] (not Can_Send_ti)) V ([send_i_green] (Can_Send_ti)))
```

3) Once the RLight (road light) has been switched to red, cars in the crossing will be allowed to leave before the barrier is lowered.

```
prop Accident_prevention = (not <send_r_red>tt)
∨ AG([car_in]
EF (<car_out>tt ∧ <movegate>tt))
```

4) Once the RLight (road light) has been switched to red, cars outside the crossing will not be allowed to enter into crossing.

```
prop Can_Car_In = min Y = <vehicle_in> tt \/ <-> Y
prop Car_Not_Allowed = AG([send_r_red](not
Can_Car_In))
```

5) Once the RLight (road light) has been switched to green, cars outside the crossing will be allowed to enter into crossing.

```
prop Car_Allowed =
AG([send_r_green](Can_Car_In))
```

6) If the crossing is open for cars, the RLight (road light) must be green and the ALight (approach light) must be red.

```
prop Crossing_Open = (not<car_in>tt) ∨
AG(<send_a_red>tt ∧
<send_r_green>tt)
```

7) If the gates are closed, the RLight (road light) must be red.

```
prop Gate_Close = (Crossing_Open) ∨
AG((not<car_in>tt) ∧
(<send_r_red>tt))
```

8) In the model there should not be a deadlock. The below property will give false while executing unlike all other properties which gives true.

```
prop Can_Deadlock = min X = [-]ff ∨ <->X
```

CONCLUSION

The level crossing control system is a real-world application and has been modeled in CCS. Also the properties have been verified with CWB-NC tool. Despite being a small system, it is nevertheless too large to be analyzed as a whole system.

The system has been partitioned into smaller communicating subsystems as different processes, and so more amenable to analyze. It has been shown that at no point of time there were both cars and trains in the crossing, which satisfies the main safety requirement. It has been implemented that, cars must be counted in and counted out in the crossing before a train enters into it, and train must leave before the cars get entered into the crossing. Even if a train or a car enters the crossing but fails to leave, the corresponding lights will remain red, not allowing other cars to enter into the crossing. The required behavior of the driver (both train and car) has been specified explicitly. However, drivers who attempt to use the crossing unlawfully do so at their own risk.

REFERENCES

[1] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. Journal of the ACM (JACM), 32(1):137–161, 1985.

[2] Matthew Hennessy. Algebraic theory of processes. MIT press, 1988.

[3] Jean Baillie. A ccs case study: a safety-critical system. Software Engineering Journal, 6(4):159–167, 1991.

[4] Tadeusz Cichocki and Janusz Górski. Formal Support for Fault Modelling and Analysis, pages 190–199. Springer Berlin Heidelberg", Berlin, Heidelberg, 2001. URL: http://dx. doi.org/10.1007/3-540-45416-0_19, doi:10.1007/3-540-45416-0_19.

[5] Robin Milner. A Calculus of Communicating Systems, volume 92 of Lecture Notes in Computer Science. Springer, 1980. URL: https://doi.org/10.1007/3-540-10235-3.

[6] Faron Moller and Chris Tofts. A temporal calculus of communicating systems. In Interna-tional Conference on Concurrency Theory, pages 401–415. Springer, 1990.

[7] Robin Milner. Communication and concurrency. PHI Series in computer science. Prentice Hall, 1989.

[8] Rance Cleaveland, Joachim Parrow, and Bernhard Stefen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. ACM Transactions on Programming Languages and Systems (TOPLAS), 15(1):36–72, 1993.

[9] Ram Chandra Bhushan and Dr. D. K. Yadav. Number of Test Cases Required in Achieving Statement, Branch and Path Coverage using 'gcov': An Analysis . 7th International Workshop on Computer Science and Engineering (WCSE 2017) Beijing, China 25-27 June, 2017, pp.176-180. ISBN 978-981-11-3671-9.

[10] Ram Chandra Bhushan and Dr. D. K. Yadav. Modelling and Formally Verifying A Safety-Critical System through mCRL2. Submitted to 8th International

Conference on Cloud Computing, Data Science & Engineering CONFLUENCE-2018. unpublished