Chapter 8
Model Checking – Case Study of a Temporary Structures Monitoring System

Dongpeng Xu, Xiao Yuan, Dinghao Wu, and Chimay J. Anumba

8.1 Introduction

Model checking or property checking is a formal computer science method for evaluating the extent to which a model meets a given specification. This involves using appropriate symbolic algorithms to traverse the model and check if the specifications, typically expressed as temporal logic formulas, are met. The effectiveness of cyber-physical systems is dependent largely on how well the cyber and physical elements work together. In such systems, any inconsistencies in the system model could result in either failure or unintended consequences. In this chapter, we present a case study of the model checking of a CPS-based temporary structures monitoring system, which was the presented in detail in Chap. 7. We use model checking to try to detect the vulnerabilities in the system. The case study shows that model checking can identify vulnerabilities in a CPS-based system and help developers fix the vulnerabilities.

D. Xu
University of New Hampshire, Durham, NH, USA
e-mail: dongpeng.xu@unh.edu

X. Yuan
Pacific Asset Management Co., Ltd., North Bergen, NJ, USA
e-mail: yuanxiao-006@cpic.com.cn

D. Wu (✉)
Pennsylvania State University, University Park, PA, USA
e-mail: duw12@psu.edu

C. J. Anumba
College of Design, Construction and Planning, University of Florida, Gainesville, FL, USA
e-mail: anumba@ufl.edu

C. J. Anumba, N. Roofigari-Esfahan (eds.), *Cyber-Physical Systems in the Built Environment*, https://doi.org/10.1007/978-3-030-41560-0_8

8.2 Temporary Structures Monitoring System

With the objective of real time and remote inspection of temporary structures, Temporary Structures Monitoring (TSM) system has been developed based on the principles of Cyber-Physical System (CPS). The development of the TSM involves the selection of hardware, such as data acquisition (DAQ) instruments, physical temporary structures, and software environments, such as virtual modeling system, DAQ system for data calibration and transmission, database for data storage, and the communication network. These are fully described in Chap. 7 and by Yuan et al. (2016).

8.2.1 System Design

The CPS-based TSM system in our case study consists of physical temporary structures and their virtual models, which are integrated through a CPS bridge. The architecture of TSM system is shown in Fig. 8.1. In general, the physical and cyber system is connected through the CPS bridge, which enables the mutual

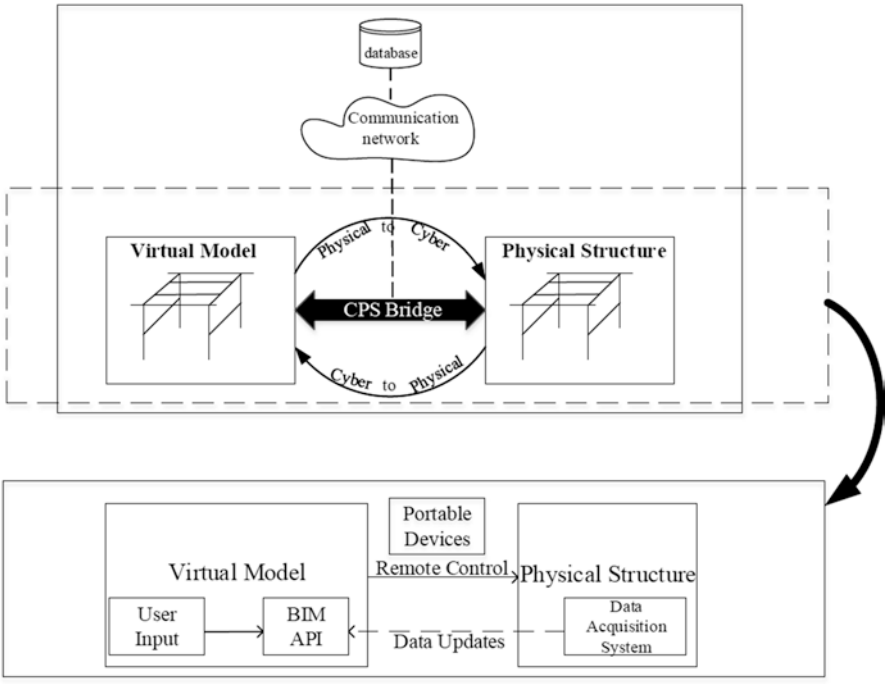

Fig. 8.1 System architecture of CPS-based TSM system

communication between the physical structures and the computing system. In particular, the CPS bridge is supported by the DAQ system and on-cloud database for information collection and exchange; from the physical to cyber system, DAQ system works in transforming information from temporary structures to their virtual model; from the cyber to physical system, the potential hazards of the physical structures will be identified and located in the 3D model, and then communicated to the project managers and safety supervisors through portable devices to take actions to prevent potential accidents.

Generally the prototyping system works in the way that first of all the DAQ system attached to the temporary structures continuously collect and sends information to the database; second, the CPS-based TSM system conducts structural performance analysis every 2 seconds; third, if potential structural failure has been identified, the 3D model will highlight the corresponding components in alarming color. To guarantee timely communication of the potential hazards, the warning message will also be sent to the construction workers and other safety supervisors through portable devices for immediate attentions. Based on the warning information of potential location and causes of problems, the construction workers and project managers shall take corresponding actions to prevent temporary structures accidents. The system workflow is displayed in Fig. 8.2.

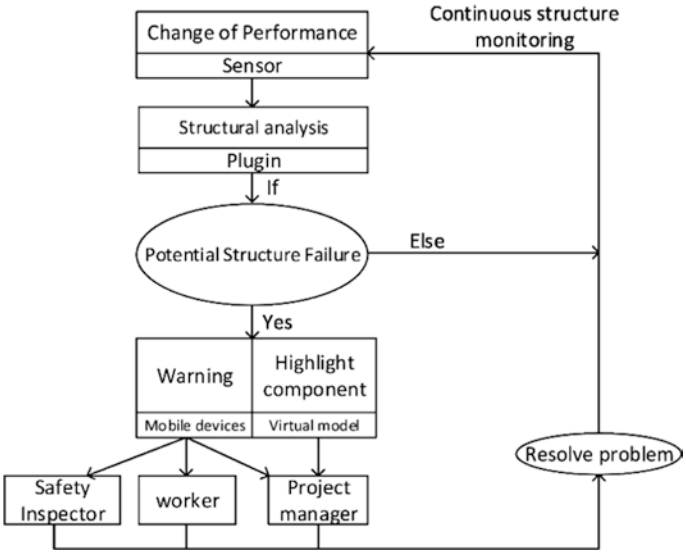

Fig. 8.2 System workflow

8.2.2 System Operation of CPS-Based Temporary Structural Monitoring

As shown in Fig. 8.3, the TSM system prototype enables the interaction between temporary structures and computing system through a virtual model platform, which is developed as a plug-in at Autodesk Navisworks. This plug-in is named as "CPS Monitor". The user-interface of the TSM system consists of the 3D model of temporary structures and the table of property of each component. By clicking the tool button of "CPS Monitor", the window of TSM system appears with the value of warning threshold and system log window presented. The threshold value for each component is identified based on several criteria, including official regulation, practical concerns as well as the capability of the materials used. In particular, the threshold value of the inclination of each component is set according to the requirement of safety managers based on their project experience. The loading threshold is identified based on the component capability specified by the manufacturers' guide while the threshold value of base settlement is based on practical concerns, as the disconnection is an obvious signal indicating the base settlement. The threshold value for plank displacement is set according to the OSHA regulation. For user-friendly consideration, the threshold value is updated and saved based on the last record. Users can modify the threshold value frequently at their convenience. By clicking the "start" button, the TSM system evaluation of temporary structures starts. In general, the information is updated every 2 seconds continuously to check the change of temporary structural integrity. Base on the evaluation result, the TSM system responds correspondingly if potential hazards have been identified. At the virtual model, the components in question will be highlighted with alarming color for immediate attention. To make sure all the workers have received the warning

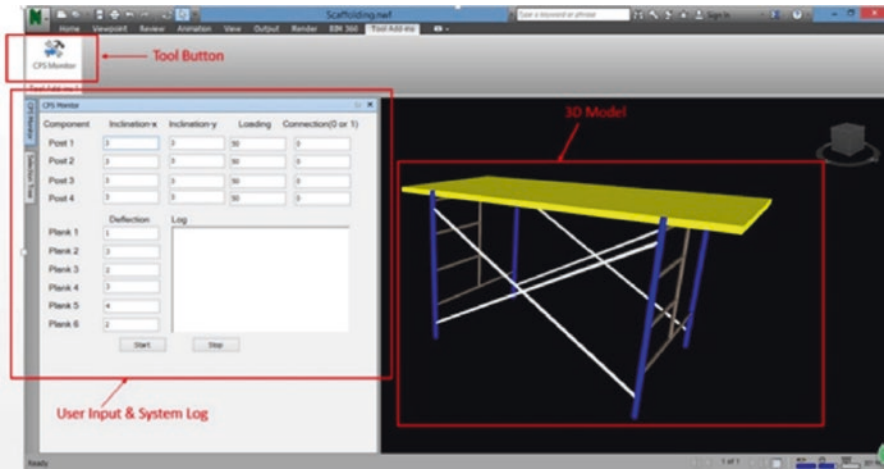

Fig. 8.3 User interface of CPS-based TSM system

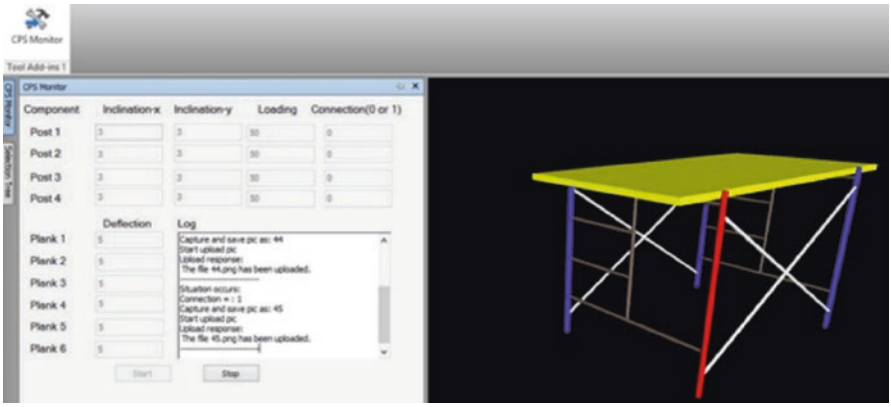

Fig. 8.4 Warning instructions at the 3D virtual model

message, the log window close to the virtual model demonstrates the working progress of the system indicating if the warning messages have been sent to the workers on the construction job site (as shown in Fig. 8.4). Take the base settlement of a frame scaffold for example; one can easily noticed that there is a potential problem due to the structural deficiency of one scaffold post (highlighted in red). Meanwhile, for the end users carrying portable devices, alarms along with detailed information of the structure deficiencies will be displayed through the mobile app of "TSM" installed in the portable devices. For a quick view of the potential problem, a picture of the 3D model with highlighted components in alarming color will be presented with a few text message stating the components in problem (as shown in Fig. 8.4). If the end user wishes to have detailed information, he or she can tap the button of "detail", which then displays the analysis of structure performance of each component (as shown in Fig. 8.5). In this case, the construction workers can immediately understand that the structure deficiency of the post is due to the ground settlement of post 1. With this information, the end users are expected to have a better understanding of the hazardous situation and take corrective actions to address potential structural failures and injuries.

8.3 Model Checking CPS

Model checking methods are useful for systems that have complicated transition relations. The design of CPS is one of these situations. There are some existing work on CPS verification using model checking related techniques. Akella and McMillin (2009) encode the physical system properties into a discrete event system and the CPS is described using the Security Process Algebra. Then the authors apply a model checking called CoPS to check the system's security against all high level potential interactions. At last, they verify a model problem of invariant

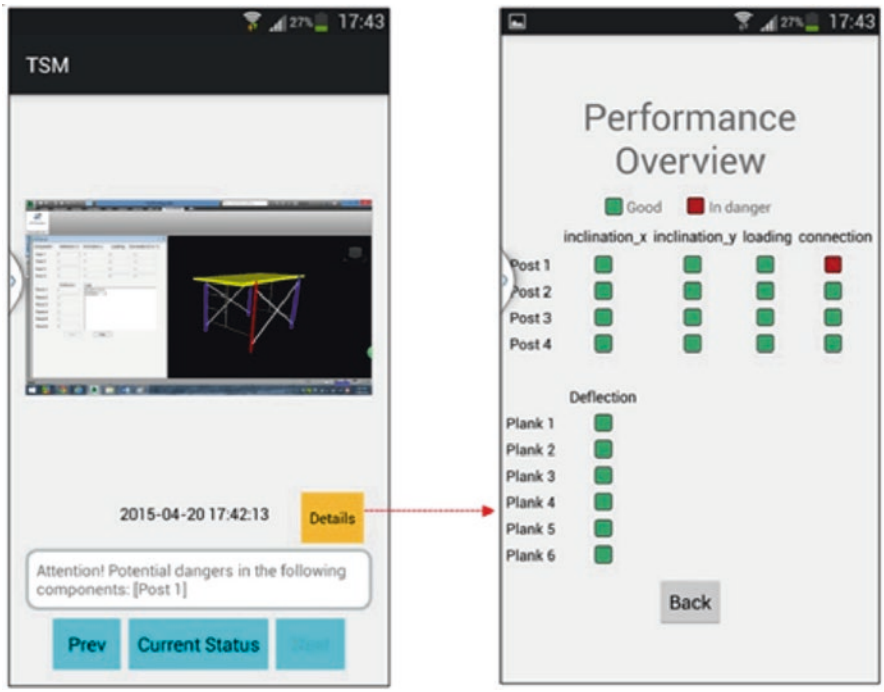

Fig. 8.5 Warning instructions at the portable devices

pipeline flow. In this paper, we use temporal logic model checking to verify and improve a CPS model. Compared to their method, our work is more general to check more potential bugs in a cyber-physical system.

Clarke and Zuliani (2011) apply statistical model checking to CPS verification. Statistical model checking combine the Monte Carlo method with temporal logic model checking. It samples the behaviors of the system model, check their consistency with the temporal logic formula, and finally calculate an approximate value for the probability that the formula is satisfied. The authors successfully verified a fuel control system for a gasoline engine as a CPS example. Different from this work, we use conventional model checking rather than statistical model checking, so our method verifies all system behaviors.

Bu et al. (2011) propose that instead of offline modeling and verification, many CPS should be modeled and verified online. The authors focus on the system's time bounded behavior in short-run future, which is more describable and predictable. They study two cases of their ongoing projects, one on the modeling and verification of a train control system, and the other on a Medical Device Plug-and-Play (MDPnP) application to show that fast online modeling and verification is possible. Compared to their method, our research has different application focus. The CPS in our research does not develop fast. Moreover, we introduce model checking during the process of CPS design and improve the design in several iterations.

There are also other research work (Derler et al. 2012; Karsai and Sztipanovits 2008; Bhave et al. 2011; Banerjee et al. 2012) involving modeling cyber-physical systems. These works mentioned the possibility of applying model checking to verifying CPS, but did not propose a detailed method. In our research, we describe a concrete method to build and verify a CPS model, and provide a case study to show the model checking procedure in a temporary structures monitoring system.

8.4 Spin: The Model Checker

Model checking is a formal method for automatically verifying whether a system meets a set of properties. Model checking tools, such as SMV (Clarke 2009), NuSMV (Cimatti et al. 2000), Spin (Holzmann 1980), and Java Pathfinder (Visser et al. 2003), have been widely applied in verification of hardware and software systems.

Spin (Holzmann 1980) is a widely used model checker. It was originally designed for verifying communication protocols and gradually grows into a powerful verification tool. It is often used to check concurrent systems such as multi-thread software or distributed systems.

Spin is often applied to verify the logic consistency of concurrent systems, which is described using Promela (Holzmann 2004), the specification language for Spin. People use Promela to model the communication between different processes. For ease of understanding the TSM system model written in Spin, we briefly introduce the core part of the Promela language and how to write the specification in Spin. We only present the essential components that is related to the model in this chapter. More details are included in reference books such as Holzmann (2004). In addition, there are many textbooks about verification of systems by formal methods checking (Huth and Ryan 2004; Manna and Pnueli 2012a; Manna and Pnueli 2012b; Roscoe 1997; Holzmann 2007). A Promela model usually includes the following parts:

1. Variable declarations
2. Process declarations
3. Initial process

Figure 8.6 shows an overview of a typical Promela program. Line 1 and 2 declare variables a and b and initialize them. a is an integer variable which is initialized to 0. Similarly, b is a Boolean variable that is initialized to one. Line 4–12 declares two process called Foo and Bar. In Spin, a process is a function that can execute concurrently with all other processes. Line 14–17 define the initial process for launching processes. We will explain more details about the Promela language in Sect. 8.4 when describing the model of the temporary structural monitoring system.

As mentioned before, model checking method is to automatically verify whether $\mathcal{M}, s \vdash \phi$ holds, in which $\mathcal{M}$ refers to the model described using Promela. We also need to use a formal notation to present the properties ϕ. As a practical model checker, Spin provides several properties that can be checked, such as assertions,

```
1  int  a = 0;
2  bool b = 1;
3
4  proctype Foo(int x)
5  {
6    ...
7  }
8
9  proctype Bar(bool y)
10 {
11   ...
12 }
13
14 init
15 {
16   run Foo(a);
17   run Bar(b);
18 }
```

Fig. 8.6 Promela program overview

deadlock, linear temporal logic (LTL) formulas and unreachable code. In this chapter, we only use assertions and LTL formulas to specify the properties. The syntax of assertion statement is assert (<expr>). If the expression expr is evaluated to a non-zero value, the assertion statement is passed; otherwise, Spin will terminate and report an error. Moreover, Spin has its own syntax for LTL formulas as follows:

1. [] P: Always P, which corresponds to the G ϕ predicate in LTL.
2. <> P: Eventually P, which corresponds to the F ϕ in LTL.
3. P U Q: P is true until Q becomes true, which corresponds to the ϕ_1 U ϕ_2 predicate in LTL.

8.5 Modeling the Temporary Structures Monitoring System

This section presents the model checking process and result. We use the Spin model checker (version 6.4.3) released in December 2014. In our implementation, each time when Spin report one property does not hold, the checking process stops and will not continue checking the following properties. It is because that usually one bug leads to multiple violations of the properties. The violation report for one property is enough for the designers to fix the problem. Moreover, reporting one violation each time helps designers focus on one bug each time to save their effort and time.

In this case study, we apply the model checking technique as follows. First, the designers provide us with the initial version of the system design, which is called "Design A." Then we encode it as a Spin model, which is called "Model A". We also implement the properties in Spin. After that, we run the Spin model checker to

check Model A. The checking result shows whether each property passes or not. For the failed test, Spin provides a counterexample and the related backtrack information. We return the checking result to the designers and they confirm whether the counterexample is a real bug. If it is confirmed, the designers will fix them and update the system design to "Design B." After that, we also update the model to "Model B" accordingly. Note that the properties remain the same during the whole model checking procedure. Therefore, we ran Spin again to check the Model B. The checking and revising procedure was repeated until the model passed the model checking for all properties.

Table 8.1 shows the model checking result. The first column presents the models that are built based on different versions of the CPS-based TSM system design. The second to the fourth columns shows the model checking result on different properties. "X" means the corresponding property holds in that model and "$\surd$" means it does not hold. "-" represents that the property is not verified in that model. We reported the violated properties to the designers and they confirmed that they are real bugs in the corresponding design. The following sections provide the details of our discovery and the fix approach in each design and model.

8.5.1 Overview

This section presents the model to describe the cyber-physical system. As shown in Fig. 8.4, the TSM system is a scaffold monitoring system. The left part is a panel that receives the user's input to configure the threshold for the inclination, loading and so on. The log frame displays the checking ongoing status. Besides, there are "start" and "stop" buttons at the bottom. Users can click them to start or stop the checking procedure. The right part in Fig. 8.4 shows a simulation picture of the scaffold. The TSM system will highlight the component in red color when it is likely to be dangerous; otherwise, it will show the safe component in blue. The TSM system includes sensors placed on the scaffold, which are able to monitor the bending, loading, and other features of a component. The checking procedure periodically fetches data from sensors and then analyzes the data to see whether they exceed the configured threshold.

Table 8.1 Model checking result

Model	Property			
	1	2	3	4
A	✓	×	-	-
B	✓	✓	×	-
C	✓	✓	✓	×
D	✓	✓	✓	✓

8.5.2 *Design A*

In this section, we describe the initial design of the TSM system, which is called "Design 1" in this case study. This is the first version the system designers gave to us. The pseudo code in Fig. 8.7 shows "Design A". In this chapter, we use pseudo code (Fig. 8.7) to present the design and use Spin code to show the model in Spin (Fig. 8.8). In Fig. 8.7, the function ClickStart is triggered when users click the start button. Similarly, ClickStop is triggered when clicking the stop button. Thread T() implements the monitoring functions in a while loop. The thread test the global variable S before each round of monitoring. It ends monitoring when S equals zero. Therefore, ClickStart and ClickStop set S to start or stop the monitoring thread T().

As shown in Fig. 8.7, there are two global array variables: pre_states and cur_states. cur_states stores the current risk states of each component and pre_states stores the risk states in the last round of monitoring. They are used to implement the monitor function in line 23 and 24 in Fig. 8.7. Here we briefly describe how it works. First, the monitoring thread T() fetch data from the sensors and store them in a system buffer. Then T() check each component's data with the threshold and update the array cur_states accordingly. The monitoring thread sends an alarm when the cur_states of one component is true and the pre_states is false, which means the risk is happening. At the end of each round of monitoring, all states in cur_states are copied to pre_states.

```
1  global variable S;
2  global array pre_states, cur_states;
3
4  ClickStart()
5  {
6      if (S == 0) {
7          disable all inputs in the panel;
8          S = 1;
9          start new monitoring thread T;
10     }
11 }
12
13 ClickStop()
14 {
15     S = 0;
16     enable all inputs in the panel;
17     reset all states;
18 }
19
20 Thread T()
21 {
22   while (S == 1) {
23     monitoring operations ...
24     update pre_states, cur_states;
25   }
26 }
```

Fig. 8.7 The initial design of the TSMS

```
1  int S = 0;
2  int cur_states = 0;
3  int pre_states = 0;
4
5  int critical = 0;    /* Whether T() enters critical area */
6  int num = 0;         /* The number of thread T() */
7  int started = 0;
8  int connection = 1;  /* Is the network connected? */
9  int alarm = 0;       /* Whether the system sends alarms */
10
11 proctype T()
12 {
13   num++;
14   do
15   :: S != 0 -> printf("Checking...");
16                critical = 1;
17                if
18                :: connection == 0 -> cur_states = 0;
19                                      pre_states = 0;
20                :: connection != 0 -> cur_states = 100;
21                                      pre_states = cur_states;
22                fi
23                critical = 0;
24                printf("Done...");
25   :: S == 0 -> break;
26   od
27   printf("Finish checking.");
28   num--;
29 }
```

Fig. 8.8 Global variables and T() in Model A

We build a model checker in Spin to simulate "Design A." The Promela model is shown in Figs. 8.8 and 8.11, which is called "Model A." Figure 8.8 presents how the global variables and the monitor process T() is modeled in Spin. Figure 8.11 shows the model of the function ClickStart and ClickStop. The code is clearly aligned and the variables' name are consistent with those in "Design A." However, the syntax of Promela might lead to some trouble to understand the code. We briefly explain the code as follows.

In Fig. 8.8, line 1 to line 9 define several global variables. The syntax of variable declaration in Promela is the same as that in the C programming language. Variables can be initialized with an initial value when being declared. Basic type variables are initialized with 0 as the default initial value. Promela supports plenty of types such as byte, integer, Boolean, array, record and so on. We only introduce those types that appear in this chapter. The language menu provides a full description of the types in Promela. Line 1–3 in Fig. 8.8 define the corresponding global variables in the TSMS. Line 5–9 defines several global variables for checking different properties. For instance, critical indicates that the monitoring thread enters an area that could cause race condition problem. It is because T() and ClickStop both have the capability to modify the content in pre_states and cur_states. num refers to the number of thread T() at the same time. Connection simulate the network connection. Alarm means whether the system sends an alarm.

Lines 11–29 define the monitoring process T(). A process is a function that can execute concurrently with all other processes. Since Spin is originally designed to check communication protocols, which usually involve sending and receiving messages between two independent machines, the process concept naturally simulates the communicating subjects. A process is defined by the proctype keyword. Each process can have its own local variables and execution status.

The do loop from line 14–26 in Fig. 8.8 simulates the monitoring procedure. do statement is the loop statement in Promela. Inside the loop, line 17–22 use a branch statement to handle the different situations when the network is connected or not. The branch statement and loop in Spin have a similar form, which is shown in Figs. 8.9 and 8.10. For the branch statement, there are n branches in one if statement. Each branch includes a guide cond_n and the following statements stmt_n_1; stmt_n_2; … . If at least one guide is evaluated to a non-zero value, Spin will nondeterministically choose one branch to execute. The else branch will be executed if all other guards are zero. Moreover, as shown in Fig. 8.10, the loop statement in Promela looks similar to the branch statement, except replacing the keyword if with do. The difference between do statement and if is that, do statement continues the next round of execution after running one branch. A break statement can exit a do loop and transfers the control flow to the end of the loop.

In the do loop in Fig. 8.8, we ignore the trivial details such as fetching data from sensors since they are not the key parts for model checking. We only use two assignments to abstract the procedure of updating pre_states and cur_states. The variable started is used to synchronize ClickStart and ClickStop, because in the TSM system design, users are not able to click the start button or modify any inputs in the panel when T() is already running. Therefore, started is used to simulate the operation to enable or disable all inputs in the panel.

Figure 8.11 shows the remaining part of Model A. ClickStart sets S to 1 and then run the monitoring thread T() and ClickStop set S to 0 and flush pre_states and

Fig. 8.9 If statement in Promela

```
1 if
2 :: cond_1 -> stmt_1_1; stmt_1_2; ...
3 :: cond_2 -> stmt_2_1; stmt_2_2; ...
4 :: ...
5 :: cond_n -> stmt_n_1; stmt_n_2; ...
6 :: else   -> stmt_n+1_1; stmt_n+1_2; ...
7 fi;
```

Fig. 8.10 Do statement in Promela

```
1 do
2 :: cond_1 -> stmt_1_1; stmt_1_2; ...
3 :: cond_2 -> stmt_2_1; stmt_2_2; ...
4 :: ...
5 :: cond_n -> stmt_n_1; stmt_n_2; ...
6 od;
```

```
1  proctype ClickStart()
2  {
3    do
4    :: atomic {started == 1 -> skip;}
5    :: atomic {started != 1 -> started = 1; break;}
6    od
7
8    if
9    :: (S == 0) -> S = 1; run T();
10   fi
11 }
12
13 proctype ClickStop()
14 {
15   do
16   :: started == 0 -> skip;
17   :: started != 0 -> break;
18   od
19   S = 0;
20   printf("Stop.");
21
22   critical++;
23   cur_states = 0;
24   pre_states = 0;
25   critical--;
26   started = 0;
27 }
```

Fig. 8.11 ClickStart and ClickStop in Model A

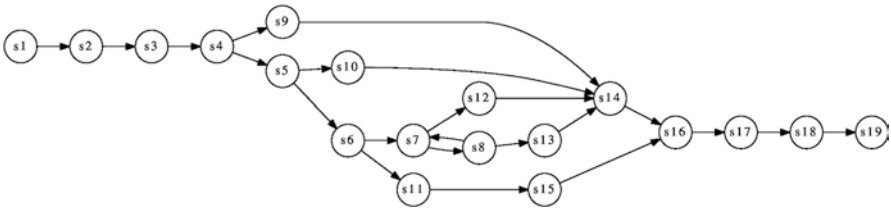

Fig. 8.12 The state transition diagram of Model A

cur_states. Moreover, since flushing pre_states and cur_states in ClickStop could cause race conditions with other threads, we also put critical update operations around for future checking.

Figure 8.12 shows the state transition diagram of Model A. The diagram shows that even a relatively simple model can have many states with complicated relation between them. It is very difficult for human to enumerate and track each path in the state transition system. Therefore, automated model checking approach is helpful to solve this problem. In this state transition diagram, each state contains every variable and its value. Due to the page limitation, we only show several example states here rather than the description of every state. For instance, s1 is the initial state. All

variables in s1 is 0 except connection. For simplicity, we ignore those variables whose value equals to zero. Thus s1 can be represented as {connection = 1}. Similarly, s2 is {connection = 1, started = 1}.

8.5.3 Properties

So far, we have introduced the initial design and model we used to describe the TSM system. In this section, we present the specifications used to check the model. The TSM system is a multi-process system, so common problems such as race condition need to be checked. Moreover, we discuss with the TSM system designers to select a set of properties based on the following two criteria. First, they are true properties, which means they should hold during the execution of the TSM system. Second, they are security related. Violation of these properties will cause severe bugs happen. Generally, we check the properties as follows:

1. `(critical==0 && started==0 && S==0) U (S==1)`: The TSMS is stopped initially until it is started at the first time. This property is used to check whether the initial values are set correctly. S = 0 means the monitoring thread is not stated. Therefore, this property ensures that the global variables critical and started are initialized correctly.
2. `[] (critical! = 2)`: Critical region is the part of a thread that involves update a global variable shared with other threads. Obviously, only one thread can run its critical region at the same time. The variable critical is increased when entering the critical region and decreased when exiting. Thus critical should never equal to 2 during the run time of the system.
3. `[] (thread_num < = 1)`: By the design of the TSMS, only one monitoring thread T() executes at the same time. Since there is no lock for the global variables such as cur_states and pre_states, running T simultaneously could cause race condition. The variable thread_num counts the number of T() running at the same time, so it should not be greater than 1.
4. `[] (connection==0 - > alarm==1)`: This property checks that the TSMS should send an alarm, when the network is disconnected. Connection equals 0 when the network is disconnected and alarm equals 1 when the system is sending an alarm.

8.5.4 Check Design A

As shown in Table 8.1, Model A passes in the model checking of property 1 however fails in that of property 2. Property 1 means that the global variables are correctly initialized. The violation of property 2 indicates that there are multiple threads entering the critical region at the same time. Spin is able to print the counterexample

```
1  ClickStart()  do
2                :: atomic {started != 1 -> started = 1; break;}
3                od
4                if
5                :: (S == 0) -> S = 1; run T();
6
7  T()           num++;
8                do
9                :: S != 0 -> printf("Checking...");
10                            critical = 1;
11                            if
12                            :: connection != 0 -> cur_states = 100;
13
14 ClickStop()   do
15               :: started != 0 -> break;
16               od
17               S = 0;
18               printf("Stop.");
19               critical++;
20               cur_states = 0;
```

Fig. 8.13 The path to produce the counterexample in Model A

for the violated property. Furthermore, Spin can show the state transition path on how to reach the counterexample. Figure 8.13 shows one path to reach the race condition that violates property 2. First, thread ClickStart is executed until line 9 in Fig. 8.11 then invoke the monitoring thread T(). Next, T() is executed to line 21 in Fig. 8.8. Finally, ClickStop is invoked and runs until line 23 in Fig. 8.11. Obviously, the thread T() and ClickStop both reach the critical region to modify the content of cur_states, which will cause the race condition. At the line 20 in Fig. 8.13, the variable critical equals to 2. Therefore, this counterexample triggers the violation of property 1: [] (critical! = 2).

We reported the counterexample and the path to the TSM system designers and they confirmed that this is a real bug in the system design. This bug means clicking stop button could lead to flushing the pre_states and cur_states when checking the status of the scaffold, and thus the TSM system could miss some risky situations. In order to fix this bug, the designers moved the flushing operation from ClickStop to the end of T(). Therefore, only T() is able to update the pre_states and cur_states and the race condition is eliminated. Figure 8.14 shows the updated design, which is called “Design B” in our case study. Due to space limitations, when presenting the updated design in each round of model checking, we only show the parts that are different from the previous design and the comments in the code indicate the modification. For instance, as shown in Fig. 8.14, the designers only updated the function ClickStop and T(). We follow this style to present each updated design in the rest of this chapter.

Fig. 8.14 The updated part in Design B

```
1  ClickStop()
2  {
3      S = 0;
4      enable all inputs in the panel;
5      /* removed: reset all states */
6  }
7
8  Thread T()
9  {
10   while (S == 1) {
11     monitoring operations ...
12     update pre_states, cur_states;
13   }
14   reset all states; /* move it here */
15 }
```

8.5.5 Check Design B

This section show the second iteration of the model checking procedure. After getting the "Design B" from the designers, we revise the model to reflex the updates. Figure 8.15 shows the updated model, called "Model B." Due to the page limitation, we only show the updated part of the model to show the difference between the current and the previous model.

We run Spin on "Model B" and find it passes the first and second properties. However, it violates the property 3, which indicates that multiple monitoring thread T() could run simultaneously. Similarly, we get a counterexample path from Spin. Due to the page limitation, we only describe the key point in the path rather than present the full-length path here. Briefly speaking, the property is violated when the user clicks the stop button and then quickly click the start button. Since in "Design B", clicking stop button enables all the input in the panel, users are able to click the start button before the monitoring thread T() exit. More specifically, when quickly clicking the start button after the stop button, the global variable S will be set to 0 and then set to 1 again. If this situation happens in one iteration of the loop in T(), the loop will not exit so the thread T() will not terminate. However, a new T() is started by ClickStart. Therefore, multiple monitoring thread T() are running in the TSM system, which violates the property 3.

Again, we reported the model checking results to the TSM system designers and they confirmed that this is a real bug. Multiple monitoring thread running simultaneously will cause the TSMS to send repeated tedious alarms. Moreover, in our counter example, there are only two T() running at the same time. In fact, this bug could cause much more T() running simultaneously, which could lead to a system crash. The designers carefully fixed the bug. They disabled all inputs in the panel until the monitoring thread T() exits. The fixed version is "Design C." Figure 8.16 shows the updated part in "Design C." The changed parts are ClickStop and T().

```
proctype ClickStop()
{
  do
  :: started == 0 -> skip;
  :: started != 0 -> break;
  od
  S = 0;
  printf("Stop.");

  /* Remove the flushing operations*/

  started = 0;
}

proctype T()
{
  num++;
  do
  :: S != 0 -> printf("Checking...");
               critical = 1;
               if
               :: connection == 0 -> cur_states = 0;
                                     pre_states = 0;
               :: connection != 0 -> cur_states = 100;
                                     pre_states = cur_states;
               fi
               critical = 0;
               printf("Done...");
  :: S == 0 -> break;
  od
  printf("Finish checking.");

  /* Move here */
  cur_states = 0;
  pre_states = 0;

  num--;
}
```

Fig. 8.15 The updated part in Model B

8.5.6 Check Design C

We repeated the model checking procedure on "Design C". First, we revised the "Model B" according to the update in "Design C". The new model is "Model C". After that, we ran Spin to model check it. This time the first three properties passed but the last property failed. Similar to the previous sections, we obtained the output of Spin and reported it to the designers. Since the whole procedure is similar to the previous rounds of model checking, we skip the description of the steps and only describe the bug we found. The bug detected in the third round of model checking was the lack of network connection checking. When the network is disconnected, the TSM system runs as usual without sending any alarms. However, since the

```
1  ClickStop()
2  {
3      S = 0;
4      /* removed: enable all inputs in the panel; */
5  }
6
7  Thread T()
8  {
9    while (S == 1) {
10     monitoring operations ...
11     update pre_states, cur_states;
12   }
13   reset all states;
14   enable all inputs in the panel; /* move it here */
15 }
```

Fig. 8.16 The updated part in Design C

```
1  Thread T()
2  {
3    while (S == 1) {
4      if (netconnection == 0)   /* check network connection */
5          break;
6      monitoring operations ...
7      update pre_states, cur_states;
8    }
9    reset all states;
10   enable all inputs in the panel;
11 }
```

Fig. 8.17 The updated part in Design D

scaffold status is transmitted via the network, the TSM system could not check the status without the network. This bug could let users think the TSM system is running normally when the network is down, while the TSM system is actually not able to find any risk.

We reported the bug to the designers and they added network connection checking to the loop in T(). The updated version is "Design D" see Fig. 8.17.

8.5.7 Final Design

So far, we ran three rounds of model checking and improved the TSM system design iteratively. As before, we continued to update the model and model check the new model again. Finally, Model D passed all the properties checking. Therefore, as shown in Fig. 8.18, "Design D" is the final version during the model checking aided design procedure.

```
1  global variable S;
2  global array pre_states, cur_states;
3
4  ClickStart()
5  {
6      if (S == 0) {
7          disable all inputs in the panel;
8          S = 1;
9          start new monitoring thread T;
10     }
11 }
12
13 ClickStop()
14 {
15     S = 0;
16 }
17
18 Thread T()
19 {
20   while (S == 1) {
21     if (netconnection == 0)
22         break;
23     monitoring operations ...
24     update pre_states, cur_states;
25   }
26   reset all states;
27   enable all inputs in the panel;
28 }
```

Fig. 8.18 The final design of TSM system

8.6 Discussion

Generally speaking, we identified three bugs by iteratively model checking the TSM system as follows:

1. Multiple threads modify the same array without any lock.
2. Multiple monitoring threads are invoked simultaneously, which will cause racing problems.
3. The TSM system does not send any alarm when the network is down.

Since Spin is able to provide the counterexamples to trigger those bugs, the designers and we can easily reproduce them. Note that it significantly reduces programmers' workload, because usually reproducing a bug is the most difficult and time-consuming step when debugging a program. In our case study, the model checking tool prints the program path to reach each bug. We verified the path and then sent them to the TSM system designers. They found those paths were very helpful when trying to reproduce the bugs. The designers fixed each bug and patched the source code as follows:

1. Rewrite the ClickStop thread to avoid writing to the same buffer simultaneously.
2. Disable all inputs in the panel until the monitoring thread exits to avoid creating multiple monitoring threads.
3. Add a function to check network connection.

In this work, we acquired important experiences about model checking a cyber-physical system. First, we realized that critical bugs do exist in cyber-physical systems, even in those that look as simple as the temporary structures monitoring system in our case study. We really did not expect to detect three defects in such a simple system. The reason is that usually a simple system consists of a large number of states and paths, especially when the system involves concurrent execution. It is quite difficult for the designers to enumerate and track every path to see whether it will lead to a security problem. Model checking is exactly the approach to solving this problem. Modern model checkers such as Spin are able to enumerate, search and verify whether the model meets a set of properties in a very short time.

The second experience is the importance of the iterative model checking procedure. After each round of model checking, we returned the results to the designers, and they fixed it. We then updated our model accordingly and checked the new version again. The iterative model checking prevented the potential bugs introduced by the patch of the previous version. The repeated "checking-fixing" procedure gradually improved the design of the system.

8.7 Conclusions

Cyber-Physical Systems (CPS) have been widely used in different domains recently. The safety and robustness of CPS are becoming more and more important, attracting the attention of researchers. In this chapter, we presented a model checking method to verify and improve the design of CPS. In particularly, we used a temporary structures monitoring system (TSM system) as a case study. First, we worked with the designers to abstract the state transition system and properties. Second, we built a model to encode the TSM system and specification. After that, we ran the model checker to check whether the properties hold in the model. We reported the model checking results to the designers and they revised the original design accordingly. The model checking and revising procedure repeated until all properties held. During several rounds of model checking, the designers worked with us gradually to improve the CPS design. Our case study shows that model checking approach can help improve the design of CPS and reduce human labor.

References

Akella, R., & McMillin, B. M. (2009). Model-checking BNDC properties in cyber-physical systems. *33rd Annual IEEE International Computer Software and Applications Conference, 1*, 660–663.

Banerjee, A., Venkatasubramanian, K. K., Mukherjee, T., & Gupta, S. K. S. (2012). Ensuring safety, security, and sustainability of mission-critical cyber-physical systems. *Proceedings of the IEEE, 100*(1), 283–299.

Bhave, A., Krogh, B. H., Garlan, D., & Schmerl, B. (2011). View consistency in architectures for cyberphysical systems. In *2011 IEEE/ACM second international conference on cyber-physical systems* (pp. 151–160).

Bu, L., Wang, Q., Chen, X., Wang, L., Zhang, T., Zhao, J., & Li, X. (2011). Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior. *SIGBED Review, 8*(2), 7–10.

Cimatti, A., Clarke, E., Giunchiglia, F., & Roveri, M. (2000). NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer, 2*(4), 410–425.

Clarke, E.M. (2009): Model checking at CMU. http://www.cs.cmu.edu/~modelcheck/index.html.

Clarke, E. M., & Zuliani, P. (2011). Statistical model checking for cyber-physical systems. In *Automated technology for verification and analysis* (pp. 1–12).

Derler, P., Lee, E. A., & Vincentelli, A. S. (2012). Modeling cyber-physical systems. *Proceedings of the IEEE, 100*(1), 13–28.

Holzmann, G.J. (1980). Spin: Formal verification. http://spinroot.com/spin/whatispin.html.

Holzmann, G. J. (2004). *The SPIN model checker: Primer and reference manual* (Vol. 1003). Boston: Addison- Wesley Reading.

Holzmann, G. J. (2007). *Design and validation of computer protocols*. Upper Saddle River: Prentice Hall.

Huth, M., & Ryan, M. (2004). *Logic in computer science: Modelling and reasoning about systems*. Cambridge, MA: Cambridge University Press.

Karsai, G., & Sztipanovits, J. (2008). Model-integrated development of cyber-physical systems. In *IFIP international workshop on software technologies for embedded and ubiquitous systems* (pp. 46–54).

Manna, Z., & Pnueli, A. (2012a). *The temporal logic of reactive and concurrent systems: Specification*. Springer Science & Business Media.

Manna, Z., & Pnueli, A. (2012b). *Temporal verification of reactive systems: Safety*. Springer Science & Business Media.

Roscoe, A. W. (1997). *The theory and practice of concurrency*. Upper Saddle River: Prentice Hall.

Visser, W., Havelund, K., Brat, G., Park, S., & Lerda, F. (2003). Model checking programs. *Automated Software Engineering, 10*(2), 203–232.

Yuan, X., Anumba, C. J., & Parfitt, M. K. (2016). Cyber-physical systems for temporary structure monitoring. *Automation in Construction, 66*, 1–14.