System safety property-oriented test sequences generating method based on model checking

Y. Zhang[1], X. Q. Zhao[1], W. Zheng[2] & T. Tang[1]
[1]*State Key Laboratory of Rail Traffic Control and Safety, Beijing Jiaotong University, China*
[2]*School of Electronic and Information Engineering, Beijing Jiaotong University, China*

Abstract

In this study, model checking is used to generate a suite of test sequences to validate whether the System Under Test (SUT) satisfies the defined safety properties. Firstly, a Coloured Petri Net (CPN) model is abstracted and derived from the system requirement specification of the SUT with a hierarchical modelling approach. A state space analysis is used to verify the model with respect to a set of correctness criteria that include the absence of deadlocks and livelocks. Secondly, some system safety properties defined by the experts are described with a non-standard query and extended computation tree logic. Finally, based on the model without deadlocks and livelocks, the negation of safety properties could be checked by analyzing the occurrence graph and the strongly connected components graph of the model. If the model does not satisfy the specified property, the process of model checking could return some counterexamples. From these counterexamples, the nodes and directed arcs that include the interface information are picked out as the interface messages, which are used to construct a test sequence. A case study of using this method on a railway control system is presented, where the CPN Tools is used to model and generate test sequences. All reachable states are analyzed to detect violations and generate the safety related test sequences, which include the required data to be executed on the SUT. The result shows this method is time-saving, labour-saving and can guarantee the conformance between the SUT and the safety properties.

Keywords: model checking, test sequence generation, CPN, railway control.

WIT Transactions on The Built Environment, Vol 114, © 2010 WIT Press
www.witpress.com, ISSN 1743-3509 (on-line)
doi:10.2495/CR100681

1 Introduction

In the 'computer science' sense, safety is defined as nothing bad happening [1]. Safety requirements are those properties that none of the paths in a model satisfy. To check a model for the absence on all paths of specific behaviour means that effectively all paths in the model have to be explored. In order to check for the absence of a property, exhaustive testing of all paths for a safety property is necessary but often infeasible [2]. It is better to verify these properties using model checking [3]. In another sense, safety is defined as freedom from unacceptable levels of risk of harm. As a safety critical system, the railway control system demands greater safety and reliability than other control systems and should not contribute to hazards [4]. Hazard analysis is used to identify hazards and their causes in the safety life-cycle. We can get some failure modes that may cause accidents by Failure Mode Effects Analysis (FMEA) [5]. We want to increase the level of railway system safety by testing the system's ability to defend failure modes, which are called safety properties in this study.

The use of model checkers for automated testing was originally proposed by Callahan *et al.* [2] and Engels *et al.* [6], and since then several different methods to create test sequences with model checkers have been proposed. There are two main categories of approaches to test sequence generation with model checkers [7]: the first category uses special properties that are intended to be violated by a model [8–10]. These properties are called trap properties, and express the items that make up a coverage criterion by claiming that these items cannot be reached. For example, a trap property might claim that a certain state or transition is never reached. A resulting counterexample shows how the state or transition described by the trap property is reached. This counterexample can be used as a test sequence. The second category of test sequence generation approaches uses mutation to change a model such that it violates a given specification [11–13]. Here, the model checker is used to illustrate the differences between changed models and the original model.

The model checkers of Cad SMV, NuSMV, NuBMC and SPIN have been used to generate the test sequence [7, 8, 11, 12, 14, 15]. To the best of our knowledge, CPN Tools have not been used in this area, one main reason is that the latest version of the CPN model checking tool can only determine the correctness of temporal logic formulas, and no counterexample is available. Men and Duan [16] extended the CPN Tools, and made it possible to give the counterexample of the model checking result.

2 Approach description

Normally, a model checker is used to analyze a finite-state representation of a system for property violations. If the model checker finds a reachable state that violates the property, it returns a counterexample, a sequence of reachable states beginning in a valid initial state and ending with the property violation. We base our method on two ideas. The CP-net is used to compute expected outputs and construct the test sequence.

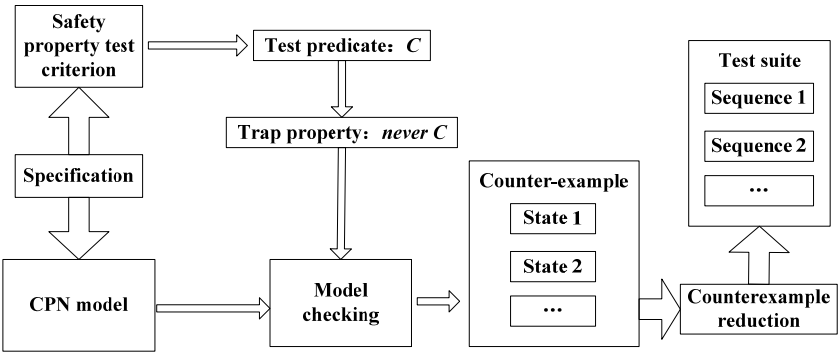

Figure 1: The test sequence generation process.

The general test sequence generation method is shown in fig.1. Given a specification, it is first translated into the model of CPN. A set of test predicates is then picked out from the specification depending on the coverage criterion. These test predicates are generated from safety properties. The negation of a test predicate is called a trap property. CPN Tools is run on the model to see if the negations of test predicates are ever violated. If some of the test predicates are violated then the corresponding counterexample, produced by the model checker, can be used to generate a test sequence. Those test sequences compose one test suite and are guaranteed to cover the corresponding test predicate in the specification. Our target is to construct a suite of safety property test sequences, where a test sequence is a sequence of inputs and outputs In the following we present a summary of our test generation approach.

2.1 Step 1: Safety properties description using ASK-CTL

FMEA is widely used to get system failure modes [5]. These failure modes may cause accidents, which may do great harm to human beings. System safety property test sequences are used to test whether the system can avoid getting into these failure modes. For example, one of the potential RBC failure modes is that RBC does not terminate a communication session after receive the massage 156 (Communication session termination) from onboard. Its related safety property is that RBC terminates a communication session after it receives the massage 156.

Generally, trap properties can be converted to Computation Tree Logic (CTL), Linear Temporal Logic (LTL) or other format depending on the model checker used. CPN Tools use ASK-CTL, which is a CTL-like temporal logic. The logic is an extension of CTL [17] which is branching time logic. Ammann *et al.* [11, 18] described safety properties with CTL following a format of *AG(SafetyInvariant)*, where *SafetyInvariant* is a safety invariant. Then two broad categories of mutation operator are used to generate failing tests and passing test.

In this study, trap properties are used to construct safety property test sequences. Similar with Gargantini and Heitmeyer [8], we use a safety property *P* to describe a predicate. The predicate can be a failure mode state that cannot be

got into after some conditional events happen. Because our goal is not to verify P but to construct a test sequence from P, we should translate the negation of P's premise into ASK-CTL into the format of $M, S_0 \models Inv(not(P))$, which means that it start form the initial state, and the safety property P will never happens. All the safety properties P can be written in the following format: *INV(OR(NOT(Condition)), EV(Safe state))*. It means that it is always true that if the condition event happens, then a safe state will eventually be reached.

2.2 Step 2: SUT model and its environment models

Train control system consists of some subsystems, such as Radio Block Centre (RBC), Onboard and Interlocking. Anyone of them could be the SUT and its environment systems include all of the sub-systems which have Input/Output (I/O) interface with SUT. Every environment model can read its script file and get the input messages set of SUT. An environment model can input an unfixed sequence of input messages to SUT. When an environment model receives a message from SUT model it can send confirm message automatically if it is need to be confirmed. The SUT model described complexly using CPN, begins execution in some initial state and then responds to each input message in turn by changing state and by possibly producing one or more output messages.

Every message in an input message set of SUT is made up by some variables. Values of these variables should be abstracted, for example, a variable named *D_LRBG* described in [19] means the distance between a Last Relevant Balise Group (LRBG) and an estimated front end of the train. The length of variable *D_LRBG* is 15 bits. It has a continuous integral value from 0 to 32767. If we need to test four different situations, its value set can be {"0", "50", "32766", "32767"}, which means 0m, 50m, 32766m and unknown, when its scale is set to be 1m. Given a message M_i, we should get a message set $SM_i = \{m_{i1}, m_{i2}, \ldots, m_{ij}\ldots\}$ ($0 \le j \le |SM_i|$). $/SM_i/$ means the number of elements in SM_i. And m_{ij} is a possible combination of values of all the variables in M_i. SM_i covers all the combinations of values of all the variables in M_i. Suppose a scenario includes N kinds of input messages $M_1, M_2, M_3, \ldots, M_N$, we need to get $SM_1, SM_2, SM_3, \ldots, SM_N$ and create the set of $INF = \{inf_1, inf_2, inf_3, \ldots\}$ for all possible input script files. For example, one script file in INF could be $inf_1 = \{m_{13}, m_{22}, m_{31}, \ldots, m_{N3}\}$, and formula (1) means the number of all the possible files.

$$|INF| = \prod_{i=1}^{N} |SM_i| \tag{1}$$

2.3 Step 3: Deadlocks and livelocks analysis

Regarding the analysis below needs to prove system correct termination, all the deadlocks and livelocks in model should be detected. The finding of self-loop terminal markings is crucial for correctly expressing the CTL-based formulae used to verify the safety properties. In Occurrence Graph (OG), dead markings include system correct termination or deadlock states. Deadlocks states should be

checked out from dead markings. A livelock is detected, when the state space contains a cycle that leads to no markings outside the cycle. In this case, once the cycle is entered it will repeat forever. A convenient way to check the absence of livelocks is to study an automatically generated Strongly Connected Components Graph (SCCG) with the method given by Katsaros [20].

2.4 Step 4: Counterexample generation via state space

The latest version of CPN model checking tool can only determine the correctness of temporal logic formulas, and not counterexample is available. In CPN Tools, there are two model checking functions: *eval_node* and *eval_arc*, which is used to verify state formulas and transition formulas. Men and Duan [16] extended the CPN Tools, and made it possible to give the counterexample of the model checking result. Another method to get special counterexample for trap property P mentioned above is to analyze the state space using ML language. If the model checking result of trap property $M, S_0 \models Inv(not(P))$ is violated, we can find its counterexample in the state space which satisfy safety property P = *INV(OR(NOT(Condition)), EV(Safe state))*. Firstly, we can find out all the condition states which satisfy *Condition,* and all the safety states which satisfy *Safe state*. Secondly, we need to find a route R_2 from a condition state to a safety state. If R_2 is found, then the route of $R_1 + R_2$ is the counterexample, where R_1 is the route from the start state to the condition state.

2.5 Step 5: Test suite creation

Suppose a safety property was got in step 1, a SUT model and a set of $INF = \{inf_1, inf_2, inf_3, \ldots\}$ were got in step 2. Each time, we choose an inf_i from *INF*, and implement step 3 and step 4. If the trap property is violated, the counterexample can be got. If the loop executes $|INF|$ times, the trap property is satisfied and the counterexample cannot be got, the test sequence for this safety property cannot be generated. All the states in a counterexample should be scanned in sequential order. Some of these states include some I/O messages of SUT. I/O messages should be written into a file, which will be used to generate test sequence. After we get a set of safety properties, we can generate a suite of test sequences. For each safety property that it processes, we should check whether the property is already covered by one or more existing test sequences. If so, it proceeds to the next property. If not, we should transform the counterexample generated into a test sequence. If we find that a new test sequence t_2 covers all sequences associated with a previously computed test sequence t_1. In this situation, the test sequence t_1 is discarded because it is no longer useful.

3 Case study

To evaluate the correctness of a railway control system implementation in new lines, black-box testing could be used to determine whether the implementation, given a sequence of system inputs, produces the correct system outputs.

However, these function test sequences do not concentrate on testing the safety properties of the SUT. Therefore, a safety property-oriented test sequence generating method is required. In this case study, RBC is chosen as the SUT and the scenario of Start of Mission is chosen as the example scenario.

3.1 Safety properties in the scenario of start of mission

Suppose we got four potential RBC failure modes in the scenario of Start of Mission using FMEA. Four corresponding safety properties which need to be tested are listed as follow:

1. RBC should terminate a communication session after it receives the massage 156.
2. After RBC receives the valid position report from onboard, and there is not train's LRBG in its recorded balises list. RBC should send the message 24 (general message) to the onboard, in which the packet 24 (session management packet) is used to terminate communication session.
3. If RBC receives a position report and the position is unknown. RBC should send message 24, which includes "position unknown" to onboard.
4. If RBC receives a valid position report from onboard, and there is a train's LRBG in its recorded balises list. However, there are some points between the LRBG and the estimated front end of the train. RBC should send message 24, which includes "position unknown" to onboard.

These safety properties need to be described by ASK-CTL following the specification formats: *INV(OR(NOT(Condition)), EV(Safe state))*, where *Condition* and *Safe state* can be described complexly using ML language. For example, for the safety property 1, *Condition* is that RBC receives the message 156 (Communication session termination) and *Safe state* is that RBC terminates the communication session.

3.2 CPN model for a railway control system

CPN extended the function of PN, inducing data structure and hierarchical decomposition with ML language, used for modelling the behaviour of processes, systems and components. CPN serves data indicating colour on every token. The arc in it should be with an arc–inscription, defining the transform condition between place and transition. Schulz *et al.* [21] defined four different types of nets construct railway control system model. As shown in fig. 2, we defined three types of nets, context net, process net and function net. Context net describes the system architecture and is depicted on the uppermost level. The next level is formed by the process net. This level defines what functions can be passed in what sequence. Function nets are depicted on the lowermost level and implement the function modules in the process net. Fig. 3 presents the top level CP-net that includes substitution transitions of all the three sub-systems. This net corresponds to the representation of the system architecture.

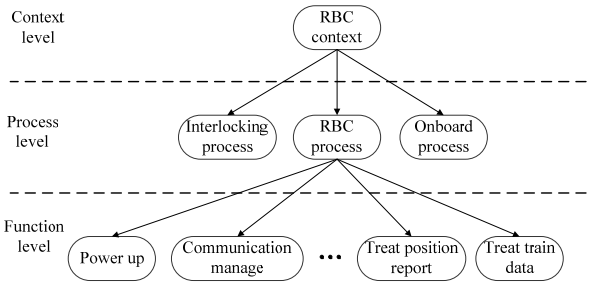

Figure 2: The hierarchical structure of CP-net.

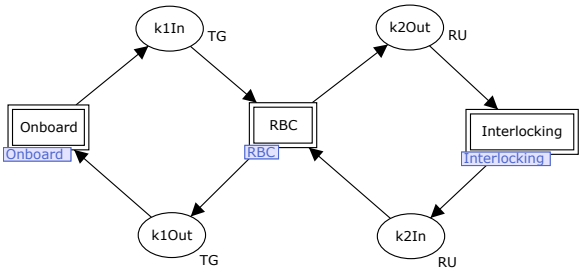

Figure 3: Context net level of CP-net.

We just build the environment sub-systems which exchange information with RBC in the scenario of Start of Mission. For example, if there is no information to be exchanged with neighbour RBC in the scenario of Start of Mission, the neighbour RBC model does not need to be built. RBC is the SUT sub-system. Interlocking and Onboard are the environment sub-systems. Places between the SUT sub-system and the environment sub-systems represent the I/O interfaces of RBC. On this level, all the interfaces are defined as uni-directional channels.

Environment sub-systems just need to read the script file and get the input messages set of SUT at the very beginning. It should send a message when a stimuli message needs to be inputted into the SUT. For example, after an environment sub-system received the message from the SUT, it chooses randomly a message from the input messages set and send it to the SUT. The function of environment models is not complex, it does not need the function net for an environment sub-system and all of its functions can be described in a process net. In the scenario of Start of Mission, we modelled totally seven kinds of messages, which need to be sent to RBC from onboard. For example, *{MS=155,VRS=[{V_N="NID_ENGINE",V_T="201"}]}* means message 155 (initiation of a communication session), which can be written into onboard script file, where the variables of *L_MESSAGE* and *T_TRAIN* were omitted.

Because our RBC model does not judge message length and time stamp and we assume they are always right. The value set of variable *NID_ENGINE* was abstracted to be {“201”} with only one possible value. Interlocking sends routing

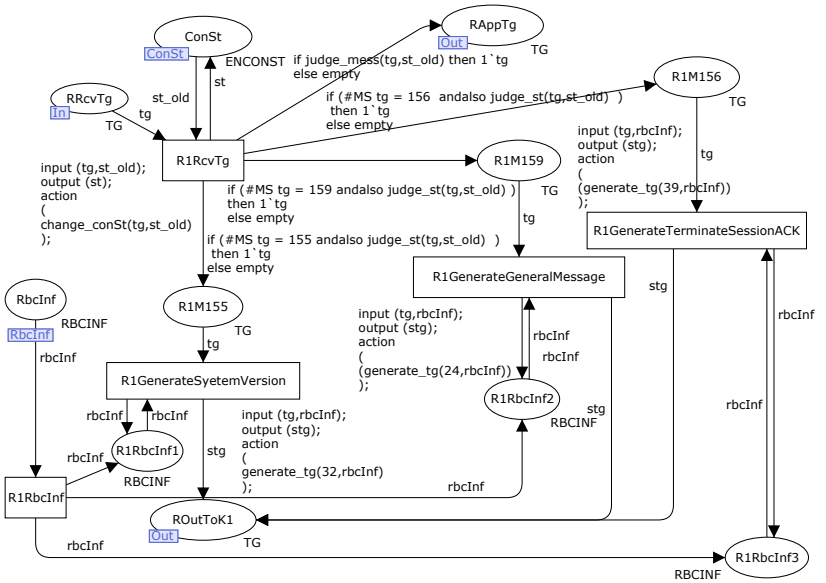

Figure 4: Function net of the communication session management module.

and point information at the very beginning, these information are changeless. Using the method mentioned in step 2, we got 18 possible script files for onboard and 1 possible script file for interlocking.

The SUT model needs to describe the complex behaviours. It is hard to describe all of the functions in only one net. So the functions of SUT model should be described by two levels, process net level and function net level. Process net of SUT defines what functions can be passed in what sequence and function nets implement the function modules. For example, fig. 4 shows a function net, which describes the communication session management function of RBC. RBC will receive three main communication messages, which includes message 155, message 156, message 159 (session established). After RBC receives the communication messages, it should change the communication state recorded in the communication state list (in the place of *ConSt*) and output some messages according to [18]. After a communicate session is established, it is allowed to give messages to application via the place of *RAppTg*.

3.3 Counterexample and test sequence generation

All the experiments were performed on a 2.4 GHz Pentium 4 processor, with 512 MB of main memory. This computer is good enough to generate a full state space. Its state space size was related with the environment script files when the CP-net is fixed. The biggest OG has 33690 nodes and 123774 arcs. It spent 627 seconds to generate the biggest full state OG and 10 seconds to generate the SCCG. The deadlocks and livelocks analysing is crucial for correctly expressing

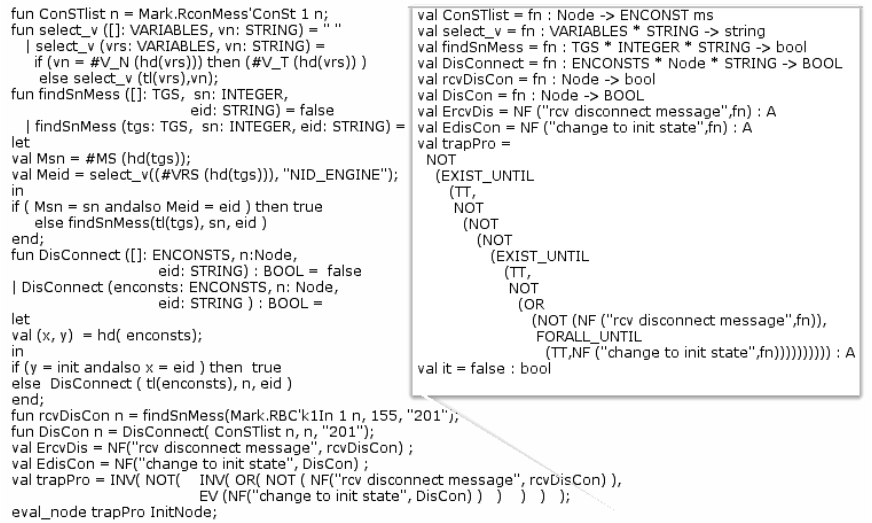

Figure 5: Model checking trap property process of safety property 1.

the CTL-based formulae. Before model checking, deadlocks and livelocks should be detected using the method given in step 3. Then all the safety properties should be translated to the trap properties and be verified by model checking. Fig. 5 shows the whole process of model checking trap property translated from safety property 1. *ConSTlist* is used to return the communication state list, which records some train identifiers and their communication states. *select_v* can select a variable in a message by variable name and return its value. *findSnMess* can judge whether RBC receives a specific message with both a given train identifier and a given message identifier. *DisConnect* can judge whether the connection session is disconnected. Operator *NF("rcv disconnect message", rcvDisCon)* is used to describe the condition of safety property 1. Operator *NF("change to init state", DisCon)* is used to describe the safety state of safety property 1. The trap property 1 can be described using *trapPro*. Model checking is performed by *eval_arc* and the result is false. If the trap property is violated, we can get a counterexample using the method mentioned in step 4. Fig. 6 shows the process of selecting I/O messages in counterexample, which can be used to construct test sequence. In an acquired message, some variables were omitted. These variables need to be added manually to construct test sequence. For example, the variable of *T_TRAIN* in the first message M_0 should be assigned with a random value t_0. We can get formula of $t_i = t_0 + T \times i, (0 \le i)$, where t_i is the value of *T_TRAIN* in M_i, and T means a communication cycle.

3.4 Test suite generation

Repeating this process for all trap properties and calling the model checker only on those trap properties that are not covered we derive the reduced test suite

```
fun Input n = Mark.OnBoard'k1In 1 n <> [];   fun Output n = Mark.OnBoard'k1Out 1 n <> [];
let
  val fid = TextIO.openOut"record/Counterexample.txt"
  val _=TextIO.output(fid, "Messages in counterexample:\n")
  val _= SearchNodes(NodesInPath(1,target() ),  fn n => (Input n orelse Output n ),  NoLimit, fn
n=>STRING.output(fid, " Nodes: "^ st_Node(n)^"\n"^st_Mark.OnBoard'k1In 1 n^"\n"^ st_Mark.OnBoard'k1Out 1
n ^"\n"), [], op ::)
in
  TextIO.closeOut(fid)
End
```

```
record\Counterexample.txt
Messages in counterexample:
" Nodes: 21
OnBoard'k1In 1: 1`{MS=155,VRS=[{V_N="NID_ENGINE",V_T="201"}]} OnBoard'k1Out 1: empty
" " Nodes: 109
OnBoard'k1In 1: empty  OnBoard'k1Out 1:
1`{MS=32,VRS=[{V_N="M_ACK",V_T="1"},{V_N="NID_C",V_T="1023"},{V_N="NID_BG",V_T="16383"}
,{V_N="M_VERSION",V_T="16"}]}
" " Nodes: 171
OnBoard'k1In 1:
1`{MS=159,VRS=[{V_N="NID_ENGINE",V_T="201"},{V_N="N_ITER",V_T="1"},{V_N="NID_RADIO",V_
T="008601033"}]} OnBoard'k1Out 1: empty
" " Nodes: 380
OnBoard'k1In 1: empty  OnBoard'k1Out 1:
1`{MS=24,VRS=[{V_N="NID_PACKET",V_T="57"},{V_N="NID_PACKET",V_T="58"}]}
" " Nodes: 459
OnBoard'k1In 1: 1`{MS=146,VRS=[{V_N="NID_ENGINE",V_T="201"}]} OnBoard'k1Out 1: empty
" " Nodes: 577
OnBoard'k1In 1: 1`{MS=156,VRS=[{V_N="NID_ENGINE",V_T="201"}]} OnBoard'k1Out 1: empty
" " Nodes: 1058
OnBoard'k1In 1: empty  OnBoard'k1Out 1:
1`{MS=39,VRS=[{V_N="M_ACK",V_T="0"},{V_N="NID_C",V_T="55"},{V_N="NID_BG",V_T="1"}]}
```

Figure 6: Process of getting the I/O messages of trap property 1.

Table 1: Test sequences created when monitoring trap properties.

no.	input number	output number	covers
1	4	3	1
2	6	4	1,2
3	5	4	3
4	5	4	4

given in Table 1. The table lists for each test sequence the safety property it was created for as well as all other safety properties that were covered. The I/O number is given as the number of I/O messages in a counterexample. As the example shows, the counterexample created for the second trap property also covers trap property 1. As mentioned in step 5, the counterexample 1 should be discarded. The reduced test suite includes test sequence 2, 3 and 4, which still covers all the safety properties.

4 Conclusion

This work's contribution is a systematic approach in the safety property test suite generation for railway control system using CPN tools. We proposed a notion of safety property *P* with respect to a failure mode. We developed safety property

coverage criteria based on failure modes and operators from ASK-CTL. To the best of our knowledge, the model checking ability of CPN Tools was firstly applied in generation of safety property test sequence suite. We showed how to use CPN to generate test sequence suite that satisfy a given safety property coverage criterion. Finally, we demonstrated the feasibility of our method via application to an example. These early results demonstrate both the method's potential efficiency and its practical utility.

Acknowledgements

The authors wish to thank the anonymous reviewers for the discussions and the help. This paper is sponsored by the National High-Technology Research and Development Program ("863"Program) of China No. 2009AA11Z221, National Science & Technology Pillar Program of China No. 2009BAG12A08, and the Fundamental Research Funds for the Central Universities No. 2009YJS013.

References

[1] Lamport, L., What good is temporal logic. *Proc. of the 9th IFIP Congress Information Processing*, North-Holland: Paris, pp. 657–668, 1983.
[2] Callahan, J., Schneider, F. & Easterbrook, S., Automated software testing using model-checking. *Proc. of 1996 SPIN Workshop*, AMS: Rutgers, pp. 118–127, 1996.
[3] Lee, J.D., Jung, J.I. & Lee, J.G., Verification and conformance test generation of communication protocol for railway signaling systems. *Computer Standards & Interfaces,* **29(1)**, pp. 143–151, 2007.
[4] Leveson, N.G., *Safeware: System Safety and Computers,* Addison Wesley: Massachusetts, pp.123–162, 1995.
[5] Bonfant, G., Belfanti, P. & Paternoster, G., Clinical risk analysis with failure mode and effect analysis (FMEA) model in a dialysis unit. *Journal of Nephrology*, **23(1)**, pp. 111–118, 2010.
[6] Engels, A., Feijs, L. & Mauw, S., Test generation for intelligent networks using model checking. *Proc. of the 3rd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'97*, Springer Verlag: Enschede, pp. 384–398, 1997.
[7] Fraser, G., Issues in using model checkers for test case generation. *The Journal of System and Software*, **82(1)**, pp. 1403–1418, 2009.
[8] Gargantini, A. & Heitmeyer, C., Using model checking to generate tests from requirements specifications. *Proc. of the ESEC/FSE'99: 7th European Software Engineering Conf.*, Springer: Toulouse, pp. 146–162, 1999.
[9] Hamon, G., de Moura, L. & Rushby, J., Generating efficient test sets with a model checker. *Proc. of the 2nd Int. Conf. on Software Engineering and Formal Methods (SEFM'04)*, IEEE Computer Society: Los Alamitos, pp. 261–270, 2004.
[10] Rayadurgam, S. & Heimdahl, M.P.E., Coverage based test-case generation using model checkers. *Proc. of the 8th Annual IEEE Int. Conf. and*

Workshop on the Engineering of Computer Based Systems (ECBS 2001), IEEE Computer Society: Washington, pp. 83–91, 2001.

[11] Ammann, P.E. & Black, P.E., Majurski, W., Using model checking to generate tests from specifications. *Proc. of the 2nd IEEE Int. Conf. on Formal Engineering Methods (ICFEM'98)*, IEEE Computer Society: Brisbane, pp. 46–54, 1998.

[12] Fraser, G. & Wotawa, F., Using model-checkers to generate and analyze property relevant test-cases. *Software Quality Journal*, **16(2)**, pp. 161–183, 2008.

[13] Okun, V., Black, P.E. & Yesha, Y., Testing with model checker: insuring fault visibility. *Proc. of 2002 WSEAS Int. Conf. on System Science*, WSEAS Press: Rio De Janeiro, pp. 1351–1356, 2003.

[14] Fraser, G. & Wotawa, F., Test-case generation and coverage analysis for nondeterministic systems using model-checkers. *Proc. of the Int. Conf. on Software Engineering Advances (ICSEA 2007),* IEEE Computer Society: Los Alamitos, pp. 45–50, 2007.

[15] Belli, F. & Güldali, B., A holistic approach to test-driven model checking. *Proc. of the 18th Int. Conf. on Innovations in Applied Artificial Intelligence*, Springer Verlag: Bari, pp.321–331, 2005.

[16] Men, P. & Duan, Z.H., Extension of model checking tool of colored petri nets and its applications in web service composition. *Journal of Computer Research and Development*, **46(8)**, pp. 1294–1303, 2009.

[17] Clarke, E.M., Emerson, E.A. & Sistla, A.P., Automatic verification of finite state concurrent system using temporal logic. *ACM Transactions on Programming Languages and Systems*, **8(2)**, pp. 244–263, 1986.

[18] Ammann, P., Ding, W. & Xu, D., Using a model checker to test safety properties. *Proc. of the 7th Int. Conf. on Engineering of Complex Computer Systems*, IEEE: Skovde, pp. 212–221, 2001.

[19] ETCS SUBSET 026, ERTMS/ETCS System Requirements Specification (SRS) V2.3.0. http://www.era.europa.eu/Pages/Home.aspx

[20] Katsaros, P., A roadmap to electronic payment transaction guarantees and a Colored Petri Net model checking approach. *Information & Software Technology*, **51(2)**, pp. 235–257, 2009.

[21] Schulz, H.M., zu Hörste, M.M., Ptok, B., Schnieder, E., Modelling and simulation of the European Train Control System for test case generation. *Proc. of Computers in Railways VI (COMPRAIL)*, eds. B. Mellit, R.J. Hill, J. Allan, G. Scuitto & C.A. Brebbia, WIT Press: Lisbon, pp. 649–658, 1998.