PAT 기반 반례로부터 테스트 시퀀스로의 체계적 변환*

B. Zelalem Mihret○, Lingjun Liu, 지은경, 배두환
한국과학기술원
{zelalem, riensha, ekjee, bae}@se.kaist.ac.kr

A Systematic Translation from PAT-based Counterexamples to Viable Test Cases*

B. Zelalem Mihret○, Lingjun Liu, Eunkyoung Jee, Doo-Hwan Bae
School of Computing, Korea Advanced Institute of Science and Technology (KAIST)

Abstract

Model checking approach has become popular as it provides the capabilities of exhaustively exploring the state space of the modeled system and generate counterexamples for properties specified over the model. The generated counterexamples can be used to derive test cases. However, counterexamples only show states, transitions and the values of their parameters. In addition, its semantics are also dependent on model specification languages and trace representation notations. In this paper, we present a focused test case generation approach from PAT model checker. The focus is driven by specific and putative attack behaviors. To this end, we devised test case specification rules to translate counterexamples to test cases. We demonstrate our approach by using air traffic control system (ATC) with a goal of minimizing safety risks due to cyberattacks during aircraft landing operation.

1. Introduction

The task of creating test cases manually is tedious, prone to error and time-consuming. There are several different approaches to address these problems. The approaches vary depending on several factors including the distinct characteristics of application domains, system development phases at which the test is planned to be conducted, abstraction level of the system, and intended goals to be achieved by the test. Due to such complexities, to a significantly larger extent, researches both in academia and industries focus on automatic generation of test cases from certain models of a system [1].

In this paper, we present our investigation and results obtained on focused test case generation using model checking technique. The focus is driven by specific and putative attack behaviors. For example, intercepting communication between systems and modifying leaked data is one of the common attack for several systems that rely on message communications. We model such attack behaviors as part of the system under study. The modeling includes permitting attacker behavior to access local variables of the system as one of the legitimate processes, and study the combined system behaviors for specific properties (such as starvation or collision properties). This is a focused approach in that the generated counterexamples show the system behavior specifically impacted by the attacker actions. Systematically we refine the counterexamples to formulate viable test cases.

For the purpose of demonstrating the feasibility of our approach, we use a prototypical case from the aviation domain, air traffic control system (ATC). ATC is composed of several autonomous systems such as flight deck systems (FD), air traffic service provider (ATSP), control towers, different level human operators, etc. [2]. In this paper, we particularly focus on ATC constituent systems that play roles during landing operations of an aircraft. The system list includes: FD, surveillance data processing (SDP, ATC decision component), and short term conflict alert system (STCA). ATC system is huge and complex. We abstracted only a few functionalities in each constituent systems. For example, an aircraft represents request for landing permission, SDP checks the status of runway, current requests, and assign aircrafts to use runways, and STCA handles warning and alerts that may result in safety and security constraints. The three systems are networked, hence there is a risk of cyberattacks. We model the interaction behavior of the collaborating systems using CSP modeling language, and verify properties (for example detecting collision or starvation during landing) using PAT model checker [3].

PAT is a model checking tool. By design, it is not directly tailored for the purpose of test case generation. We demonstrate how to work around this by using test case specification rules that can be used to transform counterexamples into a set of actions and their sequences. With these constructing elements, we further refined it in a systematic way to formulate a concrete test case.

The remaining part of the paper is organized as follows. A survey on related works is presented in section 2. The overall approach is discussed in section 3. Section 4 presents illustrative example to show the application of the proposed approach, results obtained from the implementation, and shares lessoned learned in the process. Section 5 is devoted for conclusion and

*This research was partly supported by CybWin Project (No. 287808), Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2019R1I1A1A01062946), and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2015-0-00250, (SW Star Lab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System).

future work.

2. Related work

There are several model-based test case generation methods [4]. The distinction between them lies in, among other things, the addressed modeling paradigm, application domains, the test generation methods, and the supported coverage criteria. Model checking is one of the techniques considered for test case generation [5] [6].

Despite the similarities in the technique, which is model checking, there is major difference in the way we consider input models for the model checking process. Specifically, our approach is different in that we consider the attacker behavior as input model for the model checking process. This gives chances to generate focused test cases that can address actual concerns. In this regard, we have not seen a related work that tries to exploit putative attack behavior as input model. Thus, the related works discussed below show model checking technique in general for test case generation.

S. Mohalik et al. [7] use the model checking technique to generate automatic test sequences. A test sequence represents paths in the system transition model. This sequences are used to conduct the tests. A. Armando et al. [6] deal with generation of putative attack from model checker and automatic testing on real implementations for security protocols using test execution engine. Putative attack in this work is used to represent selected and refined counterexamples based on domain expert knowledge from the counterexample pool.

3. Overall Approach

The overall approach is used to show major concepts and how the concepts are related in the processes of generating test cases from model checking. As shown in Fig. 1, an attacker behavior is considered as input model for the model checking process in our approach. This makes the approach more focused to the specific properties modeled in the attacker behavior. Major components of the overall approach are discussed as follows.

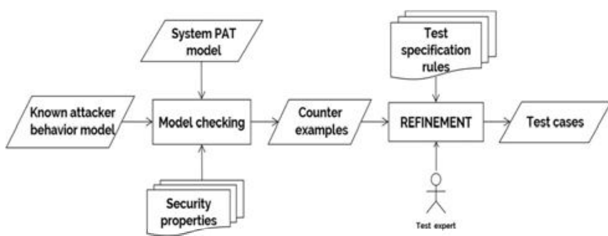

Figure 1. Overall approach

3.1. System PAT Model

PAT system model is a representation of a system behavior using a dedicated module in PAT. We use CSP module for modeling the system behavior as we are dealing with concurrent systems. The basic components of such modeling module are set of states, transitions between states, and rules of transitions. At run time, the operational semantics of the PAT model translates the behavior of the model into LTS (Labeled Transition System) which can be automatically explored by the verification algorithm [3]. One of the advantages of PAT is that it is designed with features that facilitate effective incorporation of domain knowledge with formal verification [8]. For example, complex computation and domain specific knowledge can be designed separately by using a standard programing language (for example C#) and can be integrated into the model checking process.

3.2. Putative Attack Behavior Model

Recently, information about cyberattacks including their techniques, targets and patterns are being identified and organized. For example, in the air traffic management (ATM) domain studies have identified putative cyberattacks (common to similar application domain [9]). For feasibility study, we investigate a specific and putative attack type in ATC domain.

3.3. Counterexample

A counterexample is a verification engine output that gives the simulation run leading to the state where a specified property is violated. PAT model checker has a verification engine that invokes a procedure to generate a Buchi automaton that corresponds to the negation of LTL property specification. Then the a Buchi automaton is composed with the internal model so as to determine whether the LTL formula is true for all system execution [3]. A counterexample, therefore, is generated when the search returns false.

3.4. Test Case Specification Rules

Basically a test case has components that describe inputs, actions, expected responses and explicit step by step instructions in order to determine if a system correctly deliver certain functions. In this regard, the test case specification rules we propose aims at identifying these constructs from counterexamples. The test case specification rules are set of rules to translate and refine counterexamples to test cases.

Counterexamples contains sequenced set of events that have an initial event and end events. The end event may form a loop i.e. contain more than one event. We proposed two phased step by step test case specification rules, presented as follows.

3.4.1. Test case data extraction from counterexamples

This phase is used to identify relevant events for test case synthesis, and distinguish attacker event and system responses.

(1) Step 1: Start from system event that happened immediately before the first attacker event in the counterexample, label this event as START. If the counterexample begins with an attacker event, take the first attacker event as START.

(2) Step 2: Record variables and their values both found in the START and the next event

(3) Step 3: Trace the counterexample forward skipping all system events until (3.1) a system event appears just before an attacker event, or (3.2) attacker event appears.

(4) Step 4: If (3.1) is the case from step 3, add to the record the newly discovered system event and the next attacker event parameters and their values, then repeat step 3. If (3.2) is the case from step 3, add to the record variables and their values of the attacker event.

(5) Step 5: Repeat step 3 and 4 until there is no more event to explore, or an event or set of events that create loops. Label the last event or set of events as END.

Between START and END events, collect all variables (local and global) that show change in their values at least once. These variable can be used as test inputs both for attacker events and system responses. All other variables and their values will be used to maintain sequence of events.

Fig. 2 shows counterexample system and attacker event transitions. The transition from START to END should satisfy the following two conditions to be considered for test case specification. First, a minimum one transition should exist from system event to attacker event, if START is a system event. Similarly, a minimum one transition from attacker event to system event if START is an attacker event. Second, a specific END event can be at system event or attacker event.

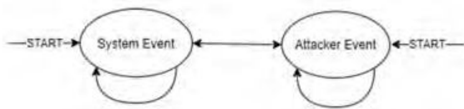

Figure 2. Counterexample event transition diagram

3.4.2. Test case synthesis

Test case synthesis systematically analyze the test case data record to drive test cases. The synthesis decides preconditions, test actions, test inputs and expected responses. It analyses the test data record based on the following steps:

(1) If START is an attack event, no precondition required to start test, otherwise the starting system event will be precondition to start test.
(2) Event's values that change at least once between START and END is used to construct test inputs. The END event represents the test case result.

A summarized form of a test case extracted from a counterexample is shown as follows.

Table 1.Test case components description

Counterexample data record	<current-event>
Precondition	Global or local variables and their values
Test inputs	Parameter values that have been changed if <current-event> is attacker event
System response	System state values if system event follows from <current-event> If <current-event> is not END, then it will be precondition for next event

4. Case study

To illustrate the feasibility of our proposed approach, we selected a small, but complete and non-trivial practical scenario from ATC. Ensuring safety during landing operations is one of the critical concerns in ATC. A test case that can be used to check that collaborating constituent systems properly function together and achieve safety goals such as no collision or starvation is one of the demanding requirements.

4.1. System PAT model

The following code snippets is taken from CSP models representing the three constituent systems: FD, SDP, and STCA. The model represent mainly processes and decisions while collaborating each other.

```
FD(i) = [rqstChnn[i] == notmade && rspnsChnn[i] !=
granted && ntfChnn[i] ==
ready]AIRCRAFT_makeRQST.i{ rqstChnn[i] = rqst}->
FD(i) [] [(ntfChnn[i] == ready || ntfChnn[i] ==
released) && alertsignal[i] == GREEN &&
rqstChnn[i] !=
notmade ]AIRCRAFT_landing.i{ ntfChnn[i] =  inaction}-
> FD(i) ...
SDP() = [state == listening && flg ==
DOWN ]ATC_checkrequest{var i = 0; var rqcntr = 0; var
gcntr = 0;  while(i < numberOfAirCraft){
if(rspnsChnn[i] == granted ){ gcntr = gcntr + 1;};
i++}; i = 0; while(i < numberOfAirCraft){
if(ntfChnn[i] == released){rspnsChnn[i] = ...
undecided };  i++}; ...
STCA() = [flg == UP && stat == 1]STCA_notifyalert{var
i = 0;  while(i < numberOfAirCraft){ if(rspnsChnn[i]
== granted){ alertsignal[i] = GREEN; stat = 0}
else{alertsignal[i] = RED}; i++};} -> STCA()
[] [stat == -1]STCA_resetalert{var i = 0; var cntr =
0; ...
```

As stated in the introduction and overall approach section, our goal is a focused test case generation driven by specific and putative attack. We model an intruder by considering a putative attack behavior in ATC domain. Communication links between collaborating system such as STCA and aircraft system are targets for cybersecurity attack. For this feasibility study, we consider an eavesdrops attack that aims at altering notification messages. A snippet showing the intruder behavior model is included as follow.

```
INTRDR() = [force == 0 && flg == UP]ATTCK_listen{var
i = 1; CNTR = 0;
if(alertsignal[0] == RED ){CNTR = 1};
if(CNTR == 1) {force = 1} else {force = 2}} ->
INTRDR() // evasdrope
[][force == 1]ATTCK_mdfy{alertsignal[0] = GREEN;}->
Skip();...
```

4.2. Property verification

One of the important safety properties in ATC domain, particularly during landing operation, is guarantying non-existence of collision i.e. more than one aircraft should not be landing on a run way simultaneously. We specified this property as follows for three FDs requesting for landing.

```
// check collision problem
#define collision (ntfChnn[0] == inaction &&
ntfChnn[1] == inaction ) ||(ntfChnn[0] == inaction &&
ntfChnn[2] == inaction)||(ntfChnn[2] == inaction &&
ntfChnn[1] == inaction);
```

4.3. Test case generation

PAT supports multiple verification options including first witness trace using DFS, shortest witness trace using BFS, and others [3]. PAT generates the counterexample events along with the variables and their values. The verification result from PAT model checker is shown in Fig 3.

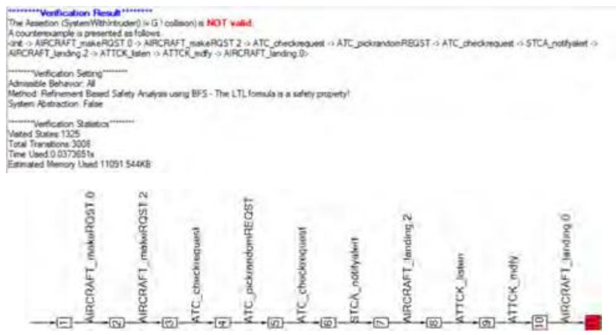

Figure 3. PAT model checker counterexample

By following the test case specification rules (test data extraction steps), we populate test case data record. The test synthesis process, finally, produces the test case. Table 2. shows the test case generated from the counterexample.

Table 2. Generated test case

Counterexample data	Precondition	Test inputs	System response
AIRCRAFT_landing.2	ntfChnn=[ready, ready, inaction]; alertsignal=[RED, RED, GREEN]; rspnsChnn=[undecided, undecided, granted]; rqstChnn=[rqst, notmade, rqst];	not applicable	none
ATTCK_listen	none	none	not applicable
ATTCK_mdfy	none	alertsignal=[GREEN, -, -];	not applicable
AIRCRAFT_landing.0	none	not applicable	ntfChnn=[inaction, ready, inaction]; alertsignal=[GREEN, RED, GREEN]; rspnsChnn=[undecided, undecided, granted]; rqstChnn=[rqst, notmade, rqst];

4.4. Result analysis

Using the generated counterexamples and the test case specification rules described, we showed how we develop a test case for aircraft landing operation. The two most popular challenges of using model checking technique for test case generation are (1) the quality of a model checking result is entirely dependent on the quality of input models, hence test case quality inherits similar limitations by association, and (2) counterexample generated based on test cases don't always map to real case concerns. This is mainly because counterexamples are results of exhaustive search for unsatisfied conditions in the state space without any bound as such for soundness or closeness check with respect to actual concerns. Our approach has constituents to attest for real case specific attack concerns because putative attack behaviors are constructed from real case reports.

Despite an attacker behavior is introduced into the system model, there is no direct way to make conclusion about which properties of the system can be violated. Therefore, the counterexamples can reveal what can go wrong even without a putative attack included in the model. In addition to this, a test case expert can use the generated test cases for a focused vulnerability analysis.

5. Conclusion and Future work

A viable test case can be synthesized systematically from counterexamples generated using PAT model checker. In this paper, we present a feasibility study on the possibility of translating PAT-based counterexamples into viable test cases that can be used for focused system behavior analysis, particularly attacker driven. Despite the challenges and limitations that exist in the model checking approach in general, and the modeling of an attacker behavior in particular; it can be seen that viable test cases can be synthesized that will reduce test expert's effort to convert it into a concrete test cases. Automating the translation process is our future work. This work can be extended to address vulnerability analysis of complex systems, and also runtime monitoring using the counterexamples as test sequences.

References

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," in *Journal of Systems and Software*, 2013.

[2] G. Carrozza, M. Faella, F. Fucci, R. Pietrantuono and S. Russo, "Engineering air traffic control systems with a model-driven approach," in *IEEE software*, 2013.

[3] J. Sun, Y. Liu, J. S. Dong and J. Pang, "PAT: Towards flexible verification under fairness," in *Springer*, 2009.

[4] W. Li, F. L. Gall and N. Spaseski, "A Survey on Model-Based Testing Tools for Test Case Generation," in *Springer International Publishing*, 2018.

[5] O. Tkachuk, M. B. Dwyer and C. S. Pasareanu, "Automated environment generation for software model checking," in *IEEE*, 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings, 2003.

[6] A. Armando, G. Pellegrino, R. Carbone, A. Merlo and D. Balzarotti, "From model-checking to automated testing of security protocols: Bridging the gap," in *International Conference on Tests and Proofs*, Springer, 2012.

[7] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. Shashidhar and S. Ramesh, "Automatic test case generation from Simulink/Stateflow models using model checking," in *Software Testing, Verification and Reliability*, Wiley Online Library, 2014.

[8] Hoare and CAR, "Communicating Sequential Processes Prentice Hall Int." London, 1985.

[9] E. P. Enoiu, A. Causevic, T. J. Ostrand, E. J. Weyuker, D. Sundmark and P. Pettersson, "Automated test generation using model checking: an industrial evaluation," in *International Journal on Software Tools for Technology Transfer*, Springer, 2016.

[10] M. E. Ruse, "Model checking techniques for vulnerability analysis of Web applications," 2013.

[11] E. Harison and N. Zaidenberg, "Survey of Cyber Threats in Air Traffic Control and Aircraft Communications Systems," in *Springer*, 2018.

[12] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala and R. Majumdar, "Generating tests from counterexamples," in *IEEE*, 2004.