2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) | 978-1-6654-4553-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/EnCyCriS52570.2021.00008

Attack-driven Test Case Generation Approach using Model-checking Technique for Collaborating Systems

Zelalem Mihret and Lingjun Liu
Software Engineering Lab (SELab), School of Computing
Korea Advanced Institute of Science and Technology (KAIST), Republic of Korea
{zelalem, riensha}@se.kaist.ac.kr

***Abstract*—The formal verification technique of model-checking can be used to derive test cases. This approach has become popular as it provides the capabilities of exhaustively exploring the state space of the modeled system and generates counterexamples for properties specified over the model. However, counterexamples only show states, transitions and the values of their parameters. In addition, its semantics are also dependent on input model specification languages and trace representation notations. In this paper, we present a focused test case generation approach from PAT model checker for collaborating systems. The focus is driven by specific and putative attack behaviours. To this end, we devised test specification rules/algorithm to translate counterexamples to test cases. The translation aims at reducing semantic gaps between counterexamples and the corresponding test cases. We assess the viability of the test cases generated from our approach by using JADE simulation framework for aircraft landing scenario in air traffic control domain.**

***Index Terms*—Attack-driven test case generation, Model checking for test case generation, Test case generation for system of systems,**

I. Introduction

The task of creating test cases manually is tedious, prone to error and time-consuming, to mention few of the challenges. 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]. Recently, it has become a trend to see the use of model-checking supporting the discovery of vulnerabilities in deployed systems, in addition to the common use at design time [2].

Essentially, model-checking is a verification technique that takes a formal system and desired property specifications as input, and generates verdicts concerning whether the system satisfies the property [3]. It is used to establish that the design or product under consideration possesses certain properties. In addition, model-checking is mainly used to verify properties of software systems i.e. systems having specific goals and highly coupled components (process-level interactions). Test case generation from a model-checking technique, however, is an approach that aims at extracting test case information from counterexamples. In this regards, the model-checking processes need to be guided to emulate test cases from counterexamples. This may require a systematic interpretation of the model-checking outputs and/or adding control inputs to dictate the model-checking processes towards to generating test cases.

In this paper, we present our investigation and results obtained from focused test case generation using model-checking technique for collaborating systems. Collaborating systems interact through message exchanges, unlike monolithic systems that depend on direct interactions (use stored knowledge or reference to past states and their values, and communicate mainly through method invoking). The focused test case generation is driven by specific and putative cyberattack behaviors. Recently, information about cyberattacks including their techniques, targets, and patterns is being identified and organized, which can be used to form putative behaviour. For example, in the air traffic management (ATM) domain studies have identified putative cyberattacks [4]. Commonly, test cases for functional testing are generated from domain models, environment models, or system behaviour models [5], [6]. This is mainly because the mentioned models provide certain information about the system under test. For the same token, the use of putative attack behaviour can be considered as another model for test case for functional testing.

Intercepting communications between systems and modifying leaked data is one of the common attacks targeting collaborating systems. We model such attacker behaviours, incorporate it with system behaviour model, and study the combined system behaviour for specific properties (such as starvation or collision properties) using model-checking technique. When we model the attacker behaviour we don't consider how the attacker presents itself in the system, but what it manipulates, alters, or introduces into the system. Our approach is a focused test case generation 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 as demonstrated in this paper.

For the purpose of demonstrating our approach, we use

978-1-6654-4553-5/21/$31.00 ©2021 IEEE
DOI 10.1109/EnCyCriS52570.2021.00008

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on August 10,2021 at 04:15:03 UTC from IEEE Xplore. Restrictions apply.

a prototypical case study from ATM, particularly air traffic control (ATC), a specific case of aircraft landing operation. We model the interaction behaviour of selected collaborating systems using communicating sequential processes (CSP) modeling language and verify properties using the PAT model checker [7].

Testing involves running specified test cases against the system under test. To evaluate the test case generated from our approach, we develop a simulation program using JADE [8]. The simulation imitates the interactions of three relevant systems in ATM domain particularly for landing operation: flight deck (FD), surveillance data processing (SDP), and short term conflict alert (STCA).

Our contribution can be summarized as follows:

- A focused test case generation approach based on specific and putative cyberattack behaviours.
- We use model-checking technique to generate test cases for collaborating systems. Model checking techniques are used mainly for monolithic systems which have highly coupled components.
- We introduce test case specification rules (algorithm) that can be used to produce concrete test cases from counterexamples. It is obvious that there are semantic gaps between counterexamples (result of static verification), and concrete test cases (input to dynamic verification).

The remaining part of the paper is organized as follows. Background information about major concepts is discussed in section II. The overall approach is presented in section III. An illustrative example to elaborate on the feasibility of the proposed approach is covered in section IV. Section V presents evaluation, where we offer some lessons learned as we assess and evaluate the proposed approach. Section VI surveys related work. Finally, conclusion including future work is presented in section VII.

II. Background

In this section, we provide some background details on PAT model checker, collaborating behaviors of ATC constituent systems, and JADE tool for simulation of the prototypical case studies.

A. PAT model-checker

PAT is a model-checking tool, extensible to support complex system behavior modeling by adding external program module such as C# library. It is a self-contained tool that supports composing, simulating, and reasoning about concurrent, probabilistic, and times system with non-deterministic behaviour [7]. It is a free software as well. Using the reasoning capability, it can be used to verify system properties. We use the generic and extensible feature of PAT to incorporate domain knowledge into formal verification using model-checking [9]. By design, PAT is not directly tailored for the purpose of test case generation. We demonstrate how to work around this by using test specification rules that can be used to transform counterexamples into test cases.

PAT system architecture has four dimensions: modeling, abstraction, intermediate, and analysis. Modeling is used for user-friendly editing environment serving different purposes such as concurrent and real-time modules. The abstraction dimension provides capabilities for domain specific linkage that can be applied for operational semantics and symbolic encoding. Different features of simulating modeled behaviour are handled by the intermediate modules. Analysis combines properties such as checking, generating counterexamples, and improving by refinements.

B. Air Traffic Control

ATC is a service provided by ground-based controller which directs aircraft on the ground and in the air. Its primary task, as scaled down for analysis purpose, can be stated as keeping aircraft at a safe distance from each other horizontally and/or vertically (ICAO [1]). It is also used to ensure orderly flow of traffic and providing information to pilots, such as radar traffic advisories, weather advisories, flight flowing, and navigation information.

Architecture level, there can be differences in how ATC constituent systems are structured and composed. However, our analysis focuses on aircraft landing, and hence we reduced interacting systems and abstracted their functionalities to manage the complexity. Fig. 1 shows the collaboration diagram of the three constituent systems considered in our model: SDP, FD, and STCA [10].

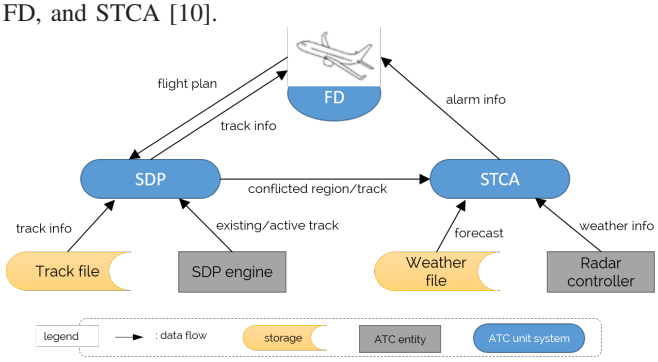

Fig. 1: System functional model for landing operation

Mainly, SDP serves requests from FDs, and SDP receives requests from FDs and validates requests. STCA makes sure constraints are satisfied.

C. JADE simulation framework

Agent-based simulation is often used by researchers in air traffic control systems to analyze system performance and simulate multiple scenarios [11]. Corresponding to our constituent systems, agents are autonomous, social, and interacts with each other to achieve their individual goals or shared goal by deriving strategies.

JADE is a well-known software framework to simplify the development of applications for interoperable intelligent multi-agent systems [8]. The agent platform is a distributed system since agents and resources can be distributed over different hosts in the fixed network or mobile terminals. Each aircraft is mapped to one agent along with SDP and STCA are

[1] ICAO Doc 4444 PANS-ATM

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on August 10,2021 at 04:15:03 UTC from IEEE Xplore. Restrictions apply.

represented as separate agents. JADE satisfied the capabilities of agents, including autonomy and negotiation (or sociality).

III. Overall Approach

The overall approach is used to show major concepts and how the concepts are related to the processes of generating test cases from model-checking outputs. Broadly, we classify the overall approach into two based on operation focuses, (1) test case development where processes mainly focus on generating test cases, and (2) test case evaluation focusing on testing the viability of test cases using the simulation program. The schematic representation of our approach is shown in Fig. 2. Each component of the overall approach is discussed as follows.

A. System PAT model

PAT system model is a representation of a system behaviour using a dedicated module created in PAT. We use CSP module for modeling the system behaviour as we are dealing with concurrent systems. The basic components of such a modeling module are a set of states, transitions between states, and rules of transitions. At run time, the operational semantics of the PAT model translates the behaviour of the model into LTS (Labeled Transition System) which can be automatically explored by the verification algorithm [7], [9].

B. Property specification

PAT supports most useful assertions including deadlock checking, reachability checking, LTL checking, and refinement checking. Several security properties can be checked using PAT such as secrecy, authentication, non-repudiation, anonymity, fairness, integrity, and temporal specification [7]. In order to utilize these features, properties need to be specified correctly according to PAT formalities. We specify safety properties such as collision and starvation that can be related directly or indirectly to security flaws.

C. Putative attack behaviour model

Recently, information about cyberattacks including their techniques, targets, and patterns is being identified and organized, which can form a common class called putative attacks. One of the distinguishing features of our approach is the inclusion of such attack behaviour as an input model to the model checker. By investigating putative attacks in the specific domain (for example in the case of ATC, survey is presented in [4]), we model specific and putative attacker behaviour and incorporate it as an input model to the model-checking process. This enables to generate attacker driven counterexamples. Such counterexamples closely resemble how systems react to intruders, which makes their use more efficient to develop focused test cases for specific vulnerability analysis. PAT is extensible and hence can facilitate effective incorporation of domain knowledge with formal verification II.

D. Counterexamples

A counterexample is a verification engine output (a property against a system model) that gives the simulation run leading to the state where the specified property is violated. For this purpose, the PAT model checker has a verification engine that invokes a procedure to generate a Buchi automaton that corresponds to the negation of LTL (linear temporal logic) property specification. Then the Buchi automaton is composed of the internal model so as to determine whether the formula is true for all system execution [12]. A counterexample is generated when the search returns false. One can then extract test information from the trace, and refine it based on test case specification rules to fit a test case as we have demonstrated in this paper.

E. Test case specification rules

Basically, a test case has components that describe input, action, expected response, 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 inputs, actions and their sequences, and expected responses from counterexamples. The test case specification rules are a set of action guides to translate and refine counterexamples to test cases.

A counterexample contains a sequenced set of events that have an initial state and end states. The end states may form a loop. We proposed two-phased step by step test case specification rules, presented as follows.

1) Test case data extraction: The following are steps to extract test case data from counterexample

- Step 1: Start from system event that happened immediately before the first attacker event in the counterexample, label this event as START. If counterexamples begin with an attacker event, take the first attacker event as START.
- Step 2: Record parameters and their values both found in the START and the next event.
- 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) an attacker event appears.
- 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 parameters and their values of the attacker event.
- 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 values at least once. These variables can be used as test input data. All other variables and their values will be used to maintain the sequence of events. The event sequences as it appears in the counterexample (START $-$ $>$ END) will be used as the instruction set for testing i.e. attacker event as test agent/expert input actions and other events as system responses along with the associated variables (identified in step 3).

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on August 10,2021 at 04:15:03 UTC from IEEE Xplore. Restrictions apply.

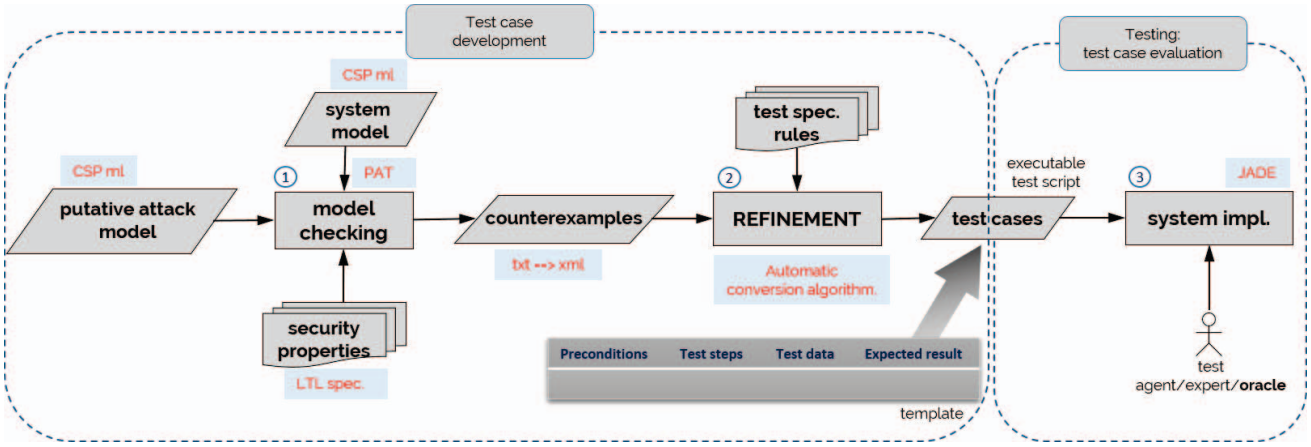

Fig. 2: Overall approach

Fig.3 shows counterexample transitions between system events and attacker events. The transition from START to END should satisfy the following two conditions to be considered for test case specification.

- 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.
- END can be at system event or attacker event.

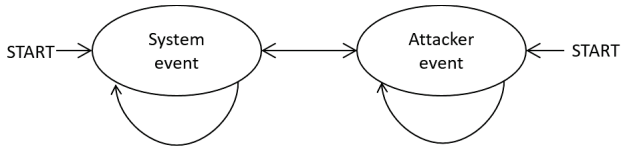

Fig. 3: Counterexample abstracted event transition diagram

Test specification rules can be easily converted to an algorithm that enables an automatic translation of counterexamples to test cases, as shown below:

Algorithm 1: Algorithm to convert a counter example to a test case

Result: Test cases as specified in the template
Input: counterexample, attacker's actionlist = { AtA0,AtA1,– AtAi –AtAn } ;
Initialize: param[] = null, cnt[action, param[]] = null ;
1. Start tracing from (init);
2. Advance search. If current state is AtAi , go to previous state PrvState; else repeat (2) if current state is not END ;
3. Add PrvState and AtAi to cnt ;
4. Advance search. if AtAi is followed by another AtAj, add AtAj to cnt, repeat (4) if current state is not END; else repeat (2 - 4) until END;
5. Add END to cnt ;

2) Test case synthesis: Test case synthesis systematically analyzes the test case record populated during test case data extraction to drive test case based on test case templates. A test case template gives the schema of test cases. The general form, as depicted in the overall approach diagram, is composed of preconditions, test steps, test data, and expected results. The synthesis steps are used to determine the test case template contents.

- If START is an attack event (usually next to *init*), no precondition is required, otherwise, the starting system event will be the precondition to start the test.
- Parameters values that change at least once between START and END (excluding END, END will be the expected result of the test case) will be used for test inputs.

A summarized form of test cases extracted from a counterexample is shown in the following Table IV.

TABLE I: Test case template components and their description

Counterexample	Current-event
Precondition	Global and local variables and their values
Test inputs	Parameter values that have been changed if Current-event is attacker event
Expected responses	System state parameters values if system event follows from Current-event If Current-event is not **END**, then it will be precondition for next event

IV. Case Study

A. Case scenario design

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.

The collaborating constituent systems for our prototypical case study include FD, SDP, and STCA. FDs communicate with SDP by sending different messages including RQST messages to submit a request for landing (flight plan), ACK messages to notify that it receives responses successfully, and RELEASE messages to inform that its landing operation is completed. SDP listens to messages coming from FDs, checks the status of the airport it covers, and responds accordingly to the messages. Depending on several factors including runway status, FDs' messages queues, expected action reports, and

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on August 10,2021 at 04:15:03 UTC from IEEE Xplore. Restrictions apply.

maintaining fair share to each FD, SDP's response to landing requests can be WAITING or CLEARED. STCA plays a middle role between FD and SDP. SDP informs its decision about granted permission to STCA. STCA broadcasts the granted permission after checking the safety and security constraints, for example, checking that there is no more than one FDs advancing to landing operation (safety requirements). In other words, FD starts landing operation when STCA broadcasts the granted permission, and FD receives the message in addition to SDP's permission grant.

Fig. 4 represents collaborating constituent systems in the form of flow diagrams, (a) depicts SDP, (b) FD, and (c) STCA. It represents the ATC system landing operation scaling down its complex functionalities to fit the goal of this paper.

As discussed in the overall approach, there are three input models for the model-checking process: system behaviour model, putative attack behaviour model, and security property specification. We directly convert the flow diagrams that show the collaborating system behaviour into CSP PAT model checker input. The following code snippets are taken from CSP models representing the collaborating system behaviours.

```
%%%%%
FD(i) = [rqstChnn[i] == notmade &&
    rspnsChnn[i] != granted && ntfChnn[i]
    == ready] AIRCRAFT_makeRQST.i{
    rqstChnn[i] = rqst}-> AirCraft(i)
  [] [(ntfChnn[i] == ready || ntfChnn[i] ==
      released) && alertsignal[i] == GREEN
      &&   rqstChnn[i] != notmade]
     AIRCRAFT_landing.i{ ntfChnn[i] =
     inaction}-> AirCraft(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
          .......
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()
           .......
```

B. Modeling case scenario input models

We also develop a CSP model of an intruder behaviour by considering a putative attack in ATC domain. Communication links between collaborating system such as STCA and FD are targets for cyberattacks. For this feasibility study, we consider an eavesdrops attack that aims at altering notification messages a STCA broadcasts. A snippet showing the intruder behaviour CSP model is shown as follows.

```
%%%%%
INTRDR() = [force == 0]ATTCK_listen{var i
    = 1; CNTR = 0; if(alertsignal[0] ==
    RED ){CNTR = 1};if(CNTR == 1) {force =
```

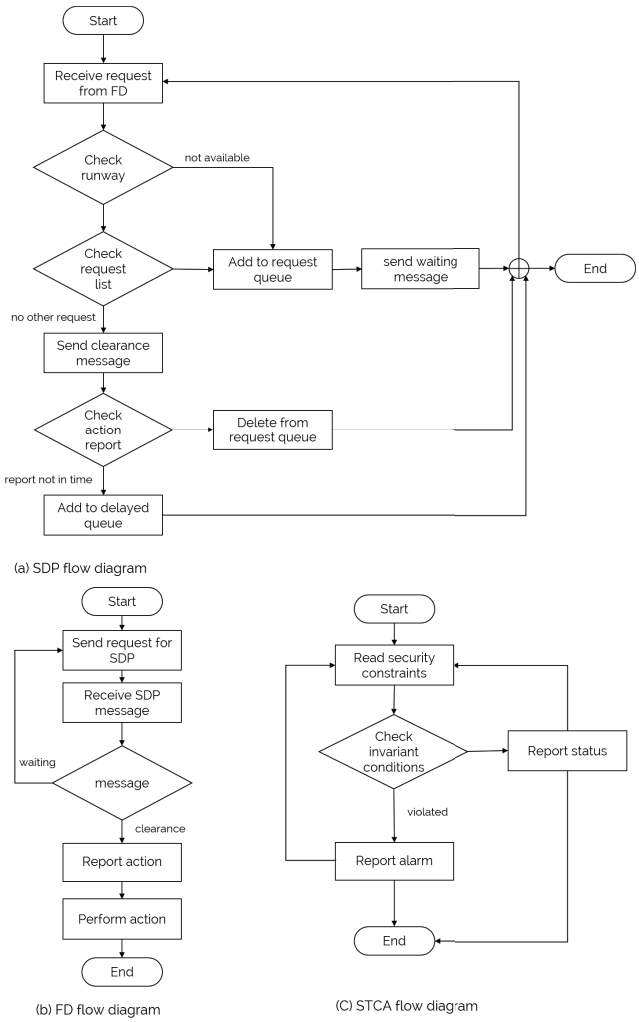

Fig. 4: Constituent systems interaction flow diagram

```
    1} else {force = 2}} -> INTRDR() /*
   evasdrope */
        .......
```

One of the important safety properties in ATC domain, particularly during landing operation phases, is guarantying non-existence of collision i.e more than one aircraft should not be landing on a runway or allowed to use the same trajectory for departure simultaneously.

We specify the safety 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);
```

V. Evaluation and Discussion

We divide this section into three parts: test case result analysis, testing using simulation, and discussion. The first

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on August 10,2021 at 04:15:03 UTC from IEEE Xplore. Restrictions apply.

subsection presents the test cases developed from the PAT model checker using our systematic approach. The second subsection focuses on viability testing of the test cases using JADE simulation. We put our reflection on the overall challenges, limitations, and opportunities of the proposed approach in the discussion part.

A. Test case generation results

As stated in the introduction and overall approach sections, our goal is a focused test case generation driven by a specific and putative attack model using a model-checking technique. The specific attack behaviour we considered in this paper is eavesdrop attack aiming at altering leaked messages contents(a snippets code is included in subsection IV-B. We run PAT verification engine to examine the system model behaviour for two different cases. In the first case, the attacker behaviour is instrumented to run along with the collaborating constituent systems. In this case, the attacker sniffs communication between a specific FD and STCA. It can also alter the variables' value in the communication channel. In the second case, we execute the verification engine excluding the attack behaviour model. By doing so, we make sure that our system model design satisfies the safety requirements.

```
%%%%%
/*Two system model inputs .*/
System = (||| n:{0 .. numberOfAirCraft - 1}
      @ FD(n)) ||| SDP ||| STCA;
SystemWithIntruder = (||| n:{0 ..
     numberOfAirCraft - 1} @ FD(n)) ||| SDP
     ||| STCA ||| INTRDR;
```

The collision-free safety property should be preserved in our system. In the case of an intruder introduced to our system, if the property is violated, the PAT model checker produces a counterexample. Using this counterexample and the test case specification rules we proposed in this paper, we show how we develop a test case from the counterexample.

The verification result from the PAT model checker is shown Fig. 5.

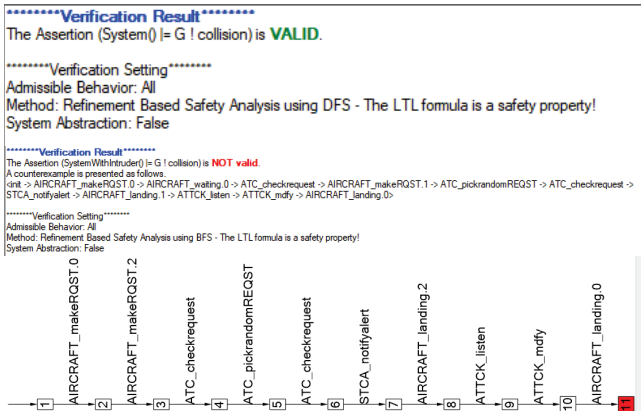

Fig. 5: PAT model checker verification result

Using the proposed algorithm that converts a counterexample to a test case (mainly focusing on test case preconditions and action sequences), we identify the following events that will be considered in test case synthesis.

TABLE II: Test case data extracted from counterexample

Counterexample states	Variables and values
AIRCRAFTlanding.2 (event-1)	ntfChnn=[ready,ready,inaction]; alertsignal=[RED,RED,GREEN]; rspnsChnn=[undecided,undecided,granted]; rqstChnn=[rqst,notmade,rqst];
ATTCKlisten (event-2)	ntfChnn=[ready,ready,inaction]; alertsignal=[RED,RED,GREEN]; rspnsChnn=[undecided,undecided,granted]; rqstChnn=[rqst,notmade,rqst];
ATTCKmdfy (event-3)	ntfChnn=[ready,ready,inaction]; alertsignal=[GREEN,RED,GREEN]; rspnsChnn=[undecided,undecided,granted]; rqstChnn=[rqst,notmade,rqst];
AIRCRAFTlanding.0 (event-4)	ntfChnn=[inaction,ready,inaction]; alertsignal=[GREEN,RED,GREEN]; rspnsChnn=[undecided,undecided,granted]; rqstChnn=[rqst,notmade,rqst];

The test case synthesis process, finally, produces the test case by extracting test data from the identified counterexample events. The test case synthesis determines test inputs by analyzing identified counterexample events' parameter values. If the parameter values change at least once between START and END, they become candidate test inputs. The test inputs are used to prompt system responses. For example, as can be seen in table II, *alertsignal* first value from ATTCKlisten event changes from *RED* to *GREEN* in the ATTCKmdfy event, and *ntfChnn* first value from ATTCKmdfy event changes from *ready* to *inaction* in the AIRCRAFTlanding.0 event. As a result, we consider *alertsignal* parameter as test input as it prompts system responses. We can also observe that the change in values of *ntfChnn* parameter is caused by attacker actions. This can be related to the reasons for changes in *ntfChnn* parameters. The goal in test case generation is to decide the required test inputs that prompt system response. The *alertsignal* values are the ones manipulated by the attacker action, and hence considered as test input. Table III shows the counterexample mapping to a test case resulted from the translation algorithm.

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

B. Evaluation using simulation

Testing involves running specified test cases against the system under test. For this purpose, we develop a simulation program using JADE for aircraft landing operations. The simulation imitates the interactions through messaging of the three relevant systems in ATC domain: FD, SDP, and STCA. A test agent system that mimics the task of a test expert observes the system behaviour using logs, and inputs test data as prescribed in the test case when preconditions are satisfied. Then, observes the system behaviour if the system responds

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on August 10,2021 at 04:15:03 UTC from IEEE Xplore. Restrictions apply.

TABLE III: A counter example mapping to a test case

	Test case		
Events	**Precondition**	**Test input**	**Expected response**
event-1	ntfChnn= [ready,ready, inaction]; alertsignal= [RED,RED,GREEN]; rspnsChnn= [undecided,undecided, granted]; rqstChnn= [rqst,notmade,rqst];	not applicable	none
event-2	none	none	not applicable
event-3	none	alertsignal= [GREEN,-,-];	not applicable
event-4	none	not applicable	ntfChnn= [inaction,ready,inaction]; alertsignal= [GREEN,RED,GREEN]; rspnsChnn= [undecided,undecided, granted]; rqstChnn= [rqst,notmade,rqst];

as predicted in the test case. Figure 6 shows the design of our evaluation approach.

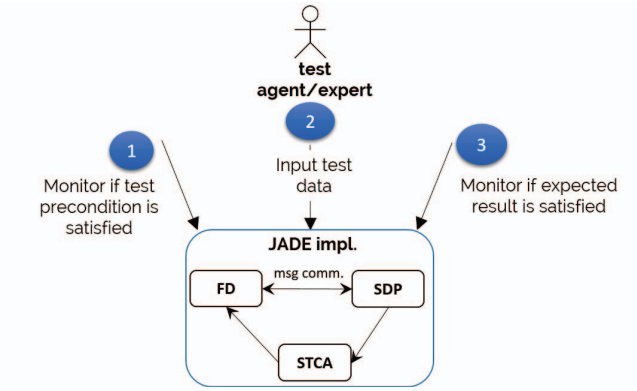

Fig. 6: Evaluation approach and design

In the simulation, we include three FDs, an SDP, and STCA. We run the simulation until all FDs' requests for landing are satisfied. We keep interaction logs that can be analyzed by respective systems to make decisions. The test agent also relies on the interaction logs to follow up the test case conditions.

To assess the effectiveness of our approach, we utilize test cases to test two variants of the system under consideration (landing operation in ATC systems). One ATC system implements an authentication mechanism between STCA and FD. The other one does not implement the mechanism so that all FDs receive unattested messages but respond based on the message content.

Due to the intruder's abilities mentioned in SectionIV-B, the test agent sniffs the messages sent from STCA. In addition, the test agent checks if the precondition is satisfied. Based on the test case data shown in TableIII, the precondition here represents that STCA broadcasts to all FDs that only *FD2* can land. Once the precondition is satisfied, the test agent starts broadcasting to all FDs the altered message. The altered message is to inform all FDs that *FD0* can land. The simulation results of vulnerable system are represented in the Table IV. Event 13 to 16 depict the regular event sequence during an aircraft (*FD8*) landing operation. In Event 17, *FD2* was granted clearance from SDP, and SDP infromed STCA. When Event 18 happened, the test agent observed that the precondition is satisfied and began broadcasting the altered messages to all FDs. Observed from Event 30 to 36, it is shown that *FD0* landed first before *FD2* finished landing operation in the system without an authentication mechanism.

TABLE IV: Interaction log of ATC system with vulnerability

Event id	**Sender**	**Receiver**	**Message**
...	...	...	...
13	FD8	SDP	Landing on progress
14	STCA	Broadcast	FD8 GREEN
15	STCA	Broadcast	FD8 GREEN
16	FD8	SDP	Released
17	SDP	STCA	FD2 granted
18	STCA	Broadcast	FD2 GREEN
...	...	...	...
30	FD2	SDP	Landing on progress
31	SDP	STCA	FD3 granted
32	FD0	SDP	Landing on progress
33	STCA	Broadcast	FD2 GREEN
34	STCA	Broadcast	FD2 GREEN
35	STCA	Broadcast	FD2 GREEN
36	FD2	SDP	Released
...	...	...	...

The result reveals that a security flaw which is unattested communication was detected by using our test case. This security flaw can be a source of safety property violation. However, there is no direct relation between the attack behaviour modeled and the security flaws detected.

C. Discussion

In general, the model-checking technique for test case generation has been in use for different application domains. The two most popular challenges of using model-checking technique for test case generation are (1) the quality of the model-checking result is dependant on the quality of the input models, hence test case quality inherits similar limitations by association, and (2) counterexample based 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.

Collaborating systems communicate primarily through message passing instead of method invocation, so existing object-oriented testing approach are not directly applicable. Testing collaborating systems is more challenging compared to testing traditional software systems because by design each constituent system has different reasons for engaging in proactive behaviour. In our experiment, one of the common challenges both in PAT model-checking and JADE simulation is the non-deterministic behaviours and asynchronous operations of constituent systems.

Concerning our knowledge, there are limited research works that focused on test case generation for collaborating systems, and putative attack behaviour as an input model for model-checking is a newly proposed idea in this paper. A focused

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on August 10,2021 at 04:15:03 UTC from IEEE Xplore. Restrictions apply.

test case generated from a model-checking technique driven by putative attack behaviour has constituents to attest for real case specific attack concerns, unlike the exhaustive search for unsatisfied conditions. Size of the illustrative example in terms of number of functionalities and complexity of the functionalities, experts role in test case refinement, and lack of test coverage, scope, and assessment can be threats to validity to the results obtained in our experiment.

VI. Related Work

There are several model-based test case generation methods [13], [14]. Model checking is one of the techniques considered for test case generation. Compared to other approaches, our approach fundamentally differs on varieties of input models considered for the model-checking process. We have not seen a related work that tries to exploit putative attacker behaviour as input model. Thus, the related works presented here show the use of model-checking technique for test case generation used in a broader sense.

S. Mohalik et al. [15] use the model-checking technique to generate automatic test sequences. A test sequence represents paths in the system transition model. These sequences are used to conduct the tests. Particularly, the work focuses on the development of automatic test sequence generation tools for SimuLink/StateFlow (SL/SF models) based on model-checking-based techniques. The proposed tool, AutoMOTGen, entails an automatic translation of SL/SF to SAL showing the instrumentation of the coverage specification.

A. Armando et al. [16] deal with the generation of putative attacks from the model checker and automatic testing on real implementations for security protocols using a test execution engine. The putative attack in this work is used to represent selected and refined counterexamples based on domain expert knowledge from the counterexample pool. It proposes a technique to automatically fill the gap between protocol specification and attack traces (counterexamples) generated from model-checking tools.

Enoiu et al [17] developed a toolbox, named COMPLETETEST. The toolbox is used to generate tests that cover the logical structure of FBD programs by transforming them first to networks of timed automata. It uses the UPPAAL model checker to generate test traces.

VII. Conclusion

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. We considered a putative attack behaviour model as an input model to the model-checking process. 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 efforts to convert them into concrete test cases. Automating the proposed approach from modeling the input models to model-checking to translating the counterexamples to test cases is our future work. This work can be extended to address vulnerability analysis of collaborating systems by using counterexamples for runtime monitoring.

Acknowledgement

This research was partly supported by CybWin Project (No. 287808), MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2020-2020-0-01795) and (No. 2015-0-00250, (SW StarLab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System) supervised by the IITP(Institute of Information & Communications Technology Planning & Evaluation).

References

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

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

[3] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[4] E. Harison and N. Zaidenberg, "Survey of cyber threats in air traffic control and aircraft communications systems," in *Cyber Security: Power and Technology*. Springer, 2018, pp. 199–217.

[5] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley Online Library, 2004, vol. 2.

[6] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[7] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "Pat: Towards flexible verification under fairness," in *International conference on computer aided verification*. Springer, 2009, pp. 709–714.

[8] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing multi-agent systems with JADE*. John Wiley & Sons, 2007, vol. 7.

[9] Y. Liu, J. Sun, and J. S. Dong, "Developing model checkers using pat," in *International symposium on automated technology for verification and Analysis*. Springer, 2010, pp. 371–377.

[10] P. Eurocontrol and A. FAA, "Comparison of air traffic management-related operational performance: Us/europe," 2013.

[11] A. Pellegrini, P. Di Sanzo, B. Bevilacqua, G. Duca, D. Pascarella, R. Palumbo, J. J. Ramos, M. À. Piera, and G. Gigante, "Simulation-based evolutionary optimization of air traffic management," *IEEE Access*, vol. 8, pp. 161 551–161 570, 2020.

[12] L. A. Tuan, M. C. Zheng, and Q. T. Tho, "Modeling and verification of safety critical systems: A case study on pacemaker," *Secure System Integration and Reliability Improvement*, vol. 0, pp. 23–32, 2010.

[13] W. Li, F. Le Gall, and N. Spaseski, "A survey on model-based testing tools for test case generation," in *International Conference on Tools and Methods for Program Analysis*. Springer, 2017, pp. 77–89.

[14] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschner, "Model-based security testing: a taxonomy and systematic classification," *Software Testing, Verification and Reliability*, vol. 26, no. 2, pp. 119–148, 2016.

[15] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. Shashidhar, and S. Ramesh, "Automatic test case generation from simulink/stateflow models using model checking," *Software Testing, Verification and Reliability*, vol. 24, no. 2, pp. 155–180, 2014.

[16] 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, pp. 3–18.

[17] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson, "Automated test generation using model checking: an industrial evaluation," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 3, pp. 335–353, 2016.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on August 10,2021 at 04:15:03 UTC from IEEE Xplore. Restrictions apply.