Concurrent Software Testing Method Based on CSP and PAT

Yizhen Cao
School of Computer Science
Communication University of China
Beijing, China
caoyizhen@cuc.edu.cn

Yongbin Wang
Science and Technology Department
Communication University of China
Beijing, China
ybwang@cuc.edu.cn

***Abstract*—The complexity and nondeterminism of software posed immense challenges for the testing of High Confidence Software, and software failures often caused system failures. This paper establishes a concurrent model for the system under test based on the Communicating Sequential Process (CSP) and completes the test by interacting of the model checking tool Process Analysis Toolkit (PAT) and C# code. This method can cover more system execution paths than the traditional software test methods. In this paper, for optimizing the defect that PAT can only use single-thread to simulate the multi-process, a middle layer is designed to dispatch and distribute the event of PAT abstract processes to execute in the actual .NET managed threads, the concurrency granularity is refined from the function level to statement level, which builds a real multi-thread testing environment that can detect software concurrent errors.**

Keywords—CSP, PAT, model checking, concurrent, testing

I. INTRODUCTION

Testing is indispensable in any software development process, especially in the development of High Confidence Software(HCS). Nowadays' software systems are more and more complex, including a large number of multi-process, multi-thread concurrent execution. The execution sequence of concurrent programs is highly random and more error-prone. The sequence of thread interleaving execution may not be the same every time, as long as a problem has occurred in one sequence, the entire program execution is not correct. This poses great difficulties for software fault elimination. Moreover, the preparation of test cases is very complex and cumbersome and can only test a given limited set of inputs, the automation is in low level. This paper explores a testing method based on model checking, which establishes the concurrency model through the process algebra CSP. The test cases are automatically generated by the model checking tool PAT to test the concurrent software code.

II. SOFTWARE TESTING METHODS BASED ON MODEL CHECKING

At present, the main method of verification and function confirmation of the Safety Critical System (SCS) software is still software testing. However, traditional testing methods select some input data to observe whether the output of the software operation was consistent with the expected value. The testing of a given data set has sampling features, test cases cannot cover all the behavior of the software. Therefore, passing the test does not guarantee that the software will accept other input without any error during the actual operation.

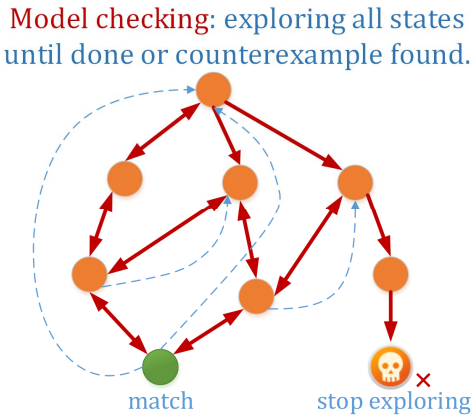
In [1] a variety of methods are introduced for generating test case from formal methods such as Z, VDM, B or Finite State Machine, Process Algebra, and the use of model checking and constraint satisfied techniques in automated testing. A symbolic model checking method that can verify the code, which combines the techniques of model checking, testing and verification is described in [2]. Based on the model checking tools SMV and SPIN, many researches have applied model checking technology in software testing from different perspectives. In [3] a test suite using the SPIN model checker is generated for the Web Service composition specification described by BPEL. In [4] a method for building test sequences based on software-based SCR, forming a set of predicates covering all possible software behaviors and then generating counter-examples by model checking. Literature [5] proposed a new theory of extended finite state machine (EFSMs) to generate tests, and temporal logic is used to describe the coverage criteria of EFSMs control flow and data flow. Reference [6] shows the preconditions of method when the code operates on complex data, which effectively improves the white box test input generation method. Literature [7] applied model checking to the study of the generation of mutation analysis test to generate counter-examples that did not comply with the specification and applied the testing method to a software implementation by Java.

We tried to study a more efficient and practical software testing methods based on model checking. In this approach, contracts [8] have been embedded in the code of the System Under Test (SUT).We first establish a test model for the concurrent system based on the communication sequence process(CSP), describe the expected software properties using temporal logic, and then use model checking techniques to check that the actual software runs dynamically to meet the specified requirements and other important properties such as deadlock, liveness and so on. When validating the model, if the contract is not satisfied, indicating that the code violated the correct specification. On the one hand, the code contract runtime checker will give an exception information. On the other hand, the model checking algorithm will produce the execution path of counter-examples, indicating under what

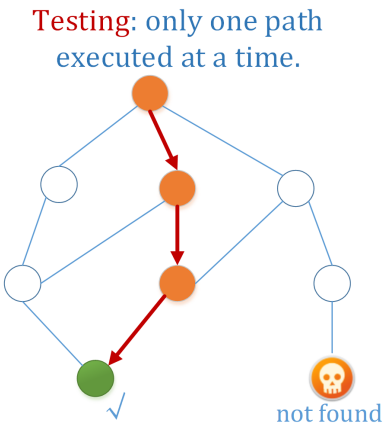
978-1-5386-5892-5/18/$31.00 ©2018 IEEE
ICIS 2018, June 6-8, 2018, Singapore

IEEE computer society

circumstances and input will violate the specified software requirements. This approach is designed to automatically detect the sequence of concurrency events constraints to help developers quickly locate specific software errors, thereby meeting the requirements of fault elimination for the high reliability of SCS.



(a) model checking



(b) software testing

Figure 1. Model checking and software testing

Model checking and software testing are two completely different technologies at an early stage. As shown in Figure 1 (a), the automated model checking analyzes the system model by exploring all states until done or counterexample found. On the contrary, testing technology is oriented toward realistic systems such as model implementations or software source code. But as shown in Figure 1 (b), tests usually cover only a limited portion of the system's execution path based on limited input. However, the differences and boundaries between model and software testing are getting blurred, and more and more model checkers can run on the source code, not just abstract models [1]. Literature [9] directly using C-code modeling and model checking, to test a software cache algorithm. VeriSoft, designed and developed by Godefroid[10][11] to model check concurrent software written by C or C++, successfully validated a 2500-line C-code concurrent search program. Java PathFinder, a model checker developed by NASA in [12], is used to explore state space in distributed multithreaded programs written in Java. When a variety of model checking tools not only support the validation of high-level abstract model, but also the low-level code testing, the use of model checking technology to test software became possible.

We use CSP to build a test model to describe the complicated concurrency in a real system. In the process of CSP theoretical research, some tools were born. This article uses Process Analysis Toolkit(PAT) as model checking tool. PAT is a flexible, common model checking tool based on CSP for modeling, simulating, and validating concurrent system and real-time system. PAT implements a variety of model checking techniques to cater for detection and validation requirements such as deadlocks, reachability, and linear temporal logic. In order to achieve good performance, PAT implements optimization techniques such as partial order reduction, symmetric reduction, and parallel model checking, etc. PAT has better performance than other currently popular generic model checking tools[13]. PAT allows users to define functions and user-defined types using C# language in the model, which means that PAT can connect events in CSPs with actual C# code while the test model is running and make C# programs as part of the test model. The transfer in the test model is the result of the actual system executing the code.

III. Communicating Sequential Processes CSP

Communicating Sequential Processes (CSP) is a kind of process algebra proposed by Hoare [14]. CSP is an event-based formal language for modeling concurrent systems composed of multiple independent processes. Processes are sequences of events, and the communication between the processes is accomplished through atomic and stateless events. CSP has been widely used in the verification of concurrent parts of various systems, such as the design of T9000 virtual channel processor [15], the conversion of Z specification into CSP, and construct secure commercial system through model checking [16], etc. CSP theory itself remains a very active research topic.

The collection of all the events in the process is called Alphabet, and αP is used to represent the process P 's alphabet. The behavior of a process can be defined as a sequence of events that combine the basic process and process operators. The basic processes in CSP include $STOP$ and $SKIP$. $STOP$ indicates that the process does not execute any events in this state and will not stay in that state all the time. $SKIP$ indicates the successful termination of the process. The operator $\rightarrow$ describes the order of the events. The expression $a \rightarrow P$ indicates that the first event of the process is a , after execution, the remaining part is P .

IV. Build the Middle Layer to Achieve Concurrent Program Testing

It is very difficult to describe some advanced data structures and function codes in formal languages CSP, such as describing lists, stacks, queues, and operations on these structures are complex and inefficient. Therefore, PAT supports the use of the C# language to define static methods or user-defined data types to facilitate formal modeling. After the C# class is compiled into an assembly (.dll), it is imported into

PAT using the #import statement, and the model can use the keyword call to directly call methods and data types in C#. This method takes the C# class as part of a formalized model. Transfer events in the model will call execution of C# code while executing simulations and verifications, so that PAT can be used for testing C# code.

However, when we use CSP for modeling in PAT, we found that multiple processes are concurrently executed in the model, but all the events and transitions are simulated in the same real thread. Some interfaces in the implementation of the software are not competing with resources when single thread is called, but problems can occur when multiple threads are used.

```
#import"C:\Users\DELL\Documents\Projects\T\bin\Debug\T.dll";
var sum;
P(i)=GetSum.i{sum=call(AddRandom)}->Skip;
system()=|||i:{0..2}@P(i);
```

Figure 2. Use CSP to simulate concurrent calls to C # code in PAT

For example, in the model in Figure 2, we call a C# method AddRandom which first generates a bounded random number n and reads out the sum. If the sum is greater than the threshold max, it returns sum, and conversely calculates sum=sum+n, finally return sum. We define in the CSP that the system is interleaved by three processes P(i), and hopefully this model will produce three threads concurrently executing the method AddRandom and synchronizing between threads, as long as one thread computes sum>max as true, other threads will no longer perform the sum=sum+n operation. Sum is a typical shared resource. According to the experience of multithreaded programs, the thread may read dirty data, that is sum<max in P(0) and sum<max in P(1). It may lead to sum>max while each of them implements addition, obviously, this is a Bug. However, when P(0), P(1), and P(2) are executed concurrently, the actual operation uses the same thread in C#, so that sum>max will certainly not occur. In other words, C# program code cannot be executed by multiple threads in the PAT so that the concurrent bugs in this program cannot be detected. Neither, in this way it will not detect the existence of deadlock, the violation of the contract and other common multi-threaded programming problems in the real code.

To address this shortcoming, we designed an middle layer with two sublayers between the PAT and the System Under Test (SUT) written by C#. As shown in Figure 3, the Interface layer first defines the delegate to the method under test in the SUT. Then follow the specification of calling C# code in PAT, declaring that the test interface with type public static corresponds to the SUT method. The main difference between the test interface in the Interface sublayer and the SUT method is that the test interface needs to receive the Process ID in the PAT in order to be able to identify the correspondence between the process P(i) and the execution thread. The test interface does not directly call the SUT method, just pass the event message to another sublayer EventPump (EP).

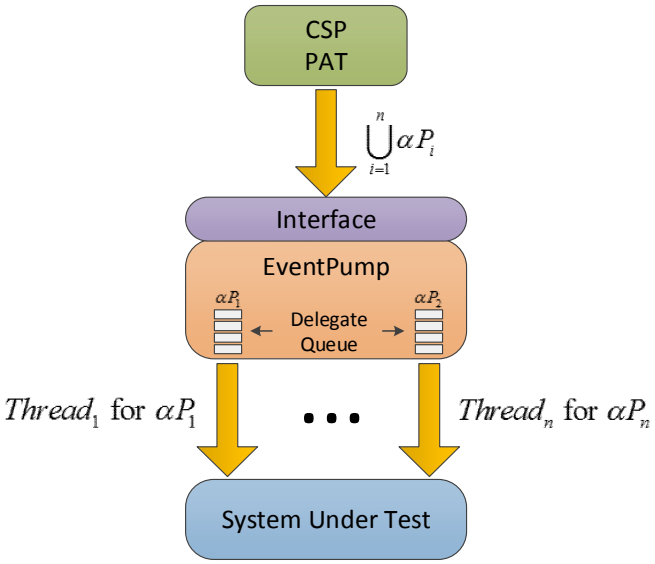


Figure 3. Use middle layer to schedule simulator events to execute on multiple threads

Algorithm 1 Construct EventPump and add event

```
1: Define delegate x mapping to the SUT operation
2: if EventPump instance pumps[i] is null then   ▷ i is the identifier for CSP process
3:     new EventPump instance pumps[i]
4:     new thread t of pumps[i]
5:     t starts executing DoPump()
6:     pumps[i].isWorking ← true
7: end if
8: Add x to event queue of pumps[i]
```

Algorithm 2 DoPump

```
1: while pump[i].IsWorking do
2:     while event queue is not empty do
3:         Dequeue x from queue of pumps[i]
4:         Invoke x
5:     end while
6:     Thread t sleep 100 ms
7: end while
```

Figure 4. EventPump algorithm pseudo-code

The pseudo code in Figure4 shows the core idea of EP:

Algorithm 1 builds an instance of EP and creates a new thread, executes the DoPump algorithm at the beginning of the thread, defines the operation of the delegate mapping to the SUT, and adds the calling event x of the CSP to the event queue.

Algorithm 2 constantly fetches the first delegate in the event queue on the thread owned by the EP instance and invokes it.

It can be seen that the role of the EP is to schedule multiple concurrency events in the PAT (call the test interface in Interface sublayer) to the corresponding .NET managed thread for execution. In the EP, we create an instance pump to each process P(i) in the CSP, which contains an event message queue. When the event of P(i) calls the real method, EP will join the commission of the method to the message queue. Each EP instance will create a new thread and will continue to get and execute the delegate method from the message queue. In order to be able to identify the correspondence between P(i)

and the thread, it is necessary to pass the process parameter i to the interface of the EP together when the CSP calls the C# method.

Testing C# code directly on the basis of PAT, in essence, its concurrency simulation constructs all the traces in units of C# function. The method proposed in this section can refine the concurrency granularity from the function level to the statement level to achieve the effect that the tested software runs truly concurrently. However, the order of statement-level concurrency is determined by the .NET execution without increasing the state space of the PAT model checking and without reducing the efficiency of the model checking algorithm. Therefore, this method is suitable for automatic testing of concurrent programs based on model checking.

V. CONCLUSION

This paper presents a software testing method for concurrent systems. The foundation of this approach is embedding the contract's implementation code so that the contract can be dynamically checked when the software is running. Due to the concurrent execution sequence is highly random, more error-prone, we try to use the model checking to automatically generate test cases by modeling in CSP and simulating and verifying in PAT. However, because PAT simulates multiple concurrent processes in only a single thread instead of multiple .NET processes, actually it is not possible to test the correctness and reliability of the SUT which is a multithreaded/multi-process software. In response to this flaw, we designed and developed an middle layer between the PAT and C# code that dispatches the abstract process events in the PAT to the actual .NET managed threads to refine the concurrency granularity from the function level to the statement level, which completely simulated the actual multi-threaded test environment. This method does not increase the state space and does not reduce the efficiency of the model checking algorithm. Therefore, this method of automated testing provides practical and specific fault elimination tools for the development of high confidence software for safety critical system.

ACKNOWLEDGMENT

This paper is supported by National Science and Technology Support Program(NO.2015BAK22B01).

REFERENCES

[1] Hierons R M, Bogdanov K, Bowen J P, et al. Using formal specifications to support testing[J]. ACM Computing Surveys (CSUR), 2009, 41(2): 9.

[2] Gunter E, Peled D. Model checking, testing and verification working together[J]. Formal Aspects of Computing, 2005, 17(2): 201-221.

[3] García-Fanjul J, Tuya J, De La Riva C. Generating test cases specifications for BPEL compositions of web services using SPIN[C]//International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006). 2006: 83.

[4] Gargantini A, Heitmeyer C. Using model checking to generate tests from requirements specifications[C]//Software Engineering—ESEC/FSE'99. Springer Berlin Heidelberg, 1999: 146-162.

[5] Hong H S, Lee I, Sokolsky O, et al. A temporal logic based theory of test coverage and generation[M]//Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2002: 327-341.

[6] Visser W, Păsăreanu C S, Khurshid S. Test input generation with Java PathFinder[J]. ACM SIGSOFT Software Engineering Notes, 2004, 29(4): 97-107.

[7] Ammann P E, Black P E, Majurski W. Using model checking to generate tests from specifications[C]//Formal Engineering Methods, 1998. Proceedings. Second International Conference on. IEEE, 1998: 46-54.

[8] Fähndrich M. Static Verification for Code Contracts[M]// Static Analysis. Springer Berlin Heidelberg, 2010.

[9] Eisner C. Formal verification of software source code through semi-automatic modeling[J]. Software & Systems Modeling, 2005, 4(1): 14-31.

[10] Godefroid P. Model checking for programming languages using VeriSoft[C]//Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1997: 174-186.

[11] Godefroid P. VeriSoft: A tool for the automatic analysis of concurrent reactive software[C]//Computer Aided Verification. Springer Berlin Heidelberg, 1997: 476-479.

[12] Stoller S D. Model-checking multi-threaded distributed Java programs[M]//SPIN Model Checking and Software Verification. Springer Berlin Heidelberg, 2000: 224-244.

[13] Sun J, Liu Y, Roychoudhury A, et al. Fair model checking with process counter abstraction[M]//FM 2009: Formal Methods. Springer Berlin Heidelberg, 2009: 123-139.

[14] Hoare C A R. Communicating sequential processes[M]. Englewood Cliffs: Prentice-hall, 1985.

[15] Barrett G. Model checking in practice: The t9000 virtual channel processor[J]. Software Engineering, IEEE Transactions on, 1995, 21(2): 69-78.

[16] Hall A, Chapman R. Correctness by construction: Developing a commercial secure system[J]. Software, IEEE, 2002, 19(1): 18-25.