Automatic integration testing through collaboration diagram and logic contracts

Yi Sun¹, Xiaohua Yang¹, Jie Liu¹, Tonglan Yu¹, Zhuoran Xu¹, Zhiqiang Wu² and Zhi Chen²

¹School of Computer, University of South China, Hengyang, Hunan, 421001, China
²Key Laboratory for Nuclear Reactor System Design, Nuclear Power Institute of China, Chengdu, Sichuan, 610041, China

E-mail: xiaohua1963@foxmail.com

Abstract. Component-based software development can effectively increase software development efficiency through component reuse. Integration testing is an important technical means to guarantee the quality of software systems composed of components. Aiming at the problem that the scale of software systems is getting larger and larger, and manual design of test cases is costly and difficult, we propose an automatic approach for integration test cases generation based on collaboration diagram and logic contracts. By extracting control flow information from collaboration diagram and combining with contracts as component specification, an intermediate model called execution tree of components is established, then test cases are automatically generated through contract solving technology. This approach not only realizes path coverage, but also completes the automatic integration testing of software system, thereby improving test efficiency and reducing test cost.

1. Introduction

With the increasing scale of software systems, component-based software development technique has been widely accepted in order to reduce development cost and difficulty. Component-based software development builds software system by integrating reusable components, and components collaborate to achieve functions of system. Although the reused components have undergone rigorous unit testing, this does not guarantee that the integrated software system will not have problems, so integration testing is needed to ensure the quality of software.

Unified Modeling Language (UML) is one of the main tools for object-oriented software development, which provides various models for modeling software system. UML models describe static structure and dynamic behaviors of the system, so they can provide guidance for test case generation in software testing. Among these UML models, collaboration diagram is mainly used to describe collaborations and interactions among system objects. The functions of component-based software are completed by interactions among components, so we select collaboration diagram as test model for generating integration test cases.

The main purpose of integration testing is to detect whether interactions among components are implemented correctly. Collaboration diagram points out the components involved in implementing system functions, and interaction relationships among these components can also be found in it. In recent years, researchers [1, 2, 3, 4] proposed some methods to generate test cases from collaboration diagram. These approaches all took advantage of interaction information among objects in collaboration diagram to generate test cases and developed towards automated testing, because it is...
unrealistic to rely entirely on manual test case generation with the growing complexity of modern software.

An important theory of component quality assurance is contract theory [5], which is widely used as the specification of components. Contracts define interaction rules among components to regulate behaviors of components, which are predicate logic formulas that describe the constraints that must be satisfied before and after the component call. In terms of utilizing contracts to generate test cases, the work in [6, 7] either lacks support for integration testing, or isn’t suitable for interactions among components. In addition, the approach presented in [8] does not provide complete system model information when generating test cases and lacks the adequacy of system integration testing.

Therefore, we combine collaboration diagram with logic contracts to automatically generate integration test cases in this paper. By extracting control flow information from collaboration diagram and combining with contracts as component specification, an intermediate model called execution tree of components is established. On the basis of this intermediate tree, contracts of components are collected along path to form path constraint. Solution to path constraint is used as test input, and the postcondition is used as test oracle. Test suite is generated to realize path coverage, thus confirming whether components behave as expected in the system.

The main contribution of this paper is to propose the approach of integrating collaboration diagram and contracts. On this basis, test cases are generated automatically by the technique of contract solving, and path coverage in integration testing of components is realized, so as to solve the problem of integration test case generation of large-scale software system.

The rest of this paper is as follows: Section 2 introduces test model based on UML collaboration diagram. Section 3 presents our proposed approach to generate test cases. Section 4 demonstrates the feasibility of this approach through an example of ATM. Section 5 is a summary of the paper, and future work is drawn.

2. Test model based on UML collaboration diagram

2.1. Component collaboration diagram

UML collaboration diagram (CD) describes collaboration relationships among components. These relationships define the interactions among components during software execution, which is the basis for component integration. Therefore, collaboration diagram can be used as test model for component integration. A CD = <CP, MS>, where CP is a finite set of components and MS is a finite set of messages. Each component in CP is a finite set of methods, and all messages in MS are arranged according to messages' sequence number.

A component is the encapsulation of a series of methods. It only provides limited information about methods to users, including method names, input and output parameters, and contracts. A contract indicates constraints that must be observed when accessing internal methods of component, including precondition and postcondition, representing constraints that must be met before and after the invocation of a method, respectively. Precondition and postcondition are both represented in first-order logic. For a correct method, if the input satisfies precondition, the output must satisfy postcondition.

In collaboration diagram, collaborations start from an external event triggering, then a series of interactions are completed among components, finally the collaborations end when no more messages are generated, resulting in an external output result. In this process, component interactions are achieved through message passing. A sequence of messages represents one execution of the system. Collaboration diagram depicts the execution process of system functions, and each system task corresponds to an execution path in collaboration diagram. Therefore, integration testing of components should be able to reflect the correctness of the system execution process under different tasks, that is, testing each execution path with test data.

Collaboration diagram describes components, which are integrated into software, and interaction messages among these components. A message corresponds to a method of component, and the
corresponding method is activated by the message. Message sequence activates corresponding method sequence, which indicates the process of components execution.

In addition to corresponding method, a message also includes other information such as sequence number, condition and type. The sequence number indicates nest relation among messages and the sequential order in which messages are executed. For example, message 1.1 and 1.2 are the first and second nested messages during message 1 execution. The condition represented in first-order logic can be true or false. A message will be executed if condition is true, otherwise not. There are three types of messages: common, conditional and recurrent message. Common message's condition is null. Conditional message is executed when its condition is true, otherwise the message and its nested messages will not be executed. Recurrent message whose nested messages are included in loop body is executed when its condition is true, otherwise not. Loop is restricted to be executed zero times or once to avoid path explosion.

It should be noted that condition of message and contract of method belong to different constraint systems. Condition of message causes different control flows, and contract of method is used to describe the constraints under which method is executed. They are both represented in first-order logic.

2.2. Execution tree of components

In collaboration diagram, the execution path of a system function starts with a message without precursor, then follows a message sequence, finally ends at a message without successor. Meanwhile, methods in components are activated by these messages and a method sequence is formed. This path is called functional execution path.

Condition of message leads to branches which form multiple functional execution paths, thus a tree is created. The nodes of this tree denote messages in collaboration diagram, which will activate corresponding methods in components. The prerequisite of activating a method is to satisfy contract of this method, so contract is also included in the path (i.e., functional execution path) constraint. If path constraint is solvable, it shows the path is reachable; otherwise, the path is unreachable.

In our proposed approach, collaboration diagram with contracts is transformed to an intermediate model called execution tree of components, then this intermediate tree is traversed to collect conditions of messages and contracts of methods to form path constraints. Eventually, test cases used to test each reachable path are acquired with the help of constraint solver.

An execution tree of components (ETC) is a directed tree where $ETC = <TN, TE>$, $TN$ is a finite set of nodes and $TE$ is a finite set of directed edges. Each node in $TN$ corresponds to a message, and contains constraint information called node constraint. The node constraint is the conjunction of condition of message and contract of method. Moreover, root node represents initial message (i.e., message without precursor) which is triggered by an external event, and leaf node represents terminal message (i.e., message without successor) resulting in external output. A sequence of nodes from root node to leaf node represents a functional execution path, which must satisfy path constraint composed of node constraints.

3. Proposed approach to generate test cases

Collaboration diagram is used as test model to generate integration test cases. Each interaction is processed until all interactions in collaboration diagrams are tested, that is, messages among all component objects in the system are tested at least once. This is adequacy criterion of the proposed approach in the paper.

In the intermediate tree, a sequence of methods activated by messages on a functional execution path represents the functional behavior to be tested, and message passing among component objects on the path describes the necessary interactions among the system functional objects. Therefore, path coverage in execution tree of components meets aforementioned adequacy criterion, so that the problem can be transformed to the analysis and processing of paths in the execution tree of components.
3.1. Constructing an intermediate tree

The way of constructing an intermediate tree ETC is to map messages in collaboration diagram to a set of nodes, which are then built into a tree according to sequence number and condition of message.

In the process of mapping, a mapping table is created to record information about mapping. Because condition of message exists, a message can be mapped to multiple nodes. For example, message 2 is conditional message in message sequence "1→2→3", so functional execution path "1→2→3" is formed when condition of message 2 is true, and functional execution path "1→3" is formed when condition of message 2 is false. In this case, message 3 is mapped to two nodes which are located in two different functional execution paths.

It should be noted that node constraint consists of condition of message which has been processed and contract of method. Still take above message sequence "1→2→3" as an example, condition of message 3 is a part of node constraint when mapping message 3 to node in the functional execution path "1→2→3". In the other functional execution path "1→3", the conjunction of condition of message 3 and negative condition of message 2 is a part of node constraint when mapping message 3 to node. Furthermore, there is a special case: take message sequence "1→2" as an example, since message 2 is conditional message, message 2 is skipped and the following message is mapped to nodes when condition of message 2 is false. However, message 2 is also terminal message, and the following message does not exist, so a virtual node v is created to construct functional execution path "1→v", so as to keep integrality of path. In the process of mapping message to virtual node, the message is null, and node constraint of virtual node consists of negative condition of message 2.

The above describes how to map message to nodes, which is mainly to process condition of message and contract of method so as to form node constraint. According to this operation, we process messages in collaboration diagram in turn.

In the message sequence of collaboration diagram, the first message may be conditional or recurrent message, which results in branches at the beginning and a forest is generated. Therefore, to avoid this situation, a null message is inserted at the head of message sequence, which will be mapped to the virtual root node. Then, remaining messages are processed according to their type.

For common message, it is mapped to nodes directly, whose node constraints are made up of condition of this common message and contract of corresponding method. The number of these nodes is the same as the number of nodes which the precursor of current message is mapped to. Then, use one-to-one correspondence to take these nodes as children of nodes which the precursor of current message is mapped to.

For conditional message, it should be firstly determined whether this message has existed in mapping table. If this message has existed in mapping table, this message will be mapped to these existing nodes when condition of the message is true. On the other hand, take it into consideration that condition of this message is false, this message and its nested messages are skipped, and the following message is mapped to nodes whose number is the same as the number of these existing nodes. Then, use one-to-one correspondence to take nodes, which the following message is mapped to, as brothers of these existing nodes.

After dealing with the above situation, new situation is considered. The conditional message is processed in the same way as common message when condition is true. On the other hand, when condition is false, the conditional message and its nested messages are skipped, and the following message, which hasn’t been processed, is mapped to nodes whose number is the same as the number of nodes which the precursor of conditional message is mapped to. Then, use one-to-one correspondence to take these nodes as children of nodes which the precursor of conditional message is mapped to. This operation may cause unprocessed conditional messages to appear in mapping table in advance, this is why it should be firstly determined whether the message has existed in mapping table before processing conditional message.

For recurrent message, it is processed in the same way as conditional message since loop is restricted to be executed zero times or once.
Collaboration diagram with contracts is automatically transformed into execution tree of components through the above operation, which lays a foundation for subsequent test case generation automatically.

3.2. Generating test cases
On the basis of execution tree of components, the automatic generation of test cases can be divided into two steps: the first is to get path constraints, and the second is to solve path constraints.

3.2.1. Getting path constraints. Constraints in collaboration diagram, which include conditions of messages and contracts of methods, have been stored in nodes of tree during the transformation from collaboration diagram to execution tree of components. Consequently, in the first step, the only work is to process constraints in nodes (i.e., node constraints).

As the path is traversed, path constraint, the conjunction of node constraints, is got. As the execution tree of components is traversed, all path constraints are got.

3.2.2. Solving path constraints. A Boolean satisfiability (SAT) problem [9] is a typical constraint solving problem, which is the problem of determining if there exists an assignment that satisfies a given propositional logic formula. The object to be solved of SAT problem is propositional logic formula, whose expressive power is limited to propositional calculus, so researchers extend SAT to satisfiability modulo theories (SMT) [10] by adding theories to SAT to enhance expressive power of SAT. SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in first-order logic.

International SMT competitions have been held every year since 2005, and the top several solvers in 2016 and 2017 are Z3, CVC4, Yices2, SMTInterpol, veriT and MathSat5. The analysis results show that the comprehensive solving ability of Z3 solver [11] is the strongest. Therefore, we select Z3 solver to solve path constraints, thereby generating test cases.

A test case is a tuple <ID, path, input, testoracle>, where the elements are defined as follows. ID is a unique identifier used to distinguish different test cases. The path represents path corresponding to this test case. The input represents test input which is solution to path constraint, and it can be got with the help of Z3 solver. The testoracle represents test oracle which is used to check whether actual output is expected. We take postcondition of contract of method which corresponds to leaf node as test oracle.

4. Case study
This paper takes ATM machine as an example to illustrate how to automatically generate integration test cases through collaboration diagram and logic contracts. UML collaboration diagram obtained from the specification is shown in figure 1.

This collaboration diagram describes the process of using ATM machine to realize the withdrawal function. There are five components in it: Client, CardReader, ATMScreen, Account and Withdrawer. Besides, there are also thirteen messages in it, and they are shown as follows.

- 1: isBankCard:=(boolean)insertCard(boolean bankCard)
- 2: isValidCard:=(boolean)readCard(boolean bankCard)
- 3: initializeScreen()
- 4: inputPwd(boolean validPwd)
- 4.1, 5.1: isValidPwd:=(boolean)validatePwd(boolean validPwd)
- 5: *[isValidPwd && i:=1...3] inputPwd(boolean validPwd)
- 6: displayMenu()
- 7: selectAction(int action)
- 8: inputAmount(int amount)
- 9: [amount<=30000] deductMoney(int amount)
- 9.1: spitSucc:=(boolean)spitMoney()
• 10: opCompleted:=(boolean)ejectCard()

Figure 1. Collaboration diagram of ATM machine.

The initial state of ATM machine is balance(30000), which represents the current account balance is 30000. In addition, for convenience, the corresponding functions of ATM machine are numerically denoted, and the list of functions is as follows: 0 - deposit, 1 - withdrawal, 2 - transfer accounts, 3 - query, 4 - password modification. In this collaboration diagram, message 5 is recurrent message, message 9 is conditional message, and the rest are common messages.

According to section 3.1, the intermediate model generated, execution tree of components, is shown in figure 2.

Figure 2. Execution tree of components.

Consider the functional execution path "1→2→3→4→4.1→6→7→8→10", we illustrate test case generation for it. In this path, recurrent message 5 is not executed and condition of conditional message 9 is false, i.e., password entered is valid and amount entered is greater than account balance. Message sequence and processed conditions corresponding to this path are shown in table 1, and the
corresponding method sequence and contracts are shown in table 2. Then, according to section 3.2.1, path constraint of the functional execution path generated is shown in table 3. Furthermore, according to section 3.2.2, test case corresponding to this path is generated, and shown in table 4.

Table 1. Message sequence and processed conditions.

| message | 1 | 2 | 3 | 4 | 4.1 | 6 | 7 | 8 | 10 |
|---------|---|---|---|---|-----|---|---|---|----|
| condition of message (processed) | ¬ (isValidPwd) | - | - | - | - | ¬ (amount<=30000) | - | - | - |

Table 2. Method sequence and contracts.

| method | contract |
|--------|----------|
| isBankCard:=(boolean)insertCard(boolean bankCard) | bankCard = = true isBankCard = = true |
| isValidCard:=(boolean)readCard(boolean bankCard) | bankCard = = true isValidCard = = true |
| initializeScreen() | - |
| inputPwd(boolean validPwd) | validPwd = = true - |
| isValidPwd:=(boolean)validatePwd(boolean validPwd) | validPwd = = true isValidPwd = = true |
| displayMenu() | - |
| selectAction(int action) | action = = 1 - |
| inputAmount(int amount) | amount > 0 && amount%100 = = 0 - |
| opCompleted:=(boolean)ejectCard() | - opCompleted = = true |

Table 3. Path constraint.

| path | 1→2→3→4→4.1→6→7→8→10 |
| constraint | bankcard = = true ∧ validPwd = = true ∧ action = = 1 ∧ (amount > 0 && amount%100 = = 0) ∧ (¬ (amount <= 30000)) |

Table 4. Test case.

| actual test case |
|------------------|
| ID | ts1 |
| path | 1→2→3→4→4.1→6→7→8→10 |
| input | (true, true, 1, 35000) |
| testoracle | opCompleted = = true |

What this test case represents is: a valid bank card is inserted, then valid password is entered and the withdrawal function is selected, but amount entered is greater than account balance, so it fails to withdraw money. Finally, bank card is ejected. "opCompleted = = true" represents succeeding in ejecting bank card. The failure to eject bank card means that interactions among components aren't implemented correctly and components doesn't behaves as expected.
5. Conclusions
As an important dynamic model in object-oriented software, UML collaboration diagram is an excellent integration test model. Utilizing component level collaboration diagram with contracts, this paper proposes a new approach for integration test case generation. In this approach, an intermediate model named as execution tree of components is constructed by analyzing messages in collaboration diagram and contracts in components. On this basis, path constraints which correspond to all paths are got and solved to generate test cases. Our proposed approach can automatically generate the intermediate tree by parsing the specification file of collaboration diagram, and then automatically generate test cases through contract solving technology, so as to realize the automation of integration testing. Compared with manual testing, this approach improves test efficiency and reduces the test cost. A case proves this approach is feasible, and our future work is to develop a tool to support it.

Acknowledgments
We acknowledge the support from Nuclear Power Institute of China (grant LRSDT2017304) and Education Department of Hunan (grant 17C1378).

References
[1] Prasanna M, Chandran K R, Thiruvenkadam K. (2011) Automatic test case generation for UML collaboration diagrams. IETE Journal of Research, 57: 77-81.
[2] Zeng Y, Liu Q X, Wang C Q, et al. (2013) Generation method of polymorphic path test scenarios with OCL constraints. Computer Engineering, 39: 92-96+102.
[3] Swain R K, Panthi V, Mohapatra D P, et al. (2014) Prioritizing test scenarios from UML communication and activity diagrams. Innovations in Systems & Software Engineering, 10: 165-180.
[4] Kaur A, Vig V. (2018) Automatic test case generation through collaboration diagram: a case study. International Journal of System Assurance Engineering & Management, 9: 1-15.
[5] Meyer B. (1997) Object-Oriented Software Construction. Prentice Hall Publishing, Englewood.
[6] Zhao Y N, Guo H L. (2014) Component testability research based on contract status checking. Modern Electronics Technique, 37: 83-85+88.
[7] Zhao Y N, Guo H L. (2014) Component testing research based on contract checking. Journal of Xi'an University of Science and Technology, 34: 290-295.
[8] Xu D, Xu W, Tu M, et al. (2016) Automated integration testing using logical contracts. IEEE Transactions on Reliability, 65: 1205-1222.
[9] Guo Y, Zhang C S, Zhang B. (2016) Research advance of SAT solving algorithm. Computer Science, 43: 8-17.
[10] Jin J W, Ma F F, Zhang J. (2015) Brief introduction to SMT solving. Journal of Frontiers of Computer Science & Technology, 9: 769-780.
[11] Moura L D, Bjorner N. (2008) Z3: an efficient SMT solver. In: the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Budapest. pp. 337-340.