

End-To-End Integration Testing Design

W. T. Tsai*, Xiaoying Bai*, Ray Paul**, Weiguang Shao+, Vishal Agarwal+,

*Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85259

+Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455

**OASD C3I Investment and Acquisition
Washington, D.C.

Abstract

Integration testing has always been a challenge especially if the system under test is large with many subsystems and interfaces. This paper proposes an approach to design End-to-End (E2E) integration testing, including test scenario specification, test case generation and tool support. Test scenarios are specified as thin threads, each of which represents a single function from an end user's point of view. Thin threads can be organized hierarchically into a tree with each branch consisting of a set of related thin threads representing a set of related functionality. A test engineer can use thin-thread trees to generate test cases systematically, as well as carry out other related tasks such as risk analysis and assignment, regression testing, ripple effect analysis. A prototype tool has been developed to support E2E testing in a distributed environment on the J2EE platform.

1 Introduction

Testing is still the primary means for quality assurance today. Even though many testing techniques have been proposed in the past, most of them focus on module testing. However, in practice, integration testing is often the most time consuming and expensive part of testing. It is common to find software development projects with 50% to 70% of effort on testing, and 50% to 70% of the testing effort on integration testing.

A review of the literature on integration testing shows that most integration testing techniques are principles, such as incremental integration, top-down, and bottom-up integration [10], or

explore the programming or design structure of the program [6][7][13][14]. The techniques that explore programming or design structure are useful, however, they are applicable to software written using the related techniques only. For example, an integration testing technique for an object-oriented program using Java may not be applicable to testing a legacy program using COBOL. It is even possible that testing technique may not be applicable to a C++ program because Java has no pointers but C++ does.

Due to these considerations, Department of Defense (DoD) initiated a project on End-to-End (E2E) integration testing [19]. This test process verifies that a defined set of interconnected systems, now subsystems of the integrated system, will perform correctly.

E2E testing is different from module testing where the focus is on single modules. E2E testing is similar to integration testing; however, E2E testing focuses exclusively from the end user's point of view while integration testing can focus on any subset of subsystems. E2E testing assumes that both module testing and integration testing have been performed and approved, which may include multiple levels of integration testing.

The project covers many aspects of E2E testing including test planning, test design, test execution, test result analysis, regression testing, risk analysis and assignment, ripple effect analysis. This paper presents the test design, which is the foundation of the E2E project.

E2E test design specifies the E2E test requirements of the system under test, generates test scenarios and test cases based on the specifications. It uses *thin threads* to specify test design, which has been successfully used during

the largest DoD testing effort ever – DoD Y2K testing. A thin thread represents a basic system function. Thin threads are organized hierarchically into a tree with each branch, called thin-thread group, consisting of related thin threads representing related functionality. This thin-thread specification can be applied to both modern applications such as distributed component-based OO programs as well as legacy programs written in COBOL.

Each thin-thread is associated with a set of conditions, or predicates, specifying its triggering events. Conditions are also organized hierarchically into a tree, with a branch consisting of related conditions.

A thin-thread represents a single function, however, in practice a user may perform multiple functions in a single session. This kind of composite or complex scenarios can be represented by combining thin threads using constructs such as sequence, condition (such as *if-then-else*) and looping constructs.

Test cases are generated from test scenarios. From a thin thread, a test case can be generated by assigning the condition variables with real values based on various techniques, such as equivalent class testing, boundary testing, random testing, and stress testing.

This paper is organized as following. Section 2 presents the E2E test specification. Section 3 explains the techniques to generate test scenarios and test cases based on the E2E test specification. Section 4 briefly outlines the E2E testing support tool. Finally, Section 5 concludes this paper.

2 E2E Test Specifications

2.1 Thin Threads

According to DOD Management Plan [17], a thin thread is defined as:

A complete trace (E2E) of data/messages using a minimally representative sample of external input data transformed through an interconnected set of systems (architecture) to produce a minimally representative sample of external output data. The execution of a thin thread demonstrates a method to perform a specified function.

Thus, a thin thread is a minimum usage scenario of the integrated system. Essentially, a thin thread is a complete scenario from the end user's point: the system takes input data, performs some computation, and produces resulting output. The thin thread describes the

whole scenario and it describes just *one* function. Taking a banking application, example thin threads can be: *a successful withdraw transaction from a local bank* or *a failed withdraw transaction from a remote bank due to insufficient fund*.

Thin threads that share certain commonalities can be grouped together into a thin-thread group. For example, the thin threads identified above can be grouped into a thin-thread group *withdraw transactions from a local bank*. Such grouping can be recursive, i.e., a collection of lower-level thin thread groups that share certain commonality can be further grouped into a higher-level thin thread group. For example, thin-thread group *withdraw transactions from a local bank* and *withdraw transactions from a remote bank* can be grouped into a super group *withdraw transactions*. In this way, all thin threads and thin thread groups can be arranged hierarchically into a thin-thread tree.

A thin thread and a use case [4] serve similar functions, i.e., describing system scenarios. However, a thin thread contains more information than a use case, and thin threads for an application are organized into a tree suitable for various analysis such as dependency analysis, risk analysis, traceability analysis, coverage analysis, completeness and consistency checking, and test case/scenario generation [1].

Identify Thin Threads

Identifying thin threads requires a thorough understanding of the system under test, including the system functionality as well as the system architecture. Thin threads identified from system functionality are black-box thin threads, since they are identified from external views of the system, while those identified from system structure are white-box thin threads as internal views are taken into consideration.

Taking a banking example, the following are hints on identifying thin threads:

- Identify thin-thread groups by analyzing the major business functions. That is, black-box thin threads are identified first. For example, in the banking system, the major functionalities of the system are identified as: *withdraw*, *deposit*, and *check-balance*.
- Decompose thin-thread groups by analyzing its inputs. For example, one can decompose the group *check-balance transactions* into two subgroups, one with *valid inputs*, and the other with *invalid inputs*.
- Decompose thin-thread groups by analyzing its outputs. For example, group *withdraw*

transactions can have different outputs: one is *successfully withdraw*, and another may be *an error message for insufficient fund*. Therefore, this thin-thread group can be decomposed into two subgroups, one is *successful withdraw transactions*, and the other *failed withdraw transactions due to insufficient fund*.

- Decompose a thin-thread group by analyzing its possible execution paths. For example, group *withdraw transactions* has two possible execution paths: local withdraw transactions and remote withdraw transactions. Hence, it can be decomposed into two sub-groups as *withdraw transactions from a local bank* and *withdraw transaction from a remote bank*.

Relationships Among Thin Threads

Thin threads relate to each other with respect to their execution paths:

- Contained in: The execution path of a thin-thread can be part of the path of another thin-thread.
- Identical: Two thin threads have the same paths. In this case, they may share a certain set of attributes, such as conditions.
- Independent: Thin-threads have completely different paths.

This relationship among thin threads can be useful in scheduling test case execution. For example, if a thin thread is on a critical path of the system, it should be tested thoroughly and probably as early as possible. If a set of thin threads will be selected for testing, it may be appropriate to select thin threads with independent execution paths to ensure some kinds of coverage.

Construct Thin-Thread Trees

The root of a thin-thread tree represents the overall integrated system under test, a branch node represents a collection of related thin threads (or thin-thread group) and a leaf represents a concrete thin thread. In most cases, the thin threads within a thin-thread group are related by their common functionality. Thus, a thin-thread tree can be viewed as a functional decomposition of the system under test. Thin-thread tree can be constructed in many ways:

- Top-down: identify the major thin-thread groups and construct tree by decomposing of thin-thread groups level by level;

- Bottom-up: identify the atomic usage scenarios and construct tree by composition thin threads (or groups) level by level;
- Combined approach: Use the top-down and bottom-up approaches together during the process.

Figure 1 shows an example thin-thread tree for a banking application.

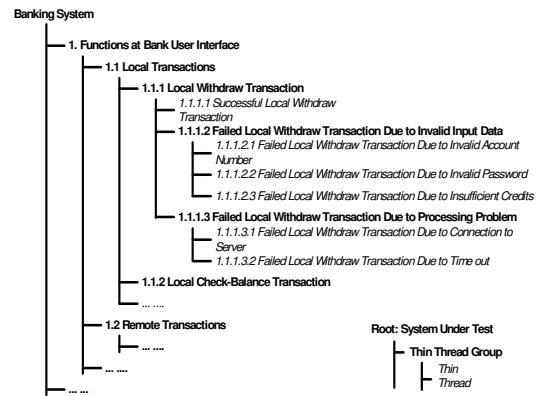


Figure 1. An Example Thin-Thread Tree

Attach Data to Thin Thread

For each thin-thread/thin-thread-group, it is necessary to identify input and output data as well as the data manipulated. For example, in a banking example, a *withdraw* transaction may involve the following data:

- Bank ID;
- Account information;
- Account password;
- Transaction information.

These data should be attached to the affiliated thin thread and stored in the test database. They are useful for test case generation.

2.2 Conditions

Conditions are predicates that affect the execution of thin threads. A thin thread is activated if and only if all its conditions are satisfied. Some possible conditions include:

- Communication conditions: Examples are communication delays in the time-out mechanisms, recovery mechanism, security mechanisms, and protocol.
- Sequencing and timing conditions: Examples include ordering sequences, minimum delays, maximum allowable delays, daily, monthly and quarterly updates, and coordination among cooperating parties.
- Data conditions: These conditions involve data items. For example, a user account

number is required for any transaction in a banking application, and absence of a valid number causes the transaction fail.

- Environmental conditions: External environment may also affect thin threads to act differently. For example, in a banking system, heavy load on a bank may cause a transaction to be denied even if all the transaction data are valid.

Identify Conditions

Similar to thin threads, conditions can also be identified from system functionality as well as structure. It is important to identify both positive and negative aspects of the condition. Some hints on condition identification are:

- a. Identify data conditions from input and output of a thin thread. For example, a thin thread *successful withdraw transactions* may have input data as *account number*, *password*, *withdraw amount*, and *account balance*. Accordingly, the conditions for this thin thread include:
 - 1) *Valid account number*;
 - 2) *Valid and matching password*;
 - 3) *Valid withdraw amount*; and
 - 4) *Sufficient account balance*.
- b. Identify data conditions from the external user interface. For example, an ATM user interface may have data items like *account ID*, *password*, and *transaction type*. Hence, conditions can be derived such as *valid* and *invalid account ID* and *password*.
- c. Identify data conditions from the data exchanged among software components. For example, an ATM machine encrypts a transaction before sending it to a bank. Hence the conditions for a successful transaction include *valid transaction message* and *successful encryption*.
- d. Identify communication conditions from the logical and physical connections between components. For example, *correct communication protocol* is a condition identified from the communication between an ATM machine and its banking system.
- e. Identify network conditions from the physical connection between distributed subsystems. For example, conditions identified from the connections between an ATM machine and its banking system may include *point-to-point connection*, and *maximum delay of 1 second*.

Relationships Among Conditions

It may not be feasible to exhaustively test all the combinations of conditions. Thus, it is important to analyze the relationships among conditions so that a tester can build a suitable set of test conditions to test the application. Typical relationships among conditions are:

- Independent: two conditions are independent if one can happen regardless of the other.
- Trigger/trigger-by: One condition may trigger the other condition to activate.
- Mutually exclusive: two conditions are mutually exclusive if they cannot exist at the same time. For example, *valid bank account* is a condition that is mutually exclusive with another condition *invalid bank account*.
- Related: two conditions are related to each other if they are used in the same thread, or they are mutually exclusive.

Conditions can also assist in risk analysis. A thin thread is risky if

- 1) It has a condition that has a high probability of failure; or
- 2) It has a condition whose failure can cause serious consequences.

Still, conditions can also be used in carrying out regression testing. A condition may be shared by a set of thin threads. A change in a condition potentially affects all the thin thread that share the same condition, and they are candidates for regression testing [16].

Construct Condition Trees

Conditions can also be grouped and organized into a tree structure to facilitate reuse and management. Similar to thin thread tree, the root of the condition tree is the system under test; a branch node is a collection of conditions related to each other; and a leaf node represents a concrete condition.

2.3 Relationships Between Thin Thread Tree and Condition Tree

The construction of condition tree and that of thin thread tree affect each other.

First, the thin thread tree can lead to identification of conditions. For example, some thin threads may share the same condition, e.g., *successful local withdraw transactions* and *successful remote withdraw transactions* share the same input data condition. Some thin threads may have exclusive conditions, e.g., *successful local withdraw transactions* and *failed local withdraw transactions due to invalid user*

password. By analyzing the thin thread tree, one can identify conditions.

However, analyzing conditions can also lead to identification of new threads. Once the conditions of a thin thread or thin thread group have been identified, the combination of related conditions can lead to discovery of new threads. Thus, construction of the thin-thread tree and the condition tree is an iterative process, and a tester can cross-examine these two trees to identify missing items in both trees.

2.4 Completeness and Consistency (C&C) Checking

Thin threads C&C means that all the E2E system functions are identified without leakage and contradiction and can be traced back to the system definition. Furthermore, all the conditions have been completely and consistently attached to each thin-thread and thin-thread group.

Condition C&C means that all the components, interfaces, and dependencies have been completely specified by the conditions without any conflict. And condition tree should be complete and consistent with the thin thread tree. For each thin thread group, it is important that all the conditions associated with the group have been specified completely and consistently, and all the combinations of related conditions are handled by the thin threads or subgroups within the group. Concrete rules are presented in [19]

3 Test Scenario and Test Case Generation

3.1 Test Scenario Generation

Thin threads identify the atomic E2E functions of the system under test. However, testing is not limited to the execution of atomic functions. It can also be a sequence or a combination of atomic functions. A complex test scenario is used to specify composite thin threads, which are composed using following operators:

- *Sequencing*: A thin thread can be directly followed by another thin thread;
- *Looping*: A thin thread can be repeated multiple times; and
- *Conditioned execution*: This adds the control decision to a complex test scenario.

A complex scenario may be formed using a combination of operators, as shown in Figure 2.

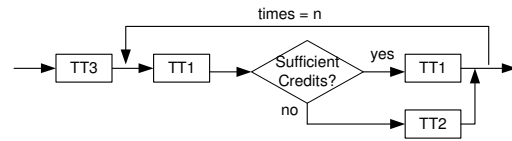


Figure 2. Complex test scenario example

In addition, complex scenarios can be formed not only from thin threads, but also from thin-thread groups and other complex scenarios.

3.2 E2E Test Case Generation

Test cases can be generated from a thin thread or complex scenario in the following ways:

- (1) Identify the input data that satisfy the conditions associated with the thread based on different testing techniques; and
- (2) Determine the expected results from the thread description.

One way to select test inputs is to select those points that are near the boundary of conditions, i.e., boundary testing [9][5]. One also may select test inputs randomly, i.e., random testing [8]. Another way is to classify the inputs into equivalent classes and select typical data in each category, i.e., partition testing. Another way is to generate test cases based on usage, e.g., the usage-based testing in the Cleanroom method [3].

Often, a thin thread (test scenario) is affected by several conditions, and each condition can be satisfied by multiple data. In this case, a decision table is helpful to generate test inputs from different combinations of conditions.

4 Tool Support

E2E testing is complex. A web-based tool [2] has been developed to support the testing. The tool has a three-tier architecture and runs on the J2EE platform using Enterprise JavaBean (EJB) [11], as shown in Figure 3.

- At the backend lie the file/database servers, the storage of test data;
- The middle layer performs the core functions of the E2E testing;
- The front tier is the presentation layer. It provides users with various graphical views of the test data and analysis results.

This tool can facilitate data collection, organization and analysis in an interactive manner. It can also enhance the collaboration and cooperation among software developers, testers, project managers, program managers and oversight organizations.

5 Conclusion

This paper presents a systematic E2E testing design approach, including test specification, test case generation, and tool support. The approach has the following characteristics: (1) It uses both black-box and white-box testing techniques, which provides sufficient information for functional test case design, coverage analysis, result analysis [12], defects identification and software evaluation; (2) Thin threads bridge the gap between high-level use case description [4] and low-level source code implementation; (3) It is a semi-formal approach and thus easy to use; (4) It can be applied to both legacy systems and applications using modern techniques such as OO; (5) It facilitates test scenario and test case reuse; (6) It supports change management so that regression testing and ripple effect analysis [15][18][16] can be carried out properly and efficiently; (7) It supports remote project management and distributed collaboration so that engineers and project managers can work together via the Internet.

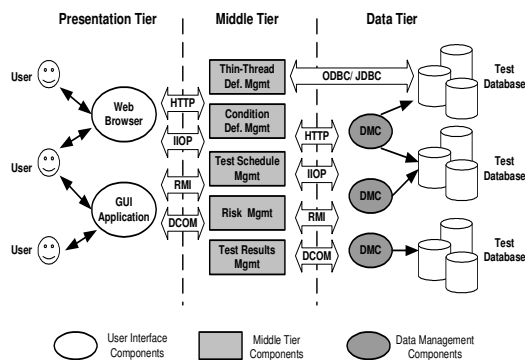


Figure 3. Architecture of E2E Test Management Tool

6 Reference

- [1] X. Bai, W. T. Tsai, R. Paul, K. Feng, L. Yu, "Scenario-Based Business Modeling", Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287, 2001.
- [2] X. Bai, W.T. Tsai, R. Paul, T. Shen and B. Li, "Distributed End-to-End Testing Management", to appear in IEEE Proc. EDOC, 2001.
- [3] M. Dyer, *The Cleanroom Approach to Quality Software Development*, Wiley, New York, New York, 1992.
- [4] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison Wesley, Reading, MA, 1999.
- [5] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*, CRC Press, 1995.
- [6] S. Kirani and W. T. Tsai, "Specification and Verification of Object-Oriented Programs", technical report, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, 1994.
- [7] D. C. Kung, P. Hsia, and J. Gao, *Testing Object-Oriented Software*, IEEE Computer Society Press, Los Alamitos, CA, 1999.
- [8] P. Loo, W. T. Tsai, W. K. Tsai, "Random Testing Revisited", *Information and Software Technology*, Vol. 30, No. 7, Sept. 1988, pp. 402-417.
- [9] G. J. Myers, *The Art of Software Testing*, New York, Wiley Inter-science, New York, New York, 1979.
- [10] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw Hill, 5th edition, 2000.
- [11] E. Roman, *Mastering Enterprise JavaBeans™ and the Java™ 2 Platform, Enterprise Edition*, John Wiley & Sons, Inc. New York, New York, 1999.
- [12] W. T. Tsai, R. Paul, W. Shao, S. Rayadurgam, and J. Li, "Assurance-Based Y2K Testing", 1999 Proc. 4th IEEE International Symposium on High-Assurance Systems Engineering, pp. 27-34.
- [13] W. T. Tsai, Y. Tu, W. Shao and E. Ebner, "Testing Extensible Design Patterns in Object-Oriented Frameworks through Hierarchical Scenario Templates", Proc. COMPSAC, 1999, pp. 166-171.
- [14] W. T. Tsai, V. Agarwal, B. Huang, R. Paul, "Augmenting Sequence Constraints in Z and its Application to Testing", Proc. 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology, 2000, pp. 41-48.
- [15] W. T. Tsai, X. Bai, R. Paul, G. Devaraj, and V. Agarwal, "An Approach to Modify and Test Expired Window Logic", Proc. of IEEE Asia-Pacific Conference on Quality Software, 2000, pp. 99-108.
- [16] W.T. Tsai, X. Bai, R. Paul, and L. Yu, "Scenario-Based Functional Regression Testing", to appear in *IEEE Proceedings of COMPSAC*, 2001.
- [17] DoD OASD C3I Investment and Acquisition, "Year 2000 Management Plan", 1999.
- [18] DoD OASD C3I Investment and Acquisition, "Repairing Latent Year 2000 Defects Caused by Date Windowing", 2000.
- [19] DoD OASD C3I Investment and Acquisition, "End-to-End Integration Testing Guidebook", 2001.