Scenario-Based Modeling And Its Applications

Xiaoying Bai*, Wei-Tek Tsai*, Ray Paul+, Ke Feng*, Lian Yu*

**Department of Computer Science and Engineering*
Arizona State University
Tempe, AZ 85287

+*Investment and Acquisition*
Department of Defense
Washington, DC

Abstract

Use-oriented techniques are widely used in software requirement analysis and design. Use cases and usage scenarios facilitate system understanding and provide a common language for communication. This paper presents a scenario-based modeling technique and discusses its applications. In this model, scenarios are organized hierarchically and they capture the system functionality at various abstraction levels including scenario groups, scenarios, and sub-scenarios. Combining scenarios or sub-scenarios can form complex scenarios. Data are also separately identified, organized, and attached to scenarios. This scenario model can be used to cross check with the UML model. It can also direct systematic scenario-based testing including test case generation, test coverage analysis with respect to requirements, and functional regression testing.

1. Introduction

Software requirement specifications provide the foundation for system comprehension, design, testing and maintenance. This paper proposes a semiformal model for identifying and organizing system requirements.

The model represents the system from three perspectives: 1) *Functional View*: this identifies and organizes the desired functions and their relationships. A s*cenario* describes a function from the end user' point of view. A s*cenario group* is a collection of functional related scenarios or scenario groups. A s*ub-scenario* is a decomposition of a scenario, representing a sub-function provided by the software; 2) *Data View*: this identifies and organizes data and their relationships. Data are analyzed and modeled using concepts like inheritance and composition. They are attached to scenarios as inputs, outputs, or intermediate, showing how they are used in the system; and 3) *Usage View*: this identifies usages and processes by *complex scenario*. A *complex scenario* is formed by combining scenarios or sub-scenarios using control constructs such as *sequencing*, *conditioned*, *concurrent*, and *iterative*. A complex scenario shows how multiple functions can collaborate to implement specific application.

Use-oriented software engineering has gained wide attentions since early 1990s [4][7]. Use cases are used to gather anticipated system usages without design considerations, and to represent them in a common language to improve the communications among various parties including customers, software analysts and software developers. A use case can be instantiated into multiple scenarios, which describe partial execution orders and message flows through subsystem components. The existing literature shows a diversity of methods of defining and using use cases and scenarios, such as UML (Uniform Modeling Language) [3], UCM (Use Case Maps) [1], and MSC (Message Sequence Charts) [19]. UCM and MSC are developed for real-time systems, specifically MSC is closely related to a state model, and UCM can be mapped to a system structure for analysis. In general, use cases are high-level descriptions. As recommended by Jacobson, the appropriate volume should be no more than 70-80 use cases for large systems, that means 20-50 in general cases [7]. In contrast, scenarios are detailed descriptions with detailed design information incorporated. It is common for an integrated system to have thousands of scenarios. Usually, one may generate many scenarios from a single use case.

This paper proposes a scenario model to specify systems and to generate test cases as shown in Figure 1. The key points are: (1) Scenarios are derived from system requirements; (2) The scenario model can cross check with the software design represented by the UML (section 3); and (3) Systematic testing can be performed using the scenario model including test scenarios and cases generation (section 4.2), testing analysis such as test coverage analysis with respect to requirements (section 4.3), and additional testing such as regression testing (section 4.4) [10][11][20].

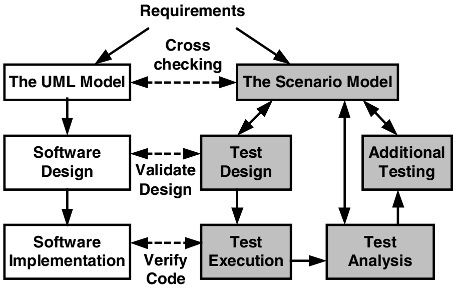


Figure 1. Scenario-based testing process

This scenario model has following features.

- Scenarios are hierarchically organized for better management. The hierarchical organization reflects scenario functional dependencies and composition/decomposition relationships. (a) An atomic scenario represents a basic function; (b) Related scenarios can be composed into scenario groups level-by-level; (c) A scenario can be decomposed into sub-scenarios level-by-level; and (d) Connected by control constructs, scenarios can also be composed into complex scenarios. The organization can also improve completeness and consistency checking (section 2.1).
- Scenario template specifications provide information to improve analysis. Such detailed information allows scenario dependency analysis, such as functional dependency, data dependency, and control dependency. Dependency analysis provides the basis for change management, ripple effect analysis [5][15][17][18] and regression testing [8][14]. It is also useful for system comprehension such as to remove redundancy and to identify critical parts.
- Scenario specifications can support abstractions at various granularities, enabling progressive system comprehension, and bridging the gap between use cases and scenarios.

The next section presents the scenario model. Section 3 observes the relationship between the scenario model and UML mode. Section 4 introduces the approach that the scenario model can support for testing. Section 5 briefly describes the supporting tool. And section 6 concludes the paper.

2. The Scenario Model

2.1 Modeling Scenarios

Figure 2 shows an example of the scenario model, in which the functions of a *Banking System* are decomposed progressively into multiple levels of scenario groups, scenarios, and sub-scenarios.

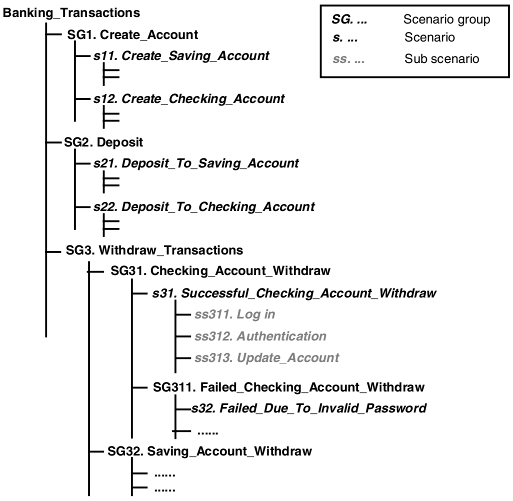

Banking_Transactions
SG1. Create_Account
s11. Create_Saving_Account
s12. Create_Checking_Account
SG2. Deposit
s21. Deposit_To_Saving_Account
s22. Deposit_To_Checking_Account
SG3. Withdraw_Transactions
SG31. Checking_Account_Withdraw
s31. Successful_Checking_Account_Withdraw
ss311. Log in
ss312. Authentication
ss313. Update_Account
SG311. Failed_Checking_Account_Withdraw
s32. Failed_Due_To_Invalid_Password
......
SG32. Saving_Account_Withdraw
......
......
SG. ... Scenario group
s. ... Scenario
ss. ... Sub scenario

Figure 2. An example of modeling scenarios

Scenarios

A scenario, like *s11* and *s12*, is specified semi-formally using a template definition as shown in Table 1.

Scenario Groups

Scenarios can be grouped hierarchically into multiple levels of scenario groups. For example, scenarios *s11* and *s12* are arranged into scenario group *SG1*, which is further grouped with *SG2* and *SG3* into super group *SG0*.

A typical way of identifying scenarios and scenario groups is by functional decomposition. The overall functions of a system are first divided into distinct features represented by high-level scenario groups, which can be decomposed level-by-level into sub-groups until basic the scenarios are identified.

A scenario group can be defined using the same template as scenario definition. In practice, developers can defer the filling of the attributes of a group to its child scenarios. And the definition of a scenario group should be consistent with specifications of its children.

Table 1.Scenario template definition

Attribute	Explanation
ID	A globally unique identification of the scenario that may be generated by system
Name	A user-defined identification. Certain naming conventions can be used to facilitate retrieval of scenarios
Context	The circumstances under which the scenario is used.
Reference	The linkage between the scenario definition and other software artifacts like code and documents
Inputs	The data, actions, and events required to trigger the corresponding function
Outputs	The data, products and deliveries of the execution
Parent	The scenario group it belongs to
Subs	The sub-scenarios contained in it
Precedents	The scenarios that should be executed prior to this one.
Successors	The scenarios that cannot execute unless current scenario finishes
Description	A brief summary of objectives and functions of the scenario

Sub-Scenarios

Scenarios may share some functionalities provided by commonly used software components. For example, "authentication" is required for all banking transactions. Sub-scenarios decompose a scenario into relatively independent functional parts. For example, scenario *s31* is decomposed into sub-scenarios *s311*, *s312*, and *s313*.

Sub-scenarios are identified with an insight into system high-level design, such as:

- Decomposed by layers
- Decomposed by tiers
- Decomposed by configuration
- Decomposed by service providers

Scenario Relationships

Scenarios may relate to each other with respect to their execution paths [11][13]:

- Path-contained: The execution path of a scenario can be a part of the path of another scenario.
- Path-identical: Two scenarios have the same paths. In this case, they may share a certain set of attributes, such as conditions.
- Path-independent: Two scenarios are said to be path-independent if they have completely different paths without any overlap.

Scenarios may also relate to each other with respect to conditions [11][13], and typical condition relationships 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.
- Related: two conditions are related to each other if they are used in the same scenario, or they are mutually exclusive.

Such relationships can assist in risk analysis. A scenario may be risky if

1) It contains the software component(s) who has a high probability to fail, and/or whose failure may cause serious consequences.
2) It has a condition that has a high probability of failure, and/or whose failure can cause serious consequences.
3) It locates on a path that covers many software components.

Relation analysis can facilitate the validation of complex scenarios (section 2.3). It can also help for dependency analysis such as execution dependency and condition dependency, and regression testing [14].

Scenario Dependencies

Dependency analysis is important for system change management. In this model, scenarios are dependent in three ways:

- *Functional Dependency*: Scenarios are functional dependent if they belong to the same scenario group in which scenarios represent the common function from different aspects, or they contain the same sub-scenario(s) in which the rely on the same services.
- *Data Dependency*: Scenarios are data dependent with respect to their template definitions. Two scenarios are
  1) *Input dependent* if they are triggered by the same data, events, and actions.
  2) *Output dependent* if they produce the same results such as output data and products.

3) *Execution dependent* if they cover common software components such as code modules and interfaces.

4) *Condition dependent* if they are affected by the same conditions.

5) *Reference dependent* if they reference to common software artifacts such as document, code, and persistent data;

- *Control Dependency*: Two scenarios are control dependent if they are involved in the precedent /successor relationships as defined in the template.

Completeness and Consistency Checking

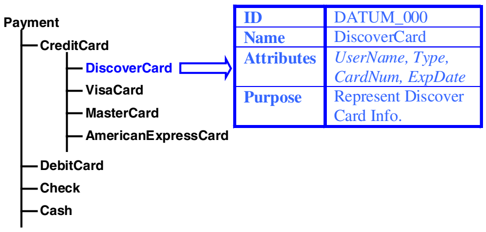
To avoid conflictions, duplications, and omissions, some generic C&C rules are as follows:

- Scenario groups are mutually exclusive. That is, for any pair of distinct scenario groups *G* and *G'*, if a scenario *s* belongs to *G*, *s* must not belong to *G'*;
- There are no duplicate scenarios. That is, for any two scenarios *s* and *s'*, they differ on at least one of the attributes of *inputs*, *outputs*, and *execution path*;
- There are no overlapping functionalities between sub-scenarios within the same parent scenario. That is, each sub-scenario represents a separate part of the overall function; and
- For each scenario identifying expected system behaviors with normal inputs, at least two corresponding scenarios exist identifying behaviors under abnormal inputs and exceptional conditions respectively.

2.2 Modeling Data

A scenario is a process that accepts, processes and transforms data, which may be inputs from end users, internal representations, or from persistent storage. Results are generated and presented to user by output data.

Data are captured, specified, and organized separately, and are associated with scenarios. Data are captured from perspectives like *business object*, *product*, *document* and *message*. *Business objects*, such as user account and paycheck, are the targets of operations and functions. *Product*, such as coffee dispensed by a vending machine, is a major form of system outputs. *Document*, such as report and file, is another popular form of outputs. *Message*, such as packed string and list, is a kind of lightweight information that is passed though subsystems. Data are represented and organized using the object-oriented (OO) approach. Each datum is attached with a form, specifying its ID, name, attributes, and purposes description. Data are related in two ways: 1) *Inheritance*, which identifies the generalization and specialization relationship; and 2) *Composition*, which identifies the part-of relationships. Related data are arranged into a hierarchical structure. Figure 3 shows part of an example data model for payment and the data specification form.



ID	DATUM_000
Name	DiscoverCard
Attributes	*UserName, Type, CardNum, ExpDate*
Purpose	Represent Discover Card Info.

Figure 3. An example of modeling data

The model integrates the data view with the functional view by associating the identified data with scenarios. Data are attached to scenario and sub-scenarios as inputs and outputs. Specially, data that are inputs to a sub-scenario but not inputs to the parent scenario are intermediate data to the scenario.

2.3 Modeling Complex Scenarios

A complex scenario is defined for two purposes: 1) as a composition of atomic scenarios to model complex usages of the system [12]; and 2) as a combination of sub-scenarios to model the execution order among the sub-scenarios belong to a scenario.

In a complex scenario, scenarios/sub-scenarios are connected using control constructs such as 1) s*equencing*, to define the following relationships; 2) l*ooping*, to define the repeated executions; 3) *concurrent*, to define the simultaneous executions; and 4) c*onditioned*, to add decisions to complex scenario. A pseudo code based expression is used to describe the complex scenario.

Suppose there is a data *D0* "*user account*". Taking following three scenarios in Figure 2:

- *s12* "*Create_Checking_Account*";
- *s22* "*Deposit_To_Checking_Account*"; and
- *s31* "*Successful_Checking_Account_Withdraw*".

Figure 4 gives an example of complex scenario.

```
s12;
s22;
SWITCH ( D0 )
     CASE less than 100:
               BEGIN
                         s22;
               END
     CASE greater than 100:
               BEGIN
                         IF ( enough balance ) THEN
                                   BEGIN s31; END
                         ELSE
                                   BEGIN s22; END
               END
```

1. Create a user account
2. deposit money to the created user account
3. If the account balance is less than 100, deposit money again
4. if the account balance is greater thatn 100 and has enough to withdraw, execute the withdraw transaction
5. if the account balance is greater than 100 but has no enough credits, exercise the deposit transaction

Figure 4. An example of modeling complex scenario

Scenario relationship and dependency analysis (section 2.1) can be used to check the validity of a complex scenario. Some rules are:

- Scenarios that are connected via *conditioned* operators, such like *if_then_else*, and *switch_case*, should be *condition dependent* with respect to the condition.
- Scenarios that are connected via *sequencing* operators should be *control dependent.*
- Scenarios that are connected via concurrent operators should be *input dependent* with respect to particular triggering actions or events.

For example, scenarios *s12*, *s22*, and *s31* are input dependent with respect to data *D0*. The example complex scenario in Figure 4 may become invalid if *s22* is replaced by *s21* "*deposit money to saving account*".

3. Relationship with UML

3.1 Application to UML Use Case Model

UML provides a use case view to capture the behavior of a system, a subsystem, or a class as it appears to outside users [3][4]. A use case describes an interaction between the system and its actors. A use case diagram describes the relationships among use cases such as association, extension, generalization, and inclusion.

Scenarios and scenario groups represent the externally visible functionalities. A group can be elaborated into simpler subgroups and scenarios at various granularities, and a scenario can be factored into sub-scenarios, the coherent functional units.

The use cases in UML can be mapped directly to scenarios or scenario groups. Related use cases can be arranged into the hierarchical structure according to their relationships: (a) use cases with no super-ordinate can be mapped to the high-level scenario groups; and (b) the subordinators (e.g. generalized or included use cases) to a use case can be mapped to the children to the scenario group representing the use case.

In addition, the scenario model can be further expanded to explore scenarios and sub-scenarios in finer granularity, and to reflect the requirements as completely and exactly as possible.

For example, the scenario model in Figure 2 is an extension to the UML use case diagram in Figure 5.

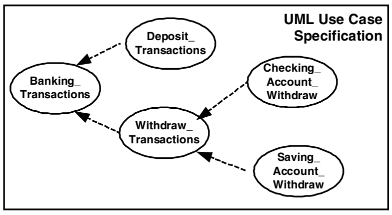

UML Use Case Specification
Deposit_ Transactions
Banking_ Transactions
Checking_ Account_ Withdraw
Withdraw_ Transactions
Saving_ Account_ Withdraw

Figure 5. An example of extension

3.2 Application to UML Static Model

UML class diagram provides a static view of application concepts, in terms of classes and their relationships including generalization, association, and dependency [3]. In the scenario model, data are also modeled using OO concepts. Hence, one can derive the class diagram from the data model, either directly or indirectly.

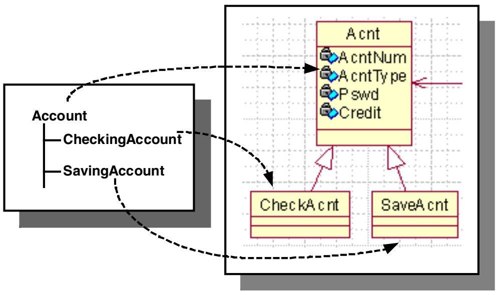


Figure 6. An example of data mapping

Figure 6 gives an example of data mapping, in which the data identified in the scenario model are mapped to the classes in UML, and their generalization relationships are mapped to the inheritance relationships between the classes.

3.3 Application to UML Activity Model

UML uses activity diagram to model computations and workflows [3]. An activity graph is made up of activities, objects, decisions, synchronizations, and swimlanes. An activity represents an execution of a task; an object describes an input/output of an activity; a decision locates branches based on a guard condition; a sync bar is the synchronization point for concurrent threads; and swimlanes separate the workspace into distinct regions for organizations and assign activities to organization units.

An activity can be considered as a scenario or sub-scenario, and the presented scenario model is an effective peer technique to verify the activity diagram from following perspectives.

- Complex scenarios can verify the control flow of activity diagram, such as (a) whether activities are modeled in the consistent sequence; (b) whether conditions in the complex scenarios are identified by guards in the corresponding activity diagram and are placed at the right positions; and (c) whether selective logics like *if_then_else* and *switch* in complex scenario are represented by decisions in the activity diagram, and the branching conditions are specified correctly.
- Scenario input/output dependencies can verify the data flow of the activity diagram.
- Scenario precedent/successor pairs can verify the synchronization of the activity diagram. If a scenario/sub-scenario is modeled as an activity after a synch point, all its precedents should be represented by activities before the point, and vise versa.

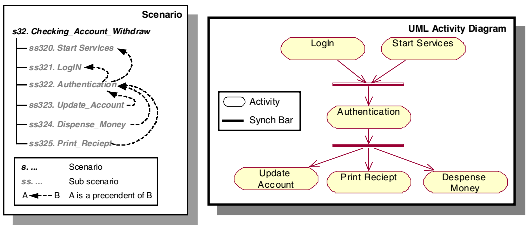


Figure 7.An example of verification

Figure 7 gives an example to illustrate the relationship between the scenario model and the UML activity model. In this example, the activity diagram models the relationships among a set of sub-scenarios {*ss320, ss321, ss322, ss323, ss324, ss325*}. Each sub-scenario in the scenario model is represented by an activity in the activity diagram. All the precedents of *ss322* are represented by activities that are synchronized before the activity representing *ss322*, while those successor activities are synchronized after it.

4. Application to Testing

4.1 Test Design

Each scenario and sub-scenario can be used as a test scenario to test a subsystem or clusters of integrated subsystems [13][20]. Complex test scenarios can be generated from complex scenarios, by combining scenarios and sub-scenarios. In general, numerous complex test scenarios can be derived based on various criteria such as path coverage, condition coverage and loop coverage [13].

While simple test scenarios verify system correctness with individual functions, complex test scenarios can test multiple functions executed in different orders to reveal defects that are hard or unable to be detected by single tests. After each scenario is separately tested and approved, testing with complex test scenarios can verify system stability, reliability, recoverability, performance, and other non-functional requirements.

In some cases, one may need to develop test driver and stubs to set up the context and to clean up after execution.

Test cases can be generated via examining the inputs and outputs of the scenario, based on various testing techniques such as partition testing, boundary value testing, random testing, and equivalent partition [6][16].

In most cases, each test scenario is associated with multiple data and each data has multiple attributes. To generate test data, one can apply different technique to different data and even to different attributes, and test cases are obtained from various combinations.

Figure 8 is an example of test scenario generation. The complex scenario is instantiated into six sequences of scenarios, three for looping with 3 times and others for looping with 0 time. On of the test scenario is then expanded to its full definition in terms of setups, inputs, actions, and expected outputs.

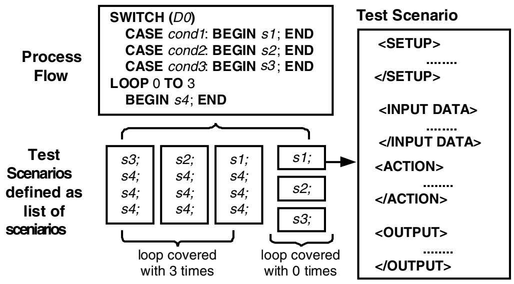


Figure 8. An example of generating test scenarios

Figure 9 gives an example of test case generation using the decision table technique. In this example, two data, *A* and *B*, are attached to a scenario, each of which has two attributes {*a1*,*a2*} and {*b1*,*b2*} respectively. Each of the attributes can be further divided into two equivalent classes, say {*a11*,*a12*}, {*a21*,*a22*}, {*b11*,*b12*}, and {*b21*,*b22*}. In Figure 9, table (a) shows the definition of the two data, table (b) shows the decision table to select values for each attribute and each data, and table (c) shows the decision table to generate test cases by combining different data values.

a. Data Definition

Datum	Attributes
A	*{a1,a2}*
B	*{b1,b2}*

b. Decision Table Identify Data Values

A.a1=ap11	T	T	F	F	*B.b1=bp11*	T	T	F	F
A.a1=ap12	F	F	T	T	*B.b1=bp12*	F	F	T	T
A.a2=ap21	T	F	T	F	*B.b2=*bp*21*	T	F	T	F
A.a2=ap22	F	T	F	T	*B.b2=bp22*	F	T	F	T
A	*A1*	*A2*	*A3*	*A4*	*B*	*B1*	*B2*	*B3*	*B4*

c. Decision Table Identify Test Cases

A=A1	T	T	T	T	F	F	F	F	F	F	F	F	F	F	F	F
A=A2	F	F	F	F	T	T	T	T	F	F	F	F	F	F	F	F
A=A3	F	F	F	F	F	F	F	F	T	T	T	T	F	F	F	F
A=A4	F	F	F	F	F	F	F	F	F	F	F	F	T	T	T	T
B=B1	T	F	F	F	T	F	F	F	T	F	F	F	T	F	F	F
B=B2	F	T	F	F	F	T	F	F	F	T	F	F	F	T	F	F
B=B3	F	F	T	F	F	F	T	F	F	F	T	F	F	F	T	F
B=B4	F	F	F	T	F	F	F	T	F	F	F	T	F	F	F	T
Test case	*T1*	*T2*	*T3*	*T4*	*T5*	*T6*	*T7*	*T8*	*T9*	*T10*	*T11*	*T12*	*T13*	*T14*	*T15*	*T16*

Figure 9. An example of generating test cases

4.2 Test Coverage Analysis

In most existing testing techniques, coverage is evaluated with respected to the program design, such as statement coverage, condition coverage, and method/class coverage. The scenario model enables coverage analysis with respect to requirements represented by scenarios. It also allows for coverage analysis combined with risk analysis and usage analysis [13]. Metrics that may be used include:

- Number of scenarios used/Total number of scenarios.
- Number of scenario groups used/Total number of scenario groups.
- The coverage of scenarios in each scenario group.
- The percentage of high-risk scenarios covered.
- The percentage of often-used scenarios covered.

4.3 Regression Testing

Change is inevitable and regression testing is important to ensure the correctness and integrity of the modified system [8][15][21]. However, most regression testing techniques are code based, such as program dependency graph and program slicing [5][18]. The scenario model enables scenario dependency and traceability analysis, which can be used for scenario slicing and ripple effect analysis [14].

Based on the generic REA (Ripple Effect Analysis) process [17], Figure 10 shows the iterative process of scenario-based regression tests selection: (1) A change request is first mapped to a set of potentially affected scenarios; (2) By traceability analysis, one can trace a scenario to its associates and identify and validate the set of other affected other software artifacts; (3) By dependency analysis, one can also identify other potentially affected scenarios via performing with preferred slicing criteria; and (4) Scenario identification and validation iterate until no more ripples exist.

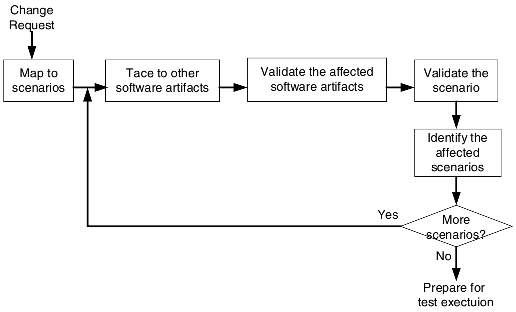

Change Request
Map to scenarios
Tace to other software artifacts
Validate the affected software artifacts
Validate the scenario
Identify the affected scenarios
Yes
More scenarios?
No
Prepare for test exectuion

Figure 10. The scenario-based REA process

5. Tool Support

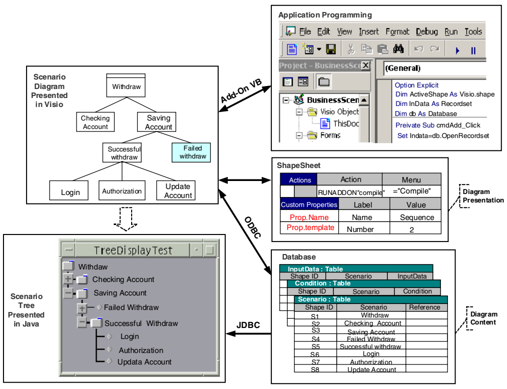
A tool is developed to support the construction of the scenario model. The tool uses MVC (Model/View/Controller) architecture and supports a GUI as shown in Figure 11:

- At the view part, it supports visual modeling using MS Visio 2000;
- At the model part, the graphic presentation is stored in Visio Shape Sheets, and diagram semantics are stored in test database; and

- At the controller part, applications are developed in MS Visual Basic to monitor user activities and to ensure the consistency between model and view.

The tool can also export the model in certain standard data format such as XMI/XML, and exchange data and integrate with other products [9].

The scenario model are stored in the database and exported into a tree structure in Java. In this way, the tool is being integrated with the End-to-End testing tool [2], which aims to support testing management in a distributed environment.



```
Option Explicit
Dim ActiveShape As Visio.shape
Dim InData As Recordset
Dim db As Database
Preivate Sub cmdAdd_Click
 Set Indata=db.OpenRecordset
```

Actions	Action	Menu
	RUNADDON"compile"	="Compile"
Custom Properties	Label	Value
Prop.Name	Name	Sequence
Prop.template	Number	2

InputData : Table

Shape ID	Scenario	InputData

Condition : Table

Shape ID	Scenario	Condition

Scenario : Table

Shape ID	Scenario	Reference
S1	Withdraw	
S2	Checking Account	
S3	Saving Account	
S4	Failed Withdraw	
S5	Successful withdraw	
S6	Login	
S7	Authorrization	
S8	Update Account	

Figure 11.The structure of the supporting tool

6. Conclusion

The model proposed in this paper improves the scenario-based technique by enforcing the hierarchical definition and organization of scenarios modeling at various abstraction levels. It can serve as a central model to direct the lifecycle process of software development, from requirements comprehension to testing. It can also integrate with other techniques and products to improve modeling capabilities.

References

[1] D. Amyot, "Use Case Maps as a Feature Description Notation," Fireworks Feature Constructs Workshop, UK, 2000.

[2] X. Bai, W.T. Tsai, R. Paul, T. Shen and B. Li, "Distributed End-to-End Testing Management", Proc. of IEEE EDOC, 2001, pp. 140-151.

[3] G. Booch and T. Quatrani, *Visual Modeling With Rational Rose and UML*, Addison-Wesley, Reading, MA, 1998.

[4] I. Jacobson, M. Christerson, P. Jonson, and G. Overgarrd, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison Wesley, Reading, MA, 1992.

[5] J. K. Joiner and W. T. Tsai, *Ripple Effect Analysis, Program Slicing and Dependency Analysis*, TR-93-84, Computer Science Department, University of Minnesota, 1993.

[6] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*, CRC Press, 1995.

[7] D. Kulak and E. Guiney, *Use Cases: Requirements in Context*, Addison Wesley, Reading, MA, 2000.

[8] A.K. Onoma, W.T. Tsai, M. Poonawala, and H. Suganuma, "Regression Testing in an Industrial Environment", Communications of the ACM, Vol. 41, No. 5, May 1998, pp. 81-86.

[9] R. Paul, W.T. Tsai, B. Li, and X. Bai, "XML-Based Test Assets Management", Proc. of IEEE ER2001, Yokohama, Japan, 2001.

[10] R. Paul, "End-to-End Integration Testing: Evaluating Software Quality in a Complex System", Proc. of Assurance System Conference, Tokyo, Japan, 2001, pp. 1-12.

[11] R. Paul, "End-to-End Integration Testing", Proc. of IEEE APAQS, 2001.

[12] W.T. Tsai, X. Bai, L. Yu, and R. Paul, "Generation of Composite Test Scenarios and Test Cases Based on End-to-End Testing Specification", Department of Computer Science and Engineering, Arizona State University.

[13] W.T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-to-End Integration Testing Design", Proc. of IEEE COMPSAC, 2001, pp. 166-171.

[14] W.T. Tsai, X. Bai, R. Paul, and L. Yu, "Scenario-Based Functional Regression Testing", Proc. of IEEE COMPSAC, 2001, pp. 496-501.

[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 APAQS, 2000, pp. 99-108.

[16] W. T. Tsai, Y. Tu, W. Shao and E. Ebner, "Testing Extensible Design Patterns in Object-Oriented Frameworks through Hierarchical Scenario Templates", Proc. of IEEE COMPSAC, 1999, pp. 166-171.

[17] W.T. Tsai, R. Mojedehbakhsh, F. Zhu, "Ensuring System and Software Reliability in Safety-Critical Systems", Proc. of IEEE ASSET, 1998, pp. 48-53.

[18] Y. Wang, W.T. Tsai, X.P. Chen and S. Rayadurgam, "The Role of Program Slicing in Ripple Effect Analysis", Proc. of Software Engineering and Knowledge Engineering, 1996, pp. 369-376.

[19] *MSCs: ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*, ITU-Y, Geneva, 1996.

[20] DoD OASD C3I I&A, *End-to-End Integration Testing Guidebook*, 2001.

[21] DoD OASD C3I I&A, *Repairing Latent Year 2000 Defects Caused by Date Windowing*, 2000.