AT&T and Teradyne Software & System Test

Use-Cases Are Not Requirements

Date: March 19, 1999

From: Steve Meyer
AT&T
E-mail sameyer@att.com

Larry Apfelbaum
Teradyne Software & System Test
E-mail larry@sst.teradyne.com

ABSTRACT: Use Cases are Not Requirements

Much work has been done in recent years to improve the process used to develop software. Increasingly, use of object oriented methodologies have become standard. One aspect common to many of them is the increased emphasis on Requirements Modeling. Capturing requirements accurately is essential to developing correct software. A currently popular Object Oriented method for requirements capture has been use cases, use scenarios or Use Case Requirements Modeling. A use case is a mechanism where an engineer can describe a specific scenario for the system, illustrating one or more key characteristics of its business functionality and processes. Use scenarios provide a valuable means for a team to review the proposed business solutions under a limited number of *specific conditions*. It is a communications tool valuable in the specification, analysis, development and testing of a system. Although a Use case approach is very effective at capturing business functionality, they are somewhat lacking in capturing usage and behavior characteristics of the system.

Outline

1. Introduction

Most systems can have very large numbers of potential usage scenarios. It is not practical for designers or system engineers to explicitly describe all usage scenarios for all Use Cases. These scenarios often only implicitly define the actual *behavioral* requirements of the system. To completely define the requirements illustrated by a usage scenario one must explicitly enumerate each possible sequence of actions. As the number of requirements becomes large, manual specification of all possible usage scenarios becomes increasingly difficult and the possibility of missing, incomplete or ambiguous behavior increases.

At a minimum Uses Case Requirements Modeling provides use scenarios which describe the business functionality to be captured as a set of enumerated steps. This may include some pre/post conditions and some exception processing alternatives. Ivar Jacobson's Requirements Model consists of: Use Case Model, Domain Object Model and Interface Descriptions. A thorough approach to Use Case Requirements Modeling can include:

- Uses Cases described via a well defined template
- Use Case diagrams to illustrate high-level use case relationships
- Domain Object Model depicting objects in the business domain and their attributes
- Interface diagrams, Business Rules, User Interface Descriptions

At best, Use cases fail to provide information that could greatly enhance subsequent steps in the development process. An approach of using behavioral modeling is an opportunity to further "nail-down" requirements in a way that can provide continuity from System Engineering through test.

This paper presents a methodology where a behavioral model describing the systems actions can augment use cases as a compact means of describing them. In addition a process of overlaying the systems requirements onto this model will provide a very efficient mechanism for determining when any arbitrary use scenario generated by the behavioral model has covered any of the specified requirements. Benefits of this approach include easy analysis of behavior, rapid response to changes in specifications of requirements; tests generation correlated to requirements as well as automated generation of requirements based tests.

2. What Are Use Cases?

Use Case Requirements Modeling is one of the first steps in an Object Oriented approach to systems development. Independent of the subsequent analysis and design methodology used, Use Cases provide a superior method for communicating the business functionality to be developed. Traditional thinking maintains requirements describe the "what" is required, whereas subsequent development steps translate from the "what" to the "how". In spite of improvements in requirement specifications this thinking has led to a gap between requirements and behavior. Use Cases describe requirements for a developer's translation; from a function but not usage point of view. They lack the ability to portray the business needs from a behavioral or user perspective. Behavioral modeling techniques and tools are now available and sufficiently mature that can bridge this gap between function and behavior. One such behavioral approach models requirements in terms of states and transitions between states. Use Case

requirements can now be extended with behavioral modeling to add context and organization. A sort of middleware between the **What** and the **How**.

2.1 Use Case Descriptions

A Use Case is a specific way an actor [a person, or process] uses the solution/system by performing some part of the functionality. A use case specifies the interaction between the actor(s) and the solution/system and describes the functionality to be performed. Typically these descriptions consist of an enumerated set of steps that are needed to be performed for the functionality to reach a successful conclusion. Pre/post conditions and limited exception conditions are often included. Frequency and arrival rate information is almost never included.

2.2 Use Case Diagrams

Use case diagrams are often used to capture relationships between use cases. Certain modeling conventions can be used to further define use case relationships. A well-structured diagram with well-focused use cases is valuable for adding clarity to large models. Following are a couple of trivial examples:

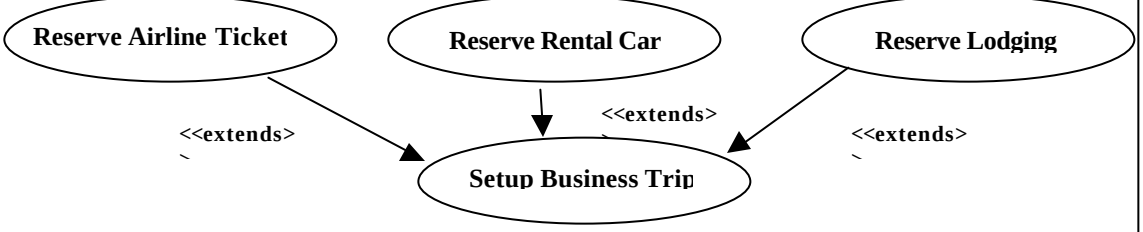


Figure 1 Use Case Diagram for Business Trip

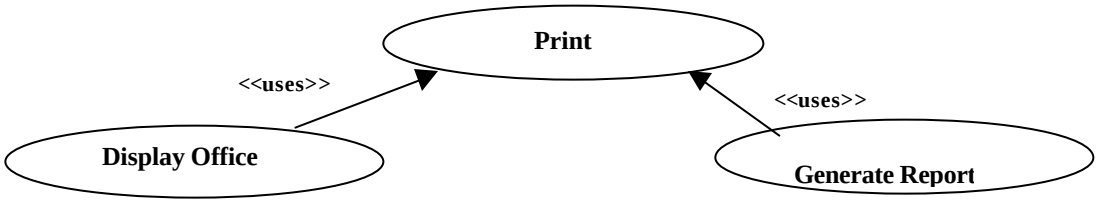


Figure 2 Use Case Diagram for Print System

2.3 Additional Supporting Documentation

Other documentation commonly employed to support Use Cases include: the domain Object model, interaction diagrams, business rules and user interface specifications. Each adds a level of detail to support development efforts by providing information relevant to the problem domain.

3. What are Requirements

I think it would be safe to say that, "The customer wants a solution (system) that meets functional requirements." There may often be constraints in terms of response time or other concerns but it is the

requirements that describe the intent. Webster's defines a requirement as "something required: something wanted or needed". Within the engineering discipline the definition [IEEE 729, Dorfman & Thayer[1]] has focused more on the reasons why we have them: "1) a software capability needed by a user to solve a problem or achieve an objective; 2) a condition or capability that must be met or possessed by a system or a component in order to satisfy a contract, specification standard or other formally imposed documentation." The objective of the requirements phase in a development project is to be able to communicate the needs of the users of a system to the entire team. To accomplish this we need to first verify that the requirements are correct then as the various part of the development team work on the project we need a means to correlate the results back to the requirements themselves.

A key aspect of a software requirement specification is to describe what the software is to do without describing how it is to do it. Davis[2] has defined it as a two phased process, problem analysis and product description. In problem analysis the team will interview people who understand the problem and it's constraints until they have a thorough understanding of the problem. Next in the product description phase the team uses this understanding to define the external behavior of a product that will solve the problem. The solution may not be complete but the issues around the tradeoffs made should all be defined and understood. In addition to describing what is desired it is often just as important to define what is not. Defining the behavior a system should not exhibit is also important in communicating clearly the problem to be addressed.

Many engineers would rather define how they think something should be built rather than focus on what the customer cares about. The answer to the how question is best covered in separate phases known as a functional and design specifications.

3.1 Why are Requirements Important

It is very important to define the requirements accurately, and unambiguously. There have been many studies documenting the cost of repairing a defect, all define a clear escalation of the cost as the development process progresses. An example of this is shown in Table 1, it is from Gause & Weinberg[3] and references data from a Boehm[4] study of 63 software projects from leading corporations (IBM, GTE, TRW).

Phase in Which Found	Cost Ratio
Requirements	1
Design	3-6
Coding	10
Development Testing	15-40
Acceptance Testing	30-70
Operation	40-1000

Table 1 Relative Cost to Fix an Error

It should be obvious to anyone that the business impact of detecting and repairing defects at the requirement stage is immense. The escalation in cost is due to two primary factors, 1) the delay from when the defect was introduced until it was detected and, 2) the cost of the rework involved to repair the defect. The delay is introduced because the detection process, also known as testing, verification or review, is based on determining a difference from the article under test and an oracle, a source of truth. The problem feeds on itself if the basis for the validation is also flawed. If the information used to drive a stage of the process is flawed then even if the process works perfectly, the output will also be flawed.

Given that each stage of the development process is not perfect what we have is a larger and larger proportion of defects. Table 2[5] describes the compounding affects of defects.

Requirements	Functional	Design	Code
Correctly defined requirements	Correctly defined functional description	Correctly designed specifications	Correctly coded software
Incorrectly defined requirements: missing, misunderstood, ignored, outdated, unneeded, unspoken (assumed)	Erroneous functional specs based on incorrectly defined requirements	Design errors based on incorrectly defined requirements	Coding errors based on incorrectly defined requirements
	Incorrectly defined functional specs	Design errors based on incorrectly defined functional specs	Coding errors based on incorrectly defined functional specs
		Design errors	Coding errors based on design errors
			Coding errors

Table 2. Compounding Effect of Defects, even good work on a faulty base will result in a defective product

This is analogous to the problems encountered in determining the yield of a multi-stage manufacturing process. Using a formula of:

$$Quality_{\text{of this stage}} = Quality_{\text{of previous stage}} * \%\ DefectLevel_{\text{of this stage}}$$

a simple analysis of the problem as shown in Table 3, results in some scary conclusions. Even scarier than the mathematical analysis is the consistency of the conclusion to many real life situations.

Scenario	Stage->	Requirements	Functional	Design	Code
All Stages at 90% Correct	% Work Done Correctly	90%	90%	90%	90%
	Cumulative Defect %	90%	81%	73%	66%
All Stages at 85% Correct	% Work Done Correctly	85%	85%	85%	85%
	Cumulative Defect %	85%	72%	61%	52%

Table 3. Cumulative Effects on a Multi-stage Process, at an 85% quality level almost half of the code delivered will be defective

3.2 Characteristics of Good Requirements

The primary goal of good requirements is an effective communications mechanism. Customers will have their expectations met and the entire development team can discuss all aspects of the product in terms that relate directly to the customer goals. In particular the work done in the system-testing phase can be directly tied to the needs of the customer. An additional long-term value of the requirements spec is the ability to quickly understand the affects of changes made to the system as it evolves.

This paper is focused on some of the issues related to creating a robust description of a system's behavior. For most modern systems there is an infinite number of potential scenarios that can be defined for a system. Clearly an engineer/analyst cannot define this many scenarios, a common substitute is a long text description of the system combined with some use cases. These tomes are often too large for effectively communicating the desired behavior of a system and lack an effective means of helping the engineers analyze the specification. The use cases do aid the engineer by providing a clear

description of a use scenario. A problem with this approach is the lack of a standard mechanism to generalize the example used in the use scenario to the robust model needed to communicate the system to engineers.

4. What Use Cases Lack

Some things Use Cases lack in describing the requirements are described below. An example of a good use case diagram and the problems described is shown in Figure 3.

1. ***Sequence and flow*** of operations. Although Use Case diagrams can illustrate some of the relationships between individual Use Cases, they do not convey sequence and flow of related operational usage. In a large model detailing to this level of behavior becomes increasingly difficult. Besides requirements traceability this is one of the biggest benefits of behavioral modeling. It gives the engineer an opportunity to describe requirements from a usage standpoint.

2. ***Frequency and arrival rate*** information of individual Use Cases. Such information can be used for performance engineering of a system and help define load test scenarios. This type of information is easily portrayed in a state transition model, not so easily captured with Use Cases. Once sequence and flow is effectively modeled annotating frequency information can enhance the graphical representation. This information can be valuable for performance and load testing

3. Use cases often describe ***only best case*** (successful completion of operation) and limited exception information. Visualizing and modeling many exception conditions can be a large task. This is more easily undertaken using state machine techniques.

4. ***How the system(s) are used***, or could be used. An improved process can be realized if the actual requirements are separated from the discrete descriptions of individual use cases. Following paths through a behavioral model has proven to be a very effective approach to understanding how requirements might be used with in a "System" framework. This modeling gives the engineers a chance to further nail-down requirements and how they behave.

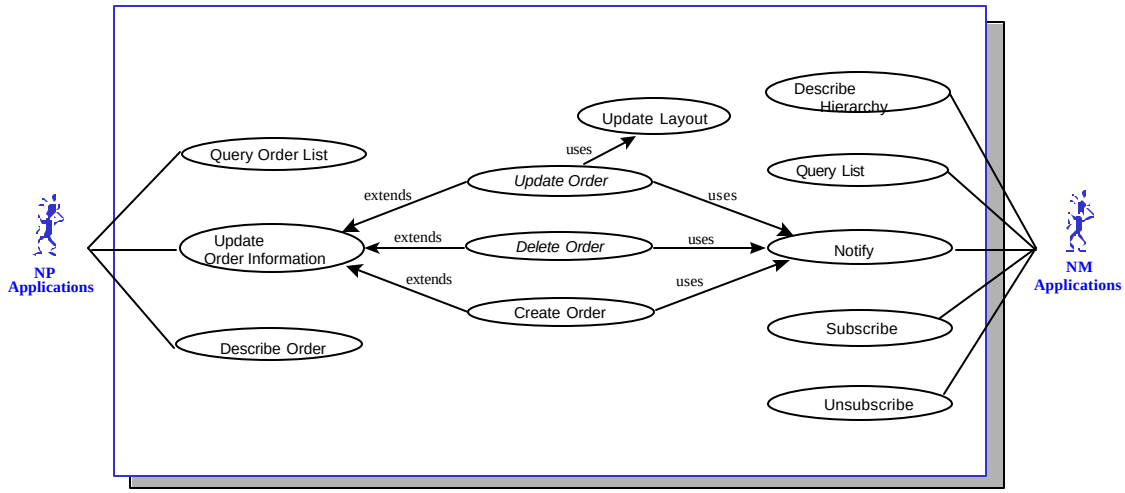


Figure 3 Simplified - High Level Use Case Model **[Ehrlich 1998[6]].** *Where is -- 1. Sequence and Flow, 2. Frequency and Arrival Rate, 3. Only Best Case, 4. How the system(s) are used?*

It is not always evident how various Use Cases interact within a developed solution. Systems engineers can use models to further refine the relationships between Use Cases at a ***high*** level. This is useful in confirming the Use Case framework used to describe the feature functionality of the solution. [Meyer May 1998[7]]

5. Fitting Behavioral Modeling into the Development Process

This discussion will focus on the "State machine models" box as the piece inserted into the development process (see Figure 4). The premise here is modeling work is appropriate for both system engineers and test engineers. There are also implications the system engineers models could be useful for the analysis, design and implementation teams. As the application progresses to integration and system test, executable test scripts will have been produced to exercise the applications behavior.[Meyer Jan 1999[8]]

5.1 System Engineers Model

The focus of the system engineers modeling work is to further nail-down requirements by describing them with state/transition methodology. This description should provide organization and sequence information and requirements traceability.

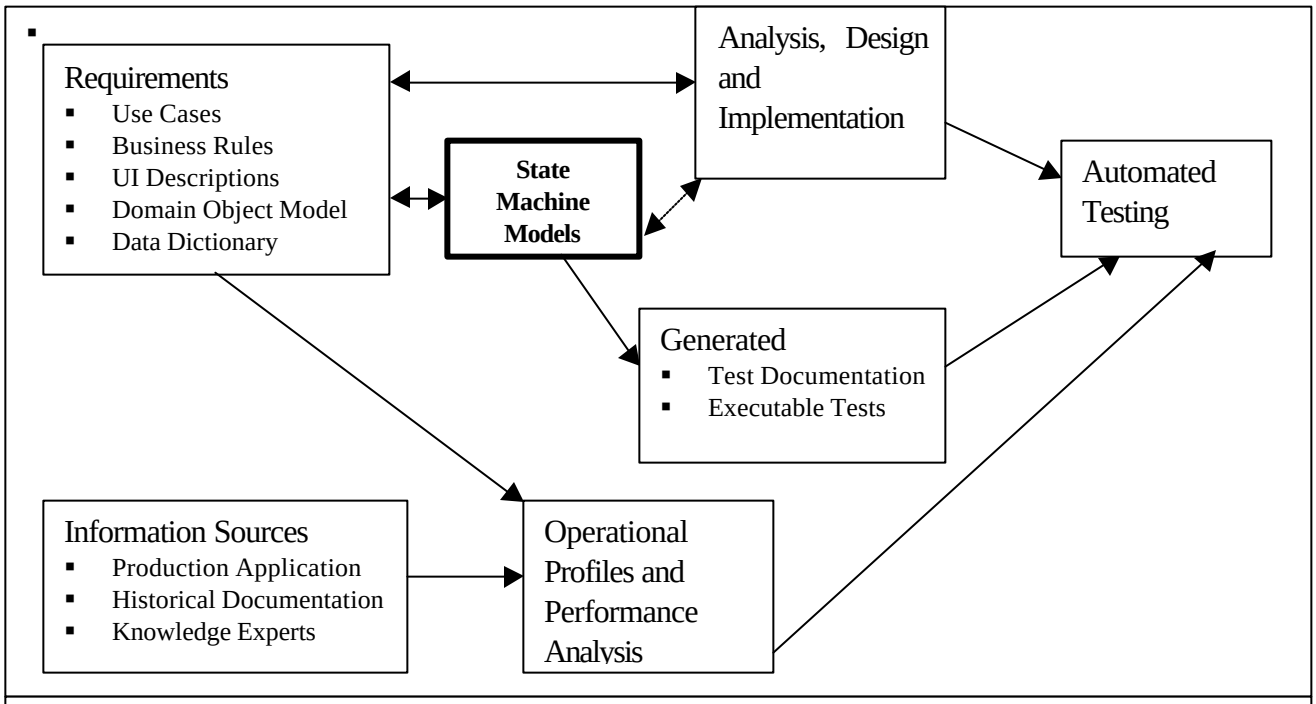


Figure 4 System Development Process, a step has been added to develop state models

5.2 Test Engineers Model

Test engineers can take the system engineers state-machine behavioral models and extend them. Calculating paths through these extended models exposes a great amount of complex behavior that needs testing. Taken a step further these paths become tests.

6. Use of Models

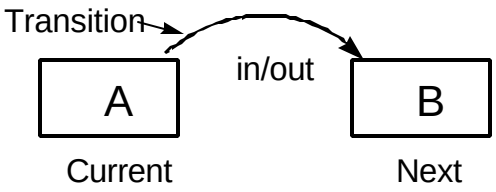
The use of models to define behavior is not new, it has been used in both development and test processes.[9,10] The recent popularity of Object Oriented Analysis and Design techniques have increased the deployment of model based techniques across the software development community.

We have developed a technique where a behavioral model of a system can be used to replace the sets of use cases used to describe a system's requirements. The model provides a means of generating use scenarios when needed. Each path through the model is equivalent to a unique use scenario. By exploring different paths, including potential loops, error conditions etc. a more robust understanding of the system can be developed. Furthermore, as the system is changed the graphical view of the model makes communicating the impact of a change much easier.

6.1 Building a Model

Modeling is not new, engineers always build and use models. The 'model' may only exist for a short time and live on a napkin or remain in the mind of the engineer; it is not always preserved in a reusable form. This model of behavior is analyzed when one creates requirements, writes a use case, designs code, or develops a test plan; an engineer must understand the basic operations and actions of the system. The means used to implement the behavior is not required until design and/or implementation begins. Modeling at the behavioral level is straightforward and can contain information from specifications, use case diagrams, sequence diagrams and flowcharts.

This approach is based on the concept of a state machine (see Figure 5) where the transactions, represented by arrows in the diagram, correspond to the actions (transitions in the state machine), while icons in a variety of shapes represent the states. State machines are an established modeling approach and have been extended[11] to provide a more powerful means of describing complex systems. Actions that can occur during the use of the system are defined by adding arrows and or states to the basic diagram. The actions that "could" occur also imply that that there may be more than one possible action at a specific point in the process. Most modeling techniques support the idea that there are multiple potential "next" actions. Some of these actions can only occur if certain conditions exist in the system, this information can be added to the basic model by associating a rule or condition with each action. In our approach these are included as predicate expressions on the transitions. An effective modeling technique allows an engineer [analyst or other specifier] to unambiguously define these alternative sequences and any rules associated with them. Another modeling technique we utilize is hierarchical models, where a state can be replaced by a 'call' to another model that defines the behavior within the state, this approach is also referred to as 'super-states'. Hierarchical models allow complex behavior to be decomposed into simpler lower level models.



- Transitions define actions that move the system from the current state to a new state
- Transitions can be based on specific stimuli and/or previous actions
- Text descriptions of user actions can also be associated with transitions
- Use cases are built by concatenating the descriptions from a sequence of transitions through a diagram

Figure 5. A Finite State Machine is Composed of States and Transitions

Developing a specification in the form of a model is a very effective means of:

1) discovering defects in the system (many are made visible by the modeling effort alone),
2) rapidly defining the basis for use scenarios of the system, and
3) preserving this investment for future releases or other similar systems.

Furthermore, the process of developing a model is best done in a measured series of small incremental steps. This incremental building approach allows the core functionality to be defined, evaluated and understood before all of the secondary features are added. This also allows a larger team to be applied to the problem. They can divide up the more detailed specifications (typically in lower level models), and integrate the work into the larger model.

Once a model has been developed, even if it is a partial model, use scenarios can be developed by finding paths. A path is a sequence of events or actions that traverse through the model defining an actual use scenario of the system. Each element in a path, a transition or state, can have some text associated with it. Concatenating all of the text found on the path elements will provide us with a textual description of the entire use scenario. This process can be repeated for another path, which defines another use scenario, and validates another sequence of actions. Many methods can be used to select paths, each with its own distinct objectives and advantages. Operational profiles, reliability/criticality data, and coverage all provide different tradeoffs to the type of scenarios and the resulting coverage. A primary advantage of this type of structured definition of the behavior is that automated tools can be

used to help in the analysis. This approach also allows an engineer or analyst to see potential secondary paths that may exist. The benefits of having this complete view of all of the potential behaviors at any point in the flow will allow a more complete analysis as well as make updates and changes easier to insert.

6.2 Overlaying Requirements on the Model

At this point we can define a behavioral requirement in terms of model objects. If a system must provide a specific behavior as one of its requirements it can be expressed in terms of the objects in the model. For example in Figure 6 we define a voicemail system. If a requirement specification included "Requirement *timeout_transfer* : On a busy line, the system shall transfer a caller to the attendant after the specified timeout period" we could define this by linking the requirements to the elements of the model [block arrows in Figure 6] with an expression like:

"*timeout_transfer* = **Busy** && (**Timeout** -> **Attendant**)"

The -> operator is defined as "is immediately followed in a sequence by", the operands are the objects in the model diagram [**Busy** and **Timeout** are events, **Attendant** is a state (actually a superstate)]. This expression now provides us with a versatile mechanism for understanding when any path is verifying the requirement. Any path where *timeout_transfer* evaluates to true is one that verifies the requirement. This model and expression based approach now allows other portions of the model to change without affecting this requirement, and as new paths are generated they can all be evaluated to

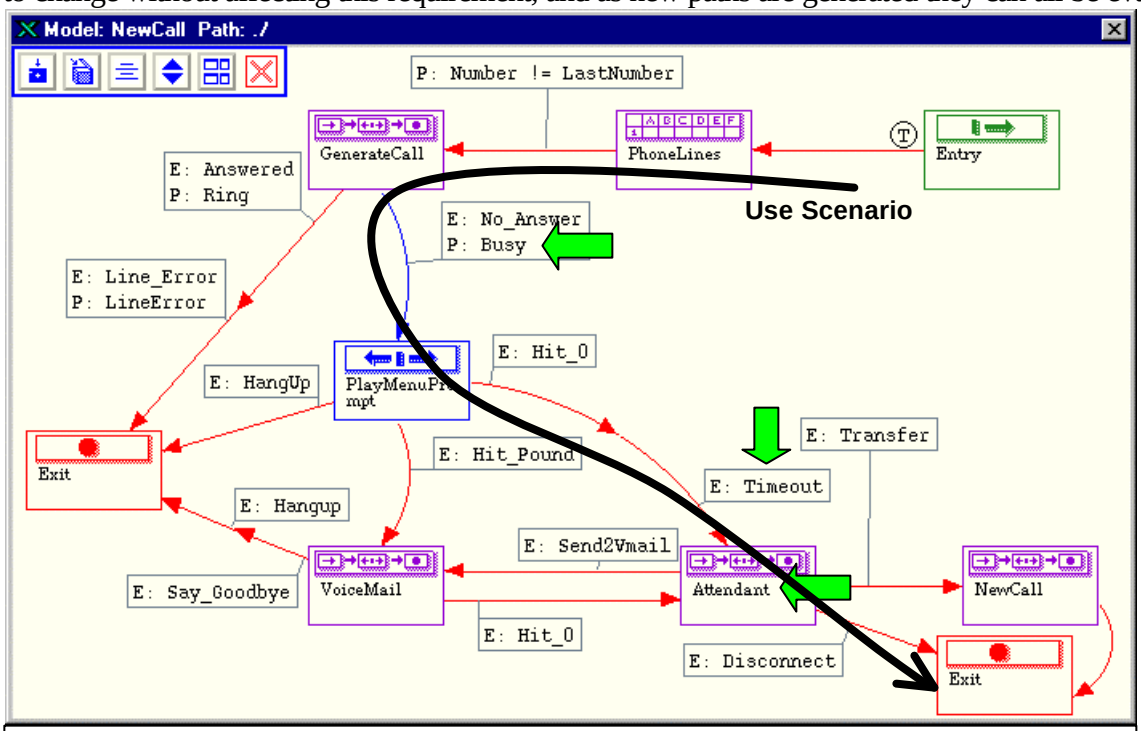


Figure 6 Voicemail Model Example. A model can show a use case in the context of the overall system functionality. This example illustrates a use scenario where after the greeting is played on a busy line a caller will be transferred to the attendant after a timeout period.

see which of them verify a requirement.

The graphical model can also be processed by an automated test generation tool to produce not only test plans and scripts but a detailed document describing how each requirement was covered in the resulting set of tests. As the tests are created and later executed a team can track the project's completeness and the product's consistency with the requirements. A sample of a report from an automated test process with the requirements tracking integrated into it is shown in Figure 7.

Another benefit occurs when changes are introduced, if a new feature is added, an incremental edit is made to the model and then paths can be regenerated to confirm that the old requirements are still covered. If the changes originate with the requirements, they can be added as new requirement expressions and then the model can be analyzed to see if the requirement is already covered by the existing functionality or new behaviors must be added.
A second type of analysis, similar to the requirement can also be performed on a model. If there are requirements that the system not allow certain behaviors to occur, they too can be expressed as expressions and verified. For example, in a system there may be relationships that should not be violated, sometimes referred to as invariants, these can also be verified using an expression. The expression, similar to the ones used for specifying requirements, is evaluated during the path generation process; if at any time it becomes false, we have uncovered a design flaw in the system.

7. Conclusions

An approach where a system's behavioral requirements are used to incrementally develop a behavioral

		Requirement Numbers						
Total Test per Req.		2	3	2	3	3	0 1	
Total Req per Test	Test Number	1.0	1.0.1	1.0.2	1.1	1.1.1	1.2	1.2.1
3	1		X			X		X
1	2	X						
2	3		X		X			
1	4				X			
0	5							
2	6		X		X			
3	7	X		X		X		
0	8							
1	9					X		
	10			X				

Figure 7 Tests vs. Requirements Report – This allows the engineer to determine which tests are redundant as well as which requirements are not yet covered. In addition, tests that are not adding new requirements can be dropped to optimize test execution time.

model can provide a team with significant advantages. It can be used to thoroughly describe application behavior from specifications or models and automate the creation of use cases as well as test programs. A path generation tool can be used to extract ***directly executable*** tests.[12]

Benefits include:

- Provides early detection of incomplete or inconsistent requirements.
- Significantly improves test coverage of requirements.
- Provides an easy method of documenting covered requirements by tests
- Manual testing can be error prone and not always reproducible
- Given a change to the requirements, it is easier to adapt model and then automatically re-generate the test scripts than to manually change all the test scripts.
- Facilitates test suite management as the model can be base-lined for a particular release [Meyer Jan 99]

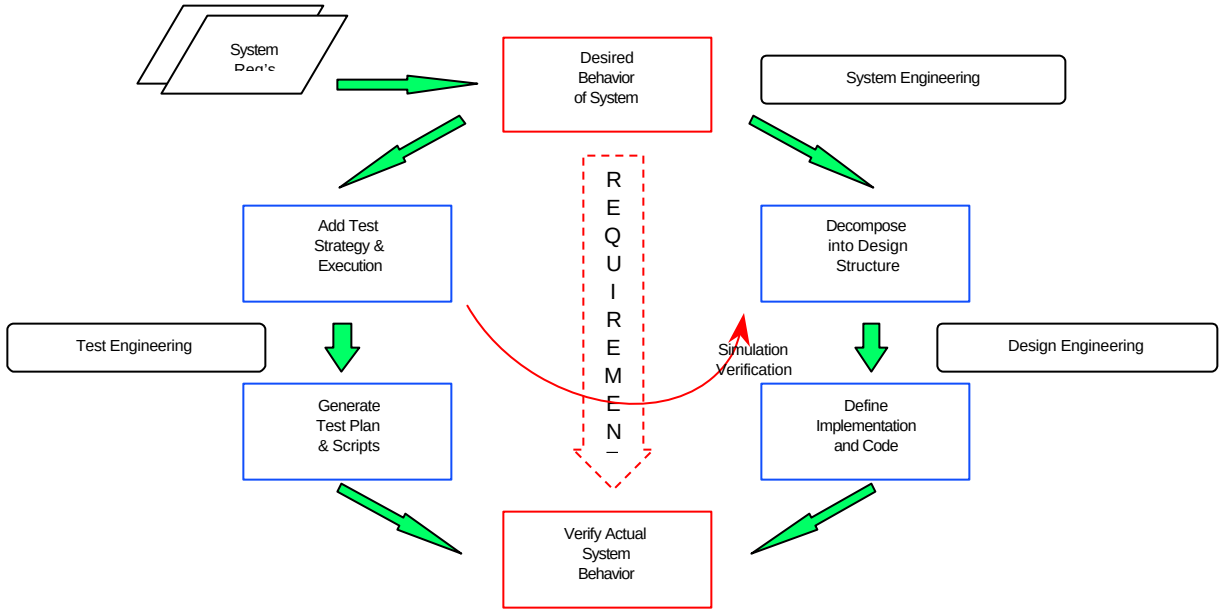


Figure 8 The System Behavior is the reference for all later stages. Requirements defined based on the system level model will carry forward into the test.

Furthermore if the requirements of the system are integrated as expressions, the system can also automatically track the progress and completeness of the team. This system level behavioral model can serve as a reference for the systems engineering as well as the development and test phases of product development, see Figure 8.

Benefits of this approach include easy analysis of behavior, rapid response to changes in specifications of requirements, tests generation correlated to requirements as well as automated generation of requirements based tests. This system provides an organization with an efficient, reusable mechanism to both maintain a suite of tests and respond to feature or requirements changes during the product development cycle.

References

[1] Dorfman M, and Thayer R. Standards, Guidelines and Examples of System and Software Requirements Engineering. Washington D.C. IEEE 1990

[2] A. Davis, Software Requirements Objects Functions and States, New Jersey, Prentice Hall, 1993

[3] D. Gause & G. Weinberg, Exploring Requirements – Quality Before Design, New York, Dorset House Publishing, 1989

[4] B. Boehm, Software Engineering Economics, Prentice Hall, New Jersey, 1981

5 J. Taft, "Implementing the 'V-Model' Quality Framework, Software Testing Analysis & Review West, 1997

6 W. Ehrlich, "Facility Component System", Use Case Diagram, 1998

[7] S. Meyer, " Foundation Architecture Evaluation Results - Release 7.0 , Test Case Generation Tools ", AT&T Internal Memorandum, 1997

[8] S. Meyer, "TestMaster Pilot Update", AT&T Internal Memorandum, 1999

[9] Beizer, B., *Black Box Testing*, New York, John Wiley & Sons, 1995. ISBN 0-471-12094-4

[10] L. Apfelbaum. Doyle, J., "Model Based Testing", Proceedings of the Software Quality Week 1997 Conference, 1997.

[11] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming 8, 1987

[12] J. Clarke, "Automated Test Generation from a Behavioral Model", Proceedings of the Software Quality Week 1998 Conference, 1998