SQ^{2}E: An Approach to Requirements Validation with Scenario Question

Daniel Aceituna[1], Hyunsook Do[1]
[1] Computer Science Department
North Dakota State University, Fargo, ND, USA
{daniel.aceituna, hyunsook.do}@ndsu.edu

Seok-Won Lee[2]
[2]Department of Computer Science and Engineering
University of Nebraska - Lincoln,
Lincoln, NE, USA
slee@cse.unl.edu

***Abstract*— Adequate requirements validation could prevent errors from propagating into later development phase, and eventually improve the quality of software systems. However, often validating textual requirements is difficult and error-prone. We develop a feedback-based requirements validation methodology that provides an interactive and systematic way to validate a requirements model. Our approach is based on the notion of querying a model, which is built from a requirements specification, with scenario questions, in order to determine whether the model's behavior satisfies the given requirements. To investigate feasibility of our approach, we implemented a Scenario Question Query Engine (SQ^{2}E), which uses scenario questions to query a model, and performed a preliminary case study using a real-world application. The results show that the approach we proposed was effective in detecting both expected and unexpected behaviors in a model. We believe that our approach could improve the quality of requirements and ultimately the quality of software systems.**

Keywords- Requirements validation, feedback-based validation, model-based validation, scenario questions

I. INTRODUCTION

It is well known that catching an error in the later stage of software development is usually more time consuming and costly than if the error is caught in the early stage [1], [2], [3], [4]. For instance, Boehm and Basili [1] indicate that finding and correcting defects after product release is often 100 times more expensive than correcting them during the requirements and modeling phase. Also, one survey performed by Hofmann and Lehner [5] found that successful IT projects have spent about 28% of the effort on the requirements phase.

One important and necessary activity in requirements engineering process that can prevent errors from propagating into a later development phase is requirements validation. Requirements validation is a difficult task because by nature a requirements specification can be interpreted differently by various stakeholders and under different contexts. To date, many researchers and practitioners have proposed and developed numerous requirements validation techniques to improve requirements validation processes and their effectiveness, such as reviews, inspections, prototyping, and formal methods [6], [7], [15], [19], [20], [21]. Each of these techniques has its inherent strengths and weaknesses over others, so choices of requirements validation techniques should be made considering the types of systems being built and software development processes practiced by organizations.

In this research, we primarily focus on a model-based requirements validation methodology, which provides a systematic way to validate functional requirements and is especially suitable for the logical analysis of behaviors of hardware controller software. In particular, we utilize scenarios, which capture typical system behaviors. The notion of scenarios has been frequently used in the requirements engineering process because it is conceptually easy to use for both engineers and stakeholders who do not have engineering background, and thus it can help extract domain knowledge of a software system from various stakeholders. Initially, scenarios vastly have been used in informal ways, but recently researchers have tried to adapt scenarios to formal approaches, such as requirements validation using models constructed based on scenarios, and specification language-based requirements analysis using scenarios [15], [19], [20], [26].

In this paper, we propose an approach to feedback-based requirements validation, which facilitates a means for requirements engineers to interact with a requirements model and validate its behavior through a set of queries. Our approach is based on the notion of having a question/answer session with a requirements model in order to determine whether the model's behavior satisfies the given requirements. In this approach, given a model built by engineers, the model is queried with scenario questions that are derived from requirements. The answer to a given question is immediately generated by traversing a behavioral model using a scenario question. This immediate feedback mechanism allows engineers to address potential problems with a requirements model and to formulate unconsidered scenario questions dynamically in response to feedback. Getting an answer to a given requirement helps determine whether that requirement is satisfied or not.

To support our approach, we implemented a Scenario Question Query Engine (SQ^{2}E), which uses scenario questions to query a model. In our preliminary study, we considered State Transition Diagrams (STD), which have been widely used in modeling the behavior of software systems [9], [11]. The SQ^{2}E then displays all the path

traversals in a model that both satisfy and fail to satisfy a query. These traversals are displayed as path strings, which are individual simple paths through a STD model. A given individual simple path is one that traverses a sequence of nodes without visiting a given node more than once.

The generated paths are then analyzed in an effort: (1) to validate that a queried scenario occurs as defined by a set of requirements; (2) to discover any unexpected behavior that exposes an error; (3) to suggest an improvement in a model; (4) and ultimately to provide feedback on a requirements specification.

To perform path traversals, we used a reasoning engine, consisting of a set of Prolog rules with parameters serving as inference constraints that specify the node(s) and/or edge(s) that each path must cross during a path traversal. We used Prolog because it facilitates keeping the model representation separate from the reasoning engine [12]. The goal is to be able to readily build any model into a dedicated instance of the SQ^{2}E. Thus, we represented a state model as Prolog facts, and used a Prolog interpreter written in C#. The use of Prolog also makes the reasoning engine expandable to accommodate any additional constraints that may increase the range of scenario questions.

To assess our approach, we performed a preliminary case study using a real-world application, a hardware controller. The results of our study show that the SQ^{2}E can provide an effective way to validate requirements models. The approach we proposed was useful in detecting expected and unexpected behavior in a requirements model by proactively utilizing engineers' domain knowledge through scenario questions they produce. We also observed that SQ querying could not only catch errors in a model, but also prompt questions that lead to a model's improvement.

The rest of the paper is organized as follows. Section II presents the framework of the SQ^{2}E, and Section III presents a preliminary case study, which shows how the SQ^{2}E is used to validate a real-world hardware controller. Section IV discusses related work, and Section V presents conclusions, and discusses possible future work.

II. THE FRAMEWORK OF THE SQ^{2}E

In this research, our goal is to develop a validation technique that analyzes a requirements model by asking a question about the model's behavior. During the SQ querying process, a Scenario Question (SQ) is first expressed in plain English, and then translated (by the user) into a set of inference constraints, which the tool uses to query a model. The tool then answers a question by generating and displaying state transition traversal path strings (in short, path strings), which represent paths through a model describing its behavior. To investigate feasibility of our approach, we have implemented a tool that uses a SQ^{2}E written in Prolog and C#, considering State Transition Diagrams (STD). The following subsections provide further details on the SQ^{2}E and its functionalities.

A. SQ Query Engine (SQ^{2}E)

The SQ^{2}E consists of the functionalities expressed in the dotted boundary in Fig. 1. The SQ^{2}E accepts scenario questions about the model's behavior in the predefined form of inference constraints (Fig. 1), and then generates answers to the questions in the form of path strings derived from a STD model, which require human interpretation of the results.

The key component of the SQ^{2}E is *automated reasoning* (the center box in Fig. 1), which provides a central idea of this research. This component is further explained using Figs. 2 and 3, which correspond to the two main functions of the automated reasoning component: (1) generating inference constraints from scenario questions; (2) generating path strings from inference constraints. Figs. 2 and 3 visually illustrate these functions, respectively.

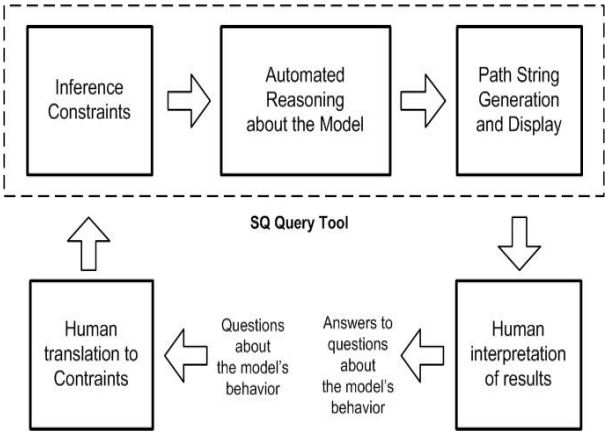

Figure 1. Block diagram of the SQ querying process.

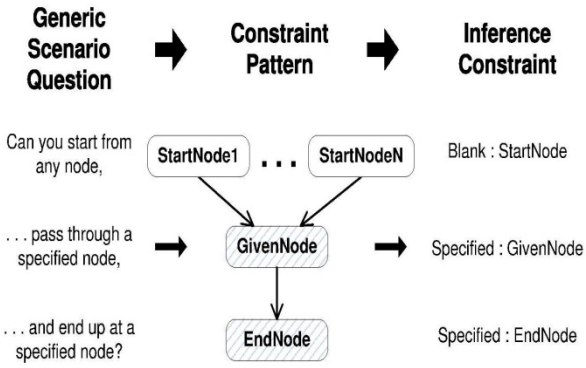

Figure 2. One of the constraint patterns used to translate a scenario question.

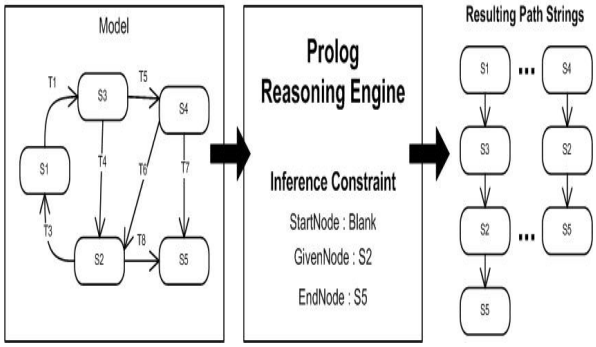

Figure 3. Path strings resulting from the inference constraints listed in the center box.

To help understand the rest of this section, we define the major components appeared in Fig. 2.

- A **scenario question (SQ)** is a naturally phrased question that queries a model about an expected behavior. The question should ideally address an expected behavior based on a given requirement. Thus, each scenario question should map to a requirement.
- A **constraint pattern** is an abstract pattern that helps to map a SQ to an inference constraint. In this research, as an initial set of constraint patterns, we identified eight patterns for nodes, and eight patterns for edges. These are further explained later in this section.
- An **inference constraint** is a set of parameters used by the Prolog inference engine. They specify the node(s) and/or edge(s) that each path must cross during a path traversal. Typically, a scenario question is initially expressed in natural language, and then translated into a set of inference constraints. Currently our tool does not automate natural language translation, thus, for the time being the translation is performed manually. Eventually, manual translation can be extended easily to automatic translation by using the constraint patterns as a guide.

After generating inference constraints, the Prolog reasoning engine (Fig. 3) generates the path strings that provide the answer to the given question. The Prolog reasoning engine converts the model shown in Fig. 3 into Prolog facts, which are used with the Prolog rules that make up the inference engine. The primary rule that is used to traverse the model and generate the path strings is shown in two boxes in the next column.

Note that the ***findPaths*** predicate uses the inference constraints: ***StartNode***, ***EndNode***, ***GivenNode***, ***StartEdge***, ***EndEdge***, and ***GivenEdge***. We will elaborate on these constraints later in the paper. The ***findPaths*** rules uses the ***path_aux*** rule (below) which performs the actual traversal. The ***arc(NID1, IDN, NID2)*** predicate represents a given present state (***NID1***) to next state (***NID2***) via a transition (***IDN***).

Prolog will expand a STD into an infinite tree, whose branches (paths) are then traversed recursively. During the traversal of a given path, the nodes and edges within that path are compiled into a list data structure, which is then used to display the resulting path string. A path string is displayed for every path traversed through the tree.

```
% Reasoning engine's primary rule
% (note the inference constraints)

findPaths(StartNode, EndNode, GivenNode, StartEdge,
EndEdge, GivenEdge, Depth, N, Nodes, Edges) :-
path_aux(StartNode, EndNode, [EndNode], Nodes, Edges1,
Depth),
reverse(Edges1, Edges),
occur (GivenEdge, Edges, N),

%Accept paths where StartEdge is first Edge
first (StartEdge, Edges),
%Accept paths where EndEdge is last Edge
last_item (Edges, EndEdge),

(occurs(GivenNode, Nodes, 1); ((last_item(Nodes, Last),
first(Ft, Nodes), Ft=Last), (GivenNode=Ft;
GivenNode=Last))), (allownoccurs(Nodes, Depth);
(last_item (Nodes, Last), first (Ft, Nodes), Ft=Last)).
```

```
% This Rule finds all the paths for a given Nodes
% at a given length and is used by
% the Test Path finding rules.

path_aux(NID1, NID2, L, [NID1|L], [IDN], Depth) :-
arc(NID1, IDN, NID2).

path_aux(NID1, NID2, L, P, [IDN|ET], Depth) :-
arc(Z, IDN, NID2), allownoccurs([Z|L], Depth),
path_aux(NID1, Z, [Z|L], P, ET, Depth).
```

B. The SQ Querying Process

Having explained the main concept and major components of the SQ^2E, we now explain how these components are used in querying a STD model and obtaining the answers to the question. First, we explain a translation mechanism for scenario questions, and then show the correlation between how questions are phrased and the use of Prolog rule constraints.

We defined six inference constraints that the inference engine uses in creating the resulting path strings (***StartNode***, ***GivenNode***, ***EndNode***, ***StartEdge***, ***GivenEdge***, and ***EndEdge***):

- ***StartNode(Edge)*** specifies the starting ending node (edge) for traversal

- ***EndNode(Edge)*** specifies the ending node (edge) for traversal
- ***GivenNode(Edge)*** forces the path strings to traverse a specified node (edge)

To illustrate this concept, consider the model shown in Fig. 3 (the leftmost box). If we specify ***StartNode=S1***, ***GivenNode=S2***, and ***EndNode=S5***, then every path string will be constrained to having ***S1*** at the beginning of the string and ***S5*** at the end, while ***S2*** appears somewhere within the string. As demonstrated in Fig. 3, setting ***StartNode=Blank***, will produce path strings that begin with all the nodes in the model.

Each of the six-node/edge constraints can either be assigned the name of a given node/edge in the model, or it can be left blank, which results in path strings that start with any node in the diagram. Setting the constraints to ***StartNode=Blank***, ***GivenNode=S2***, and ***EndNode=S5*** results in a Y-shaped constraint pattern (on the upper left-hand corner in Fig. 4).

Fig. 4 also shows all other possible patterns that can be derived. We have eight patterns in total because there are three possible nodes to define (***StartNode***, ***GivenNode***, and ***EndNode***), and each of the three nodes can take on one of two conditions (a specified node name, or blank). We denote the patterns with the letter that they resemble.

As shown in Fig. 4, we have eight patterns that are labeled, from left to right starting from the upper left: **Y**, **X**, **V**, **H**, **I**, **J**, **O**, and **A**. There are eight generic questions that map to the eight constraints patterns, and since each pattern has three parts, all scenario questions are phrased considering a three part-structure defined in our SQ^2E.

While a three-part structure is preferred, the three parts do not necessarily need to be in any specific order. This flexibility in order can accommodate relaxations in phrasing scenario questions. Further, we also need to determine from phrasing of the question whether an inference constraint may have a specified node name, or may be left blank. This can be determined easily by examining the way questions have been phrased. Table I shows how the eight constraint patterns would map to typical scenario questions for the model in Fig. 3. The letter designating the constraint pattern precedes each question and each node constraint is preceded with the word "*blank*", meaning any node, or "*specified*" meaning a specific node name was given.

Restricting scenario questions to a three-part structure may appear to decrease the degree of freedom in phrasing the questions. However, we believe that the initial set of eight constraint patterns we defined covers comprehensive and typical scenario questions as shown in Table I, which allow us to evaluate feasibility of our approach. Furthermore, these patterns can be extended to express more scenario questions by combining the three edge (transitions) constraints and the three node constraints.

C. Highlights

In addition to constraints, our tool allows for the highlighting of three nodes and three edges to help users with the results analysis process. That is, a user can highlight specified nodes and edges in the resulting path strings so that the user can visually verify if the selected node or edge has been traversed in a given path string. For example, referring to the model in Fig. 3, if the user assigns S2 as a highlighted node, then S2 will appear highlighted in every path string that it appears in. This allows the user to quickly spot a string path that does not contain S2.

The combination of constraints and highlights allows for questions that seek positive or negative validation. For example, a question such as "Does S2 always occur in paths starting with S1?" (implying positive verification) is best verified with constraints, whereas "Does S2 occur without starting with S1?" (implying negative verification) is verified using highlights.

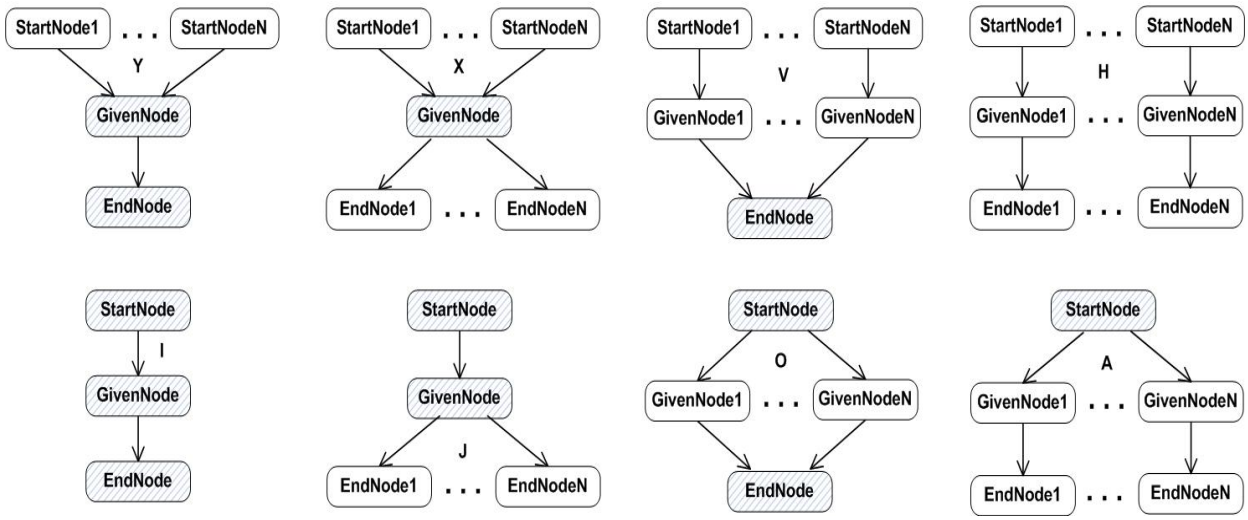

Figure 4. The eight node constraint pattern.

TABLE I. The eight constraint patterns that map to typical scenario questions.

Pattern	Query	Start Node	Given Node	End Node
Y	Starting from any state, will you always pass through ***S3***, on the way to ***S5***?	blank	S3	S5
X	Starting from any state, will you always pass through ***S3***, on the way to any other state?	blank	S3	blank
V	Stating from any state, how many other states will be visited, on the way to ***S5***?	blank	blank	S5
H	What are all the possible paths in this model?	blank	blank	blank
I	How many ways can you begin from ***S1***, pass through ***S3***, and end up in ***S5***?	S1	S3	S5
J	How many different states can you end up at, when beginning from ***S1***, and always passing through ***S3***?	S1	S3	blank
O	How many different scenarios can occur, when going from ***S1***, to ***S5***?	S1	blank	S3
A	When you start charging, how many scenarios can occur starting from ***S1***?	S1	blank	blank

The combination of constraints and highlights also allows for a larger expressive power in formulating the desired scenario question. We do not claim that every conceivable question can be expressed using the stated number of constraints and highlights, but we try to provide a user-friendly model-based requirements validation methodology using the SQ querying concept.

Fig. 5 shows a simplified architecture of the SQ^{2}E and the sections that handle the inference constraints and the highlights. The constraints are handled by the Prolog inference engine, which performs the actual traversals, whereas the highlighting of the node/edge is implemented during post processing of the path strings generated by Prolog. We highlight during post processing because the Prolog rules are unable to visually highlight a node or edge. As Fig. 5 shows, the paths that are generated as the results of traversal constraints are pipelined into the post processing where the highlighted nodes/edges are tagged with different colors.

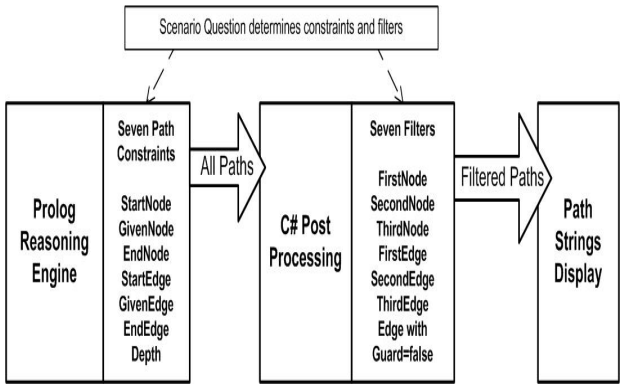

Figure 5. The constraints and highlights used with the SQ^{2}E.

The (color-coded) node highlights can also reveal path strings that have two to three of the nodes appearing in a given order. The same ability applies to the three color-coded edge markers. In other words, the tool can use highlights to indicate an expected sequence of nodes or edges. For example, if we indicate that edge "D" should precede edge "G", which in turn should precede edge "A", then path strings containing edges in the order D, G, A are flagged and the tool will notify the user which path strings have the three edges in that expected order.

III. PRELIMINARY CASE STUDY

To investigate whether our approach described in Section II can be effective in validating models that have been built during the requirements analysis stage, we performed a preliminary case study using a real world application.

The artifact used in this case study is a STD model for an end-of-line product tester used in a production plant. The production tester is a self-standing, automated hardware tester used to test one of the company's Radio Frequency (RF) products before it is shipped to the customer.

The production tester requires the following setup procedure: a production worker will open the tester's RF chamber, place the PUT inside, mount it on its test fixture, close the door, and start the test of individual devices by clicking a button on the test GUI. This setup procedure's responsibility is to insure that the unit has been properly placed and pneumatically clamp the unit (no movement during testing is critical), assure the door is closed, and allow the test to start when the operator clicks on the proper GUI control. Fig. 6 shows the requirements model for the setup tester we just described. The model in Fig. 6 was built based on the 15 requirements defined for the setup procedure by a requirements engineer.

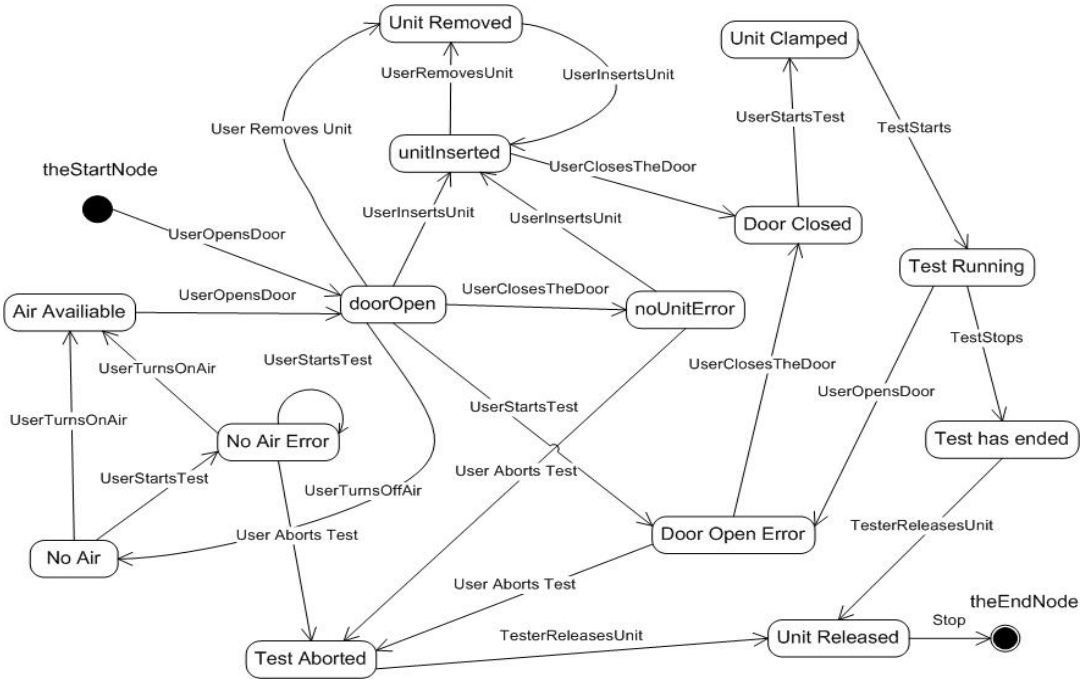

Figure 6. STD for requirements validation.

The goal in applying SQ queries to the model is two-fold: (1) validate how closely the model represents the requirements; (2) more important, try to reveal ambiguities and possible improvements to the model.

These revelations could be in the form of unexpected behavior displayed by the model, which could subsequently translate into buggy software. Based on product analysts' experiences, we started with 35 Scenario Questions derived from 15 requirements. Due to space limitations, we will elaborate on only 3 of the 35. The rest of them are described in [16].

When listing a scenario question, the notation *SQn.m* is used to achieve traceability to the requirement that inspired the question. The subscript *n* indicates a requirement number; *m* indicates a scenario question number. For example, SQ2.3 is the third scenario question for requirement 2. We derived scenario questions considering the two aspects we mentioned previously (validating the model and revealing ambiguities).

Due to space limitations, we show the three scenario questions, their corresponding inference constraints, and a brief explanation of the rationale used in translating the SQ to its inference constraints, as shown in Table II.

Based on the inference constraints that we identified, we generated the STD traversal paths. Out of the 35 questions, six questions revealed problems with the model. Table III lists our test results (from interpreting the traversal path strings) from querying the model with the scenario questions.

As Table III indicates, 28 scenario questions correctly validated the model's expected behavior, as it pertained to those questions. SQ3.2 revealed a definite error, and SQ2.3 exposed the lack of a contingency (These two cases are described in detail below). SQ15.1 revealed a possible improvement: the fact the model as it stands shows that it is possible to release a unit, when no unit is inserted. This is not a critical problem, but reveals that perhaps the setup software should be more intelligent.

SQ1.2, SQ2.2, SQ14.4, and SQ15.1 prompted follow up questions (not shown here), when their answers were ambiguous. For example, SQ2.2 asked, "Can a test run occur between a ***noAir*** state and an ***airAvailable*** state?" The answer to this question was "no." To verify that the answer is no, we asked the follow up question "Does ***airAvailiable*** precede ***testRunning*** every time?" This time we obtained ten path strings, with seven paths showing that ***airAvailable*** precedes ***testRunning*** as desired, whereas three strings show that it is possible to go from ***airAvailable*** to ***theEndNode*** without traversing ***testRunning***. Further examination of the three strings shows that the ***testAborted*** state lies between ***airAvailable*** and ***theEndNode***, which brings up a further question of whether it is possible to abort a test that is not running; thus possibly catching an unintentional functionality.

This example points to the fact that SQ querying can raise subsequent questions, whenever the answer to an initial question is not totally convincing and somewhat ambiguous.

TABLE II. Requirements, Scenario Questions, and Inference Constraints.

Requirement	Scenario Question	Inference Constraints
Requirement 1: The software shall determine if the chamber door is open or closed	**SQ1.2**: Does an open door state always precede a closed door state during a test session?	*StartNode=theStartNode, EndNode=testHasEnded, FirstNode=doorOpen, SecondNode=doorClosed*
Note: The *GivenNode* constraint remains *blank*, allowing the possibility of any *GivenNode*. Since we want to also reveal paths that violate the desired order, we will not constraint the order, but rather highlight it (using *FirstNode=doorOpen, SecondNode=doorClosed*), and allow all paths to be displayed.		
Requirement 2: The software shall determine if pneumatic pressure is present or not	**SQ2.3**: Can you lose air during testing and not abort the test session?	*StartNode=airAvailiable, EndNode=testHasEnded, FirstNode=noAir, SecondNode=testAborted*
Note: The *GivenNode* constraint is remains *blank*. We are interested in paths that cover a test session with *airAvailable*, while looking for paths that have *noAir* and *testAborted* states. Those paths that have *noAir* but no *testAborted* will satisfy the query as phrased.		
Requirement 3: The software shall determine if the *PUT* has been inserted or not inserted	**SQ3.2**: Can a test session occur with no unit inserted?	*StartNode=theStartNode, EndNode=testHasEnded, FirstNode=unitInserted*
Note: We are interested in verifying that all paths covering the entire test session have a *unitInserted* state. We use highlight (not constraint), to verify that there are no paths that do not include *unitInserted*.		

TABLE III. Results of SQ querying with the seven questions.

Result	Scenario Question
Validated behavior	28 questions
Exposed error	SQ3.2
Exposed lack of contingency	SQ2.3
Revealed need of improvement	SQ15.1
Behavior further questioned	SQ1.2, SQ2.2, SQ14.4, SQ15.1

In another example, Fig. 7 shows the results of *SQ3.2* query, which indicates that it is possible for a test session to start without a unit inserted. Note that while it does not directly violate *Requirement 3*, it does violate another requirement, which states: The software shall not begin the test, and post a ***noUnitError***, if an operator tries to start testing with no unit inserted in the unit cradle. The last (un-highlighted) path string in Fig. 7 reveals the scenario that results in the error.

As we observed in our case study, SQ querying could not only catch a model error, but also prompt questions that lead to a model's improvement.

The key to a SQ query's strength lies in involving users, and allowing them to verbally express a scenario question. The STD traversal paths strings, which are final results, are visually verified by the user, providing more information than what is normally obtained from a simple binary pass-fail result. The involvement of the human in the SQ querying process adds a human interpretation element; examining the various path strings for a given query can often bring to mind other questions, which ultimately leads to a more in-depth analysis.

Through our case study, we observed that our approach can be also applied to software testing process in addition to requirements validation; our SQ querying approach can also facilitate exploratory testing, which is an ad-hoc approach to testing software in which human testers test software systems in an exploratory manner [17].

In exploratory testing, test cases are not defined in advance; they are formulated and executed while testers walk through the product. One of the benefits of exploratory testing is that it provides a learning experience for the person doing the testing. Exploratory testing provides testers the flexibility to explore the software's behavior, and relies on feedback from the software system as human testers assess the effects of their actions on the software under test. The ability to query a model and receive feedback in the form of path strings provides a similar form of interactions that is one of key characteristics of exploratory testing. Thus, we think that SQ querying would appeal to those who gravitate toward exploratory testing.

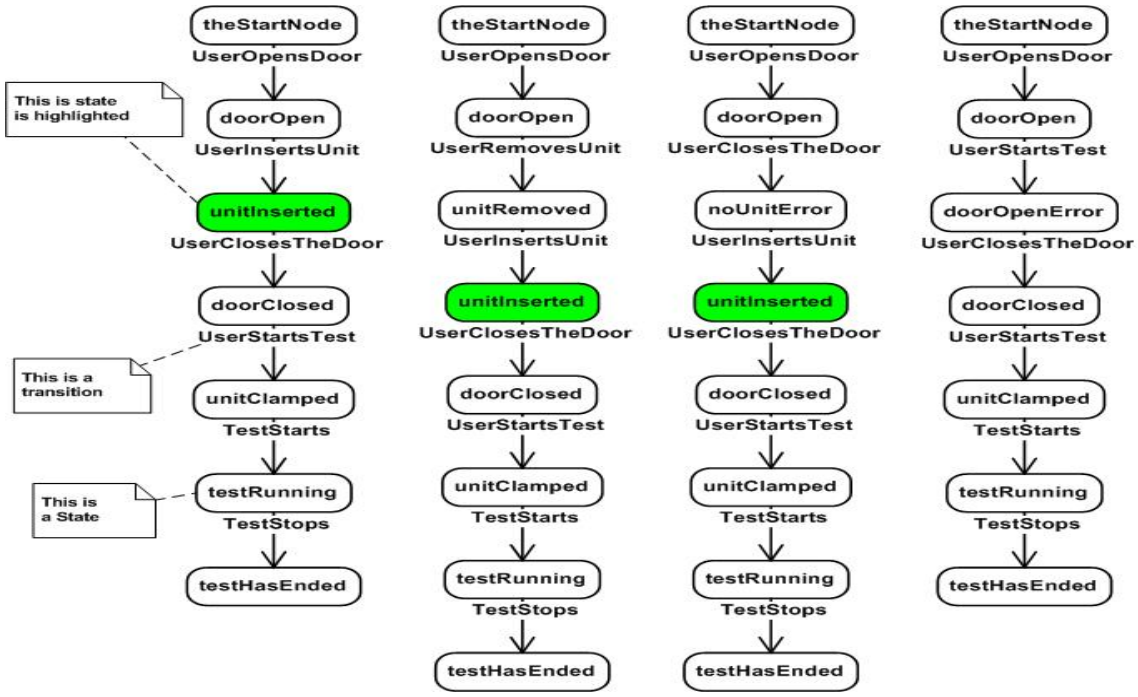

Figure 7. The results of SQ3.2 query.

IV. RELATED WORK

To date, numerous approaches to requirements analysis and validation have been proposed, but here, we limit to our discussion to tool-assisted methods that utilize scenarios, which are most closely related to our work. Typically scenarios describe examples (or counter examples) of system behaviors or use cases for system execution. Scenarios can be a scripted story, detailing a sequence of events, or they can be brief, one sentence descriptions [13], [15], [18].

Initially scenarios have vastly been used in informal ways [23], [24], but increasingly researchers have tried to adapt scenarios to formal approaches in order to improve analysis and validation processes [8], [15], [18], [20], [21], [22], [26]. Sutcliffe et al. [18], [22] propose a method that converts use cases into scenarios semi-automatically and validates scenarios using rule-based frames that detect incomplete/incorrect event patterns. An approach by Damas et al. [15], [19] addresses a problem on how to automate a modeling process using scenarios collected from end-users. Their approach captures scenarios using message sequence charts (MSCs) and then it builds a state model in the form of a labeled transition system (LST) using MSCs. Similarly, Letier et al. [20] utilize MSCs and LST to analyze requirements, but they focus on detecting implied scenarios. Others [8], [25] have further stretched the use of scenarios into helping generate test cases. Ryser and Glinz [8] propose an approach that converts scenarios into test cases in order to validate a software system, and Malik et al. [25] also use user-provided test scenarios to generate test cases.

All of these approaches and studies have shown that scenarios could be a useful resource that can be utilized throughout the entire requirements engineering processes to improve software quality. The approach we propose also uses scenarios in order to validate requirements, but our approach is different from prior research: we provide a feedback-based requirements validation method, which facilitates a means for users to interact with a requirements model and to validate its behavior through a set of queries from scenarios and query results. SQ querying, with its reliance on a user's framing of the questions, brings human cognition elements into a validation process, as is the case with requirements review. The ability to frame questions that describes a scenario allows the domain expert to think on a more end-to-end scenario basis.

Our approach also differs from other scenario-based approaches in that the human in the loop is utilized as an important part of the SQ process (Fig. 8). The involvement of the human analyzer in the SQ querying process adds a human interpretation element; seeing the various path strings for a given query can often bring to mind other questions, which may lead to previously unconsidered scenarios, which ultimately leads to a more in-depth analysis. Scenarios are thus formulated dynamically in response to feedback. Since each new scenario question, in turn, provides feedback, the result is a more immediate and detailed cycle of inquiry and response, as shown in Fig. 8. Thus, while other approaches often try to automate as much as possible, we purposely strived to integrate the cognitive element.

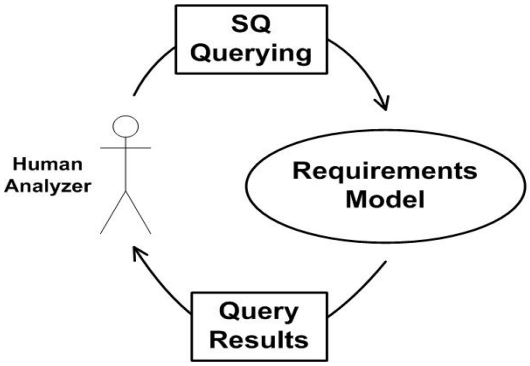

Figure 8. Human intervention in SQ querying.

V. DISCUSSION, CONCLUSIONS, AND FUTURE WORK

We have introduced a new feedback-based requirements validation approach that provides an interactive way to validate behaviors of requirements models using scenario questions, and have presented a preliminary case study performed on a real-world application. The approach is based on the notion of querying a model with scenario questions, in order to determine whether a model's behavior satisfies the given requirements.

We believe that our approach has the potential to be highly useful in validating requirements models because it leverages the advantages of the review/inspection process that has been widely used for requirements validation, while minimizing the difficulties associated with coordinating people in the thorough reading of lengthy documents by allowing the users to ask questions about system behaviors specified by the requirements. The main contributions are as follows:

- A technique that provides engineers with an interactive approach to validate and improve a requirements model, which requires less training and is easier for practitioners to adapt into practice than formal methods.
- The behavioral path string that displays the answer to a SQ query makes human engineers analyze the outcome easy to understand, and to convey to end-users. It also helps to build the common understanding of behaviors of the requirements and as a result, it reduces the gap among diverse stakeholders' understanding (i.e., customers, requirements engineers, developers, etc.)
- Through our preliminary case study, we found its applicability and potential effectiveness of the proposed approach, and obtained positive feedback; we discovered several ambiguous and unexpected behaviors from the model under study.

Because of these promising initial results, we plan to perform further research on the SQ querying concept. Our first goal is to automate the interpretation of path strings by having a tool automatically create a set of metrics that summarizes path string results. The objective is to minimize the amount of visual analysis that the user would need to perform by reducing his/her need to examine every path string when trying to assess results.

To explicitly address states and transitions during the framing of questions, we have proposed a method of translating the requirements into a STD [27]. The method results in traceability between NL requirements and its STD model, which in turn enables any SQ derived from requirements, to use the same states and transitions which appear in the STD. To address the scalability problem, we can either use a state reduction technique, such as OBDD (Ordered Binary-Decision Diagrams), or develop a means of representing the STD in a higher level of abstraction. One way could involve partitioning the state diagram into regions of functionality and then representing each region as a specific state. This is similar to creating a hierarchy of states. Other improvements include: a plan to automatically translate scenario questions into query cases. This would allow the human engineers to query the model in natural language without having to manually translate the question into inference constraints. To this end, one possible solution would be to develop a subset of the English language consisting of keywords; an approach based on the translation guidelines presented in the framework section.

The above plans, consequently, will improve our technique so that we can address scalability problems of models in a large problem space by combining with the approaches on problem decomposition and divide-and-conquer. Finally, we plan to conduct additional empirical studies using different types of models (not only limited to State Transition Diagrams) in larger and complex software systems to generalize our findings and to improve our approach. In particular, additional empirical studies will provide us good insights into what types of characteristics of the requirements model's behavior should be further considered, and will lead to a better understanding of the capacity of the framed querying mechanisms.

The concepts introduced in this paper can also be reapplied in the following possible application areas:

- *Regression testing of model design iterations*: A query's resulting path strings can be quantified into pass-fail criteria for use in automated regression testing. This makes it possible to develop a test suite of scenario questions that can be used to repeatedly query a model as it goes through design iterations. Since SQs are directly tied to requirements, they can be added or removed as requirements change.
- *Randomly exposing detrimental paths*: The Prolog reasoning engine with its six inference constraints can be used to randomly expose detrimental path scenarios in a safety critical application. A human tester could quantify the detrimental path strings as constraints could then be randomly applied in different combinations, with the resulting path

strings for each combination capture and compared against the fail criteria. The random testing would stop whenever an unwanted path string appears. The tester would then make note of what combination of constraints produced the unwanted path string, and make any necessary adjustments to restrict such a combination from appearing during normal operation.

- *A semantic net knowledge-based system*: A semantic net can be used to represent a knowledge domain of non-functional requirements. Structurally, semantic nets are similar to state transition diagrams. With proper adjustment of the Prolog rules, a semantic net, represented as Prolog facts, can be queried. Each resulting path string could represent an answer to that query. This could enable us to consider "non-functional" perspective of the requirements in addition to the functional perspective.

REFERENCES

[1] B. Boehm and V. Basili, Software defect reduction top 10 list, IEEE Computer, 2001.

[2] F. Shull et al., What we have learned about fighting defects, in Proceedings of the International Software Metrics Symposium, Jun. 2002, pp. 249-258.

[3] K. Wiegers, Creating a Software Engineering Culture, Dorset House Publishing Company, Aug. 1996.

[4] K. Wiegers, Software Requirements, 2nd ed. Microsoft Press, 2003.

[5] H. Hofmann and F. Lehner, Requirements engineering as a success factor in software projects, IEEE Software, vol. 18, no. 4, pp. 58-66, 2001.

[6] I. Sommerville and P. Sawyer, Requirements Engineering: A Good Practice Guide, Wiley, 2006.

[7] I. Bray, An Introduction to Requirements Engineering, Addison Wesley, 2002.

[8] J. Ryser and M. Glinz, SCENT: A method employing scenarios to systematically derive test cases for system test, University of Zurich, Technical Report, 1999.

[9] H. P. Afshar, H. Shojai, and Z. Navabi, A new method for checking FSM correctness, in Proceedings of the International Symposium on Telecommunications, Aug. 2003.

[10] N. Maiden, SAVRE: Scenarios for Acquiring and Validating Requirements, Journal of Automated Software Engineering, no. 5, pp. 419-446, Aug. 1998.

[11] T. Chow, Testing software design modeled by finite-state machines, IEEE Transactions on Software Engineering, 4(3), May 1978.

[12] I. Bratko, Prolog Programming for Artificial intelligence, 3rd ed. Addison Wesley, Sep. 2008.

[13] R. Kazman, G. Abowd, L. Bass, and P. Clements, Scenario-based analysis of software architecture, IEEE Computer Society Press, 2009.

[14] J. Ryser, S. Berner, and M. Glinz, A scenario-based approach to validating and testing software systems using statecharts, in Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications, 1999.

[15] C. Damas, B. Lambeau, and A. van Lamsweerde, Scenarios, goals, and state machines: A win-win partnership for model synthesis, in Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering, Nov. 2006, pp. 197-207.

[16] D. Aceituna, Validating requirements models using SQ querying, MS Thesis, North Dakota State University, Feb. 2009.

[17] J. Whittaker, Exploratory Software Testing, Addison Wesley, 2010.

[18] A. Sutcliffe and M. Ryan, Experience with SCRAM, a SCenario Requirements Analysis Method, Third International Conference on Requirements Engineering, 1998, pp. 164-171

[19] C. Damas, B. Lambeau, P. Dupont, and A. Lamsweerde, Generating annotated behavior models from end-user scenarios, IEEE Transactions On Software Engineering, 31(12), 2005, pp. 1056-1073.

[20] E. Letier, J. Kramer, J. Magee, and S. Uchitel, Monitoring and control in scenario-based requirements analysis. In Proceedings the 27th international conference on Software engineering, St. Louis, MO, USA, 2005, pp. 382-391.

[21] A. Sinha, S. Sutton, and A. Paradkar. Text2Test: Automated inspection of natural language use cases. In Proceedings of the International Conference on Software Testing (ICST), Paris, 2010, pp. 155–164.

[22] A. Sutcliffe, N. Maiden, S. Minocha, and D. Manuel, Supporting scenario-based requirements engineering, IEEE Transactions on Software Engineering, 1998, 24(12), pp 1072-1088

[23] P. Gough, F. Fodemski, S. Higgins, and S. Ray, Scenarios—An industrial case study and hypermedia enhancements, in Proceedings of the second IEEE Symposium on Requirements Engineering, IEEE Computer Society, 1995, pp. 10–17,

[24] T. Royer, Using scenario-based designs to review user interface changes and enhancements, in Proceedings of DIS95: Designing Interactive Systems, Ann Arbor, 1995, pp. 237–246.

[25] Q. Malik, J. Lilius, and L. Laibinis, Scenario-based test case generation using event-B models, in Proceedings of the International Conference on Advances in System Testing and Validation Lifecycle, 2009

[26] S. Uchitel, J. Kramer, and J. Magee, Synthesis of behavioral models from scenarios, IEEE Transactions on Software Engineering, 29(2), 2003, pp 99-115.

[27] D. Aceituna, H. Do, S.W. Lee, A Human interactive approach to building requirements models, Technical Report, NDSU-CS-TR-10-001.