Interactive Requirements Validation for Reactive Systems through Virtual Requirements Prototype

Daniel Aceituna
Computer Science Department
North Dakota State University
Fargo, ND, USA
daniel.aceituna@ndsu.edu

Hyunsook Do
Computer Science Department
North Dakota State University
Fargo, ND, USA
hyunsook.do@ndsu.edu

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

Abstract - Adequate requirements validation can prevent errors from propagating into later development phases, and eventually improve the quality of software systems. However, validating natural language requirements is often difficult and error-prone. An effective means of requirements validation for embedded software systems has been to build a working model of the requirements in the form of a physical prototype that stakeholders can interact with. However, physical prototyping can be costly, and time consuming, extending the time it takes to obtain and implement stakeholder feedback. We have developed a requirements validation technique, called Virtual Requirements Prototype (VRP), that reduces cost and stakeholder feedback time by allowing stakeholders to validate embedded software requirements through the interaction with a virtual prototype.

I. INTRODUCTION

When it comes to validating software requirements, a key question is whether the captured requirements adequately specify all the functionalities that stakeholders expect from the system being built [2]. Lack of adequate specifications is often due to incompleteness, ambiguities [1], and inconsistencies [19] that commonly occur in a set of natural language (NL) requirements. These errors are hard to expose, and various means of exposing them have been utilized in the industry [2].

One common requirements validation approach is human inspection, where stakeholders inspect the requirements, looking for the aforementioned faults, often using a checklist as a guide. Inspection provides the advantage of getting the non-technical stakeholders involved in the validation process. However, validating requirements via an inspection checklist can be problematic when trying to validate all the possible functional interactions by inspecting one requirement at a time. This problem is even more pronounced in embedded software systems, where the correct interaction of certain functionalities represent characteristics unique to embedded systems, must be addressed during validation [20, 21]. The key characteristics of embedded systems [9] are:

- *Aperiodic stimuli* – Handling interrupts. Embedded systems must be able to properly handle unpredictable events that take priority over whatever the system is doing at the time of the interrupt. Furthermore, the system must be able to recover from an interrupt and resume the program flow that was occurring prior to the interrupt. Interrupts are often generated by the operating environment. Some interrupts are internally generated as well.
- *External stimuli* – Embedded systems typically have events generated by the external and operating environments. These include user-driven controls, and signals from sensors.
- *System delays* – The time delay between state transitions can be just as critical as the transitions themselves. They need to be validated, as well.
- *Periodic stimuli* – These are timed events and/or timed transitions. They occur periodically, regardless of what else is happening in the system.
- *Concurrency* – Multiple processes can run simultaneously.

Formal approaches, such as model checking [16, 24], Petri Nets [14], or other model-based methods [21], are better suited for validating the aforementioned characteristics of embedded software systems. However, due to the technical training needed to apply such formal methods, their user groups are often limited to people who have technical background, thus they fail to include other important stakeholders in the requirements validation process.

Another validation approach, prototyping, also can model the embedded system characteristics as formal approaches, but unlike formal approaches, it can involve non-technical stakeholders in the validation process. However, embedded system prototypes tend to be physical hardware prototypes, which are often developed beyond the requirements phase, and used to validate software and hardware. In some cases, they can be used to validate requirements, but involve higher cost and longer stakeholder feedback time. It is also impractical to recreate and reevaluate a new physical prototype every time a requirement is changed, corrected, or added.

To obtain the benefits of the aforementioned approaches, and while addressing their drawbacks, we propose a new validation approach and supporting tool called Virtual Requirements Prototype (VRP). VRP provides a means of prototyping embedded system requirements, which results in a working model of the behaviors expressed in the requirements. The idea is to make the requirement behaviors operational, so that the stakeholder can validate that the

978-1-4577-0959-3/11/$26.00 ©2011 IEEE

right system (behavior-wise) is being represented by the requirements. The method is designed to expose errors both during the prototype's creation, and subsequent evaluation.

In Section II we will describe the primarily motivation behind VRP. Section III of the paper will describe the elements that make up the VRP approach. Section IV will show how the approach works, by using an example set of requirements, in Section V we briefly review what is being done in the field, in relation to our work, and in section VI, we close with some closing thoughts and future work on the VRP method.

II. BACKGROUND

The primary motivation behind VRP is to provide a requirements validation approach that overcomes drawbacks of current approaches we mentioned in Section I and brings stakeholders into the validation process. Validation generally addresses the question of whether the right system has been built. This question can be best answered by various stakeholders, such as customers, domain experts, and end-users, who typically are the ones requesting the system-to-be.

Inspection is one of the widely accepted and used validation techniques because it is relatively easy to apply and it does not require technical knowledge for stakeholders to use it, but it is error prone due to human limitations in finding faults in documents describing complicated reactive systems. Therefore, requirement engineers often seek to validate requirements in ways that minimize human errors, but this means they need to use techniques that extend beyond the technical expertise of non-technical stakeholders This creates a situation in which stakeholders (e.g., end-users and domain experts), who should be performing the validation, may not be able to do so, because requirement engineers could be the only qualified people who can perform the validation using such techniques.

In developing the concept of VRP, we were motivated by several factors that allow for stakeholders' participation. These factors include: 1) The use of a virtual prototype that allows the stakeholders to validate through their interactions; 2) Stakeholders can incrementally create the prototype themselves through a question-based means of transforming NL requirements into a Virtual Prototype (VP); based on a concept we developed called NLtoSTD [23].

Stakeholders can evaluate the VP by using a technique we developed called SQ Querying, which is a question-answer based approach of interrogating a set of requirement with Scenario Questions [22]. We facilitate the asking of questions by using matrix based user-interfaces. To answer the stakeholder's questions, we use easy-to-read path strings and requirement scenarios, which will be covered in detail. Lastly, with the emphasis of human involvement, the VRP supporting tool will incorporate automated reasoning.

III. METHODOLOGY

VRP is an approach to validating NL functional requirements by transforming them into a virtual prototype that stakeholders can then interact with. There are three major phases to VRP, as shown in Figure 1. Each phase can be revisited at any time during the course of validating a set of requirements. The three phases are as follows.

1) Virtual Prototype Creation Phase

Users transform the Natural Language (NL) requirements into a Virtual Prototype (VP). In this phase NL ambiguities can be greatly reduced, because the transformation process forces a reevaluation of the users' true intention, as worded in the requirements. The transformation process is such that a direct one-to-one trace between each NL requirement and a building block of the VP is established (the concept of a building block will be explained in subsection B.).

2) Virtual Prototype Correction Phase

Users make any necessary initial corrections in the VP's structure. In particular, users address gaps in the VP, which are direct reflections of the incompleteness in the NL requirements. Since the first phase established a direct one-to-one trace between NL requirements and the VP, any corrections made to the VP can instantly be mapped back to the responsible NL requirement.

3) Virtual Prototype Evaluation Phase

After the initial corrections have been made to the VP, it is ready for users' evaluation. This phase exercises VP behavior, and uses STD traversal path strings and requirement scenarios to expose defects, including inconsistencies.

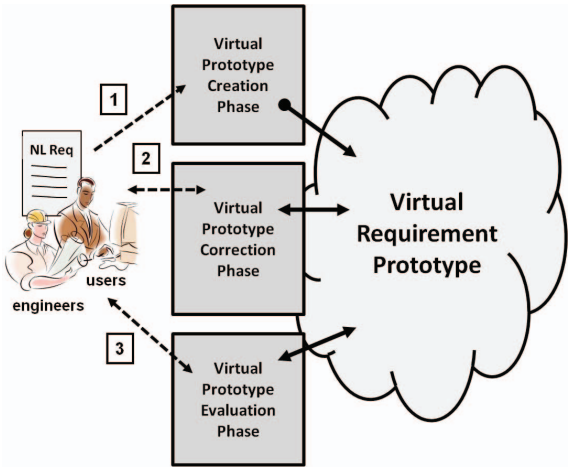

Figure 1. An overview of Virtual Requirement Prototype

VRP stresses all stakeholders' involvement in all three phases. Stakeholders' involvement in the evaluation phase is an important benefit of using a prototype. Getting all stakeholders involved in the creation and correction phases exploits some of the benefits associated with requirement inspection. Throughout all three phases, a direct traceability between components of the virtual prototype and each individual requirement is preserved. This means that prototype corrections made during any phase can be traced

back to a specific NL requirement, which in turn can be corrected. This process will be explained in detail using an example in Section IV. The following subsections provide detailed descriptions for each of the three phases.

B. *Virtual Protoype Creation Phase*

During this phase, users will manually transform a set of requirements into a VP. The transformation process occurs one requirement at a time, by having users answer questions pertaining to the given requirement being transformed (question details are covered in section IV). The answers are entered into a transformation questionnaire provided by the VRP supporting tool (Figure 3, (A)). From these answers the tool internally constructs the Virtual Prototype (Figure 2).

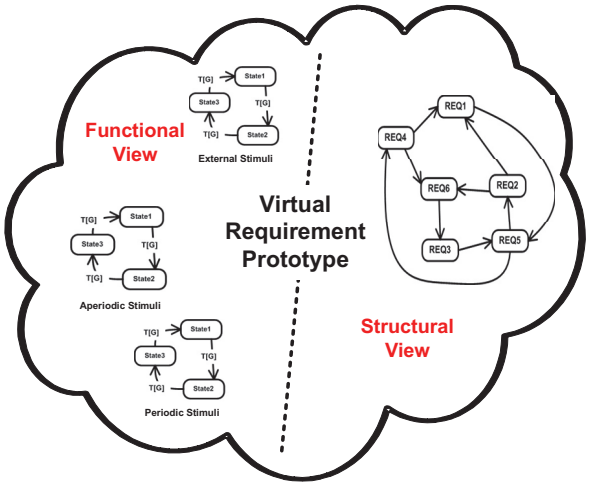

Figure 2. The various models that make up the VRP

The VRP represents the behavior and structure of the requirements. This dual representation is achieved by using two models types in the VRP (Figure 2). One model is a State Transition Diagram (STD) that provides a functional (behavioral) view of the requirements; the other model is what we call a Requirement Graph (RG) that provides a structural view of the requirements. A RG models the relationship between the requirements and how they collaborate with one another (we will explain this concept in detail in Section IV). The functional view can consist of multiple STDs, with each STD modeling a different embedded system characteristic.

For example, Figure 2 shows a VRP with three STDs making up the functional view. These three STDs can model a set of requirements that specify interrupts (a-periodic stimuli), user-driven controls and sensors (external stimuli), and timed events (periodic stimuli). Separate modeling is possible because of the *"entity(state)"* format used to build the models (section IV). By modeling embedded system characteristics separately, we can independently analyze and correct their behavioral effects on the VRP. An incorrectly transformed requirement can result in a false defect. To minimize this occurrence, the VRP supporting tool will automatically verify that each transformed requirement has the proper format. We now explain the reasoning behind two key elements in the creation phase.

The use of a questionnaire to guide the user in the tranformation of NL requirements – To implement this component, we have developed a six question transformation questionnaire, as shown in Figure 4. The users' response to the questions are used to form State Transition Diagram Building Blocks (STD-BBs) [23], which in turn, are used to build the STDs. The same building blocks are used to build the RG. The questions also help expose ambiguities that may exist in the NL requirements. This is because while answering the six transformation questions, users are forced to verify the intended meaning of a requirement's wording, if and when that wording can result in more than one answer. Having users answer questions of a requirement also introduces one of the benefits gained from using a checklist inspection approach, namely, that users are forced to examine the requirement for answers, increasing the likelihood of finding other errors. The questionnaire is also useful in assuring that everything needed to form a STD-BB has been captured.

Using a STD and a RG to represent the Virtual Prototype – We use two models to provide a two dimensional view of the virtual prototype by modeling the prototype's behaviour and structure. A State Transition Diagram is a common way to model system behaviour. It is also state-based, which lends itself to modeling embedded systems [21], including the embedded system charateristics mentioned in the introduction. A Requirements Graph shows how requirements interrelate with one another, which models the prototype's structure. This dual view plays an important role in the subsequent exposure of incompleteness and inconsistancies.

C. *Virtual Prototype Correction Phase*

During the creation phase, any incompleteness in the requirements will manifest itself in the virtual prototype as an incomplete STD and/or an incomplete RG. An incomplete STD, can expose missing information within a given requirement, while an incomplete RG can expose an entire requirement missing from the set of requirements. The VRP supporting tool will enable users to both detect and correct incompleteness in the VP. Since the STD and RG models are digraph structures, the tool will also use rules written in Prolog to analyze both models and alert users of potential incompleteness.

Using a matrix to represent the STD and RG to Users -- We use an adjacency matrix as a means of exposing incompleteness in a STD. Users can quickly see which requirements, states, and/or transitions are missing by detecting the missing adjacencies in the STD and RG matrices. The matrices also provide a tabular format interface for both modifying the STD and RG, which would help address the problems associated with trying to display a large STD to the user.

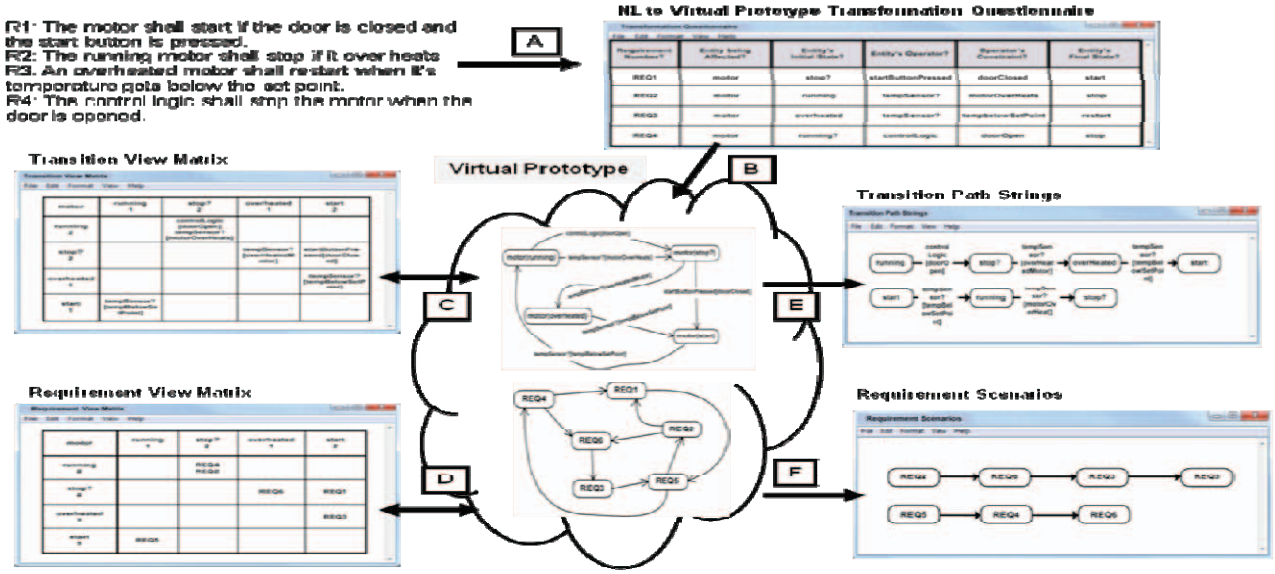

Figure 3. The VRP Process

D. *Virtual Prototype Evaluation Phase*

During the evaluation phase, users can examine the various behaviors that the virtual prototype can produce. Examining these behaviors help expose inconsistencies and other defects that may exist in the requirements. System behavior is displayed to users as STD path strings and requirement scenarios. Path strings are generated by traversing the STD and displaying each path traversal as a string of states and transitions. Requirement scenarios are generated by traversing the RG and displaying each path traversal as a sequential listing of the actual requirements.

Users can have either the STD path strings (Figure 3, (E)) or RG requirement scenarios (Figure 3, (F)) displayed in one of two modes: unconstrained, or constrained. In the unconstrained mode, all the possible strings and/or scenarios are displayed. Users then analyze each possible behavior. However, as the virtual prototype grows, the number of strings and scenarios grow as well, resulting in a large number that is harder to analyze. So the tool can generate path strings and scenarios in a constrained manner, by allowing users to select which states, transitions or requirements the VP path traversals are constrained to. The constraints can be selected through either an adjacency matrix, or by asking Scenario Questions [22].

Using STD path strings and requirement scenarios - Each path string describes a behavior defined by the requirements, as expressed by a sequence of one entity's state transitions. Each requirement scenario is a system behavior that is described when reading the requirements in a specific sequence. For example, given requirements REQ1, REQ2, REQ3, and REQ4, it may be that one possible system behavior occurs when REQ4 is enacted before REQ3, which, in turn, is enacted before REQ1. In the resulting requirement scenario the users would then read the requirements in the following sequence: REQ4, REQ3, and REQ1.

Path strings and requirement scenarios offer two different perspectives to help expose inconsistencies. Paths strings describe a specific entity's behavior, while requirement scenarios describe behavior on a system level, revealing the interactions between more than one entity. By examining path strings and requirement scenarios, users can assess if the requirements are specifying the desired behaviors, as well as determine if there are undesirable behaviors that should be addressed.

IV. Tool Support and Motivational Example

The VRP process is managed by a supporting tool. The tool's user interface and related logic are written in C#, while its reasoning engine is written in Prolog. We use Prolog facts to represent the VP, and Prolog rules to perform some automated reasoning. The VRP process is shown in Figure 3 with the interface snapshots of the tool's major components. To enable interaction with the virtual prototype, the tool provides users with two matrix-based interfaces and two displays. The tool is composed of the following components (also shown in Figure 3).

(A) NL to Virtual Prototype Transformation Questionnaire – This interface consists of a questionnaire that enables users to transform the requirements into the Virtual Prototype (B).

(C) Transition View Matrix– This interface consists of a transition adjacency matrix for each STD. Users can detect and correct defects in the STD via this interface.

(D) Requirement View Matrix – This interface consists of an adjacency matrix for the RG. The users can interact directly with the matrix, and by doing so, are interacting with the RG.

(E) Transition Path Strings Display – It displays path strings that are either unconstrained or have been selectively constrained by the users.

(F) Requirement Scenarios – It displays the requirement scenarios that are either unconstrained or have been selectively constrained by the users.

Currently, the VRP supporting tool provides a matrix interface that visualizes the STD and RG, but future iterations of the tool will provide a visual interface that shows actual diagrams of the STD and RG.

We will follow an example through the three phases described in section III. We begin by explaining how VRP views a typical requirement.

The VRP process operates on the assumption that in a typical set of reactive NL requirements, there are entities that are described as changing states. For example, in a set of requirements specifying an elevator system, the three *state changing entities* could be the "door", the "elevator motor", and the "cab request button." Using an "*entity(state)*" notation, we can describe the entities and their states in the following manner:

- "door(Open)", "door(Closed)".
- "elevatorMotor(running)" , "elevatorMotor(stop)".
- "cabRequestButton(pressed)".

Often, the requirements will also specify how a given entity transitions from one state to another. Thus, in the requirement *"The running motor shall stop if it overheats,"* the motor is the entity transitioning from a "running" state to a "stop" state, when its temperature excesses a certain point.

Another example would be: "*The door shall be open and closed by the operator*", in which case, the door is the entity transitioning from an "open" state to a "closed" state, by the operator. Note that in both examples, the transitioning from one state to another is caused by a *transition action*. In the case of the motor, the action is that the motor is "overheating." The action that transitions between the door's two states is the "operator."

A. Virtual Protoype Creation Phase

To demonstrate the creation of a virtual prototype, we will use the four requirements in Listing 1. These four requirements partially describe the behavior of the motor, door, and start button entities, in an elevator system. For the sake of space, we will focus on just the "motor" entity, and the creation of its STD.

Listing 1. Four sample requirements.

REQ1: The motor shall start if the door is closed and the start button is pressed.
REQ2: The running motor shall stop if it overheats
REQ3: An overheated motor shall restart when its temperature gets below the set point.
REQ4: The control logic shall stop the motor when the door is opened.

To create a virtual prototype, users enter pertinent data obtained from the requirements into the Transformation Questionnaire provided by the VRP tool. An example of the Transformation Questionnaire, with the Listing 1 requirements entered, is shown in Figure 4. The top row displays the six questions, and beginning with the second row, each row displays the answers corresponding to each requirement.

We now describe each of the six questions being asked of users.

1. *Requirement Number?* – This is the requirement number, which the tool uses when generating Requirement Scenarios.

2. *Entity Being Affected?* – This is the entity being affected by the requirement. Note that in REQ1 there are two entities (motor and door). The tool allows for the entry of more than one entity per requirement, providing users label them accordingly. For example, the motor entity would be labeled REQ1a, while the door entity becomes REQ1b.

3. *Entity's Initial State?* – This is the entity's initial state; this is the state the entity is in prior to being changed by the requirement. For example, REQ2's initial state is "running" as explicitly stated in the requirement. By contrast, REQ1's initial state is implied. Users assume that the motor is not moving prior to starting. This is why the entry "stop?" has a question mark. By adding a question mark, users are indicating that they have made an assumption that needs addressing by conferring with the other stakeholders, if necessary.

4. *Entity's Operator?* - This is the device or action that results in the entity changing state. In REQ1, the operator's action is clearly stated as the "start button being pressed". In REQ2, users assumed that there is a temperature sensor that detects the motor's temperature, thus the question mark is added.

5. *Operator's Constraint?* - This is the condition upon which the operator is allowed (or disallowed) to act. In REQ1, the start button starts the motor if the door is closed. In REQ2, the assumed temperature sensor (tempSensor) stops the motor if the temp is such that the motor overheats (motorOverHeats).

6. *Entity's Final State?* –This is the state the entity is in after being changed by the requirement. For example, REQ2 describes the motor (the entity) as going from a "running" state to a "stop" state. "Stop" is the motor's finals state.

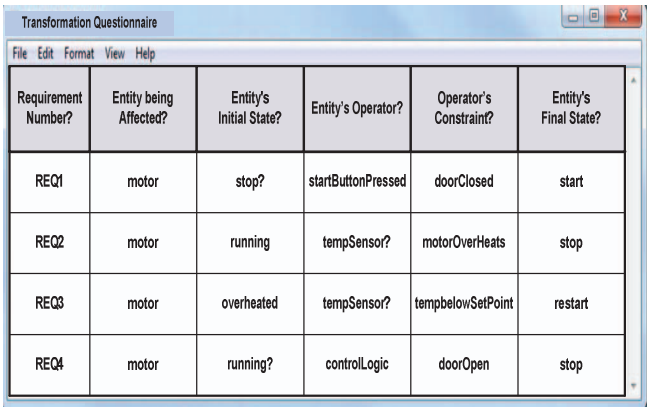

Figure 4. The NL to VRP Transformation Questionnaire

Data entered into the questionnaire is then used to produce STD building blocks (STD-BBs), which are the {state, transition, state} segments that make up the STD. In our example, each requirement produces one STD-BB; the four are shown in Figure 5. We call them building blocks, because they are used to construct the State Transition Diagram (STD) that will model half of the virtual prototype.

Note that the states on each end of the building blocks have the *entity(state)* format. This allows for the creation of entity specific STD-BBs that would subsequently create *entity specific STDs*. For example, after users entered data pertaining to the door entity, a set of door building blocks would have created, which in turn, would result in a STD that models the door. There can be multiple entity-specific STDs in one virtual prototype, which allows the prototype to provide different system perspectives and makes possible the modeling of the embedded system characteristics as mentioned in the introduction. In our example, the prototype has only a motor perspective. By connecting the matching states of the building blocks in Figure 5, the State Transition Diagram in Figure 6 is produced.

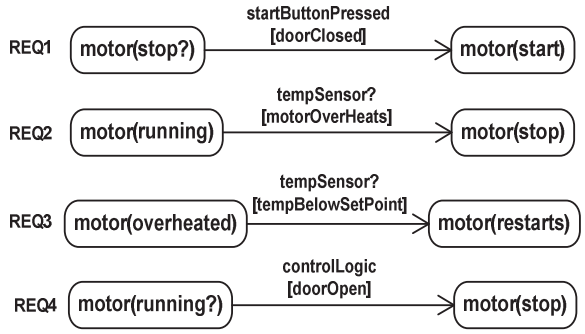

Figure 5. The four STD building blocks produced by the four sample requirements

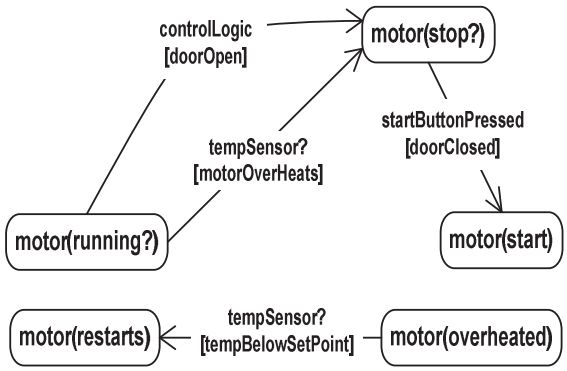

Figure 6. The State Transition Diagram created from STD-BBs.

B. Virtual Prototype Correction Phase

The Virtual Prototype Creation phase results in a STD (and Requirements Graph) whose building blocks have a one-to-one mapping directly back to the requirements. The direct mapping between STD and the requirements gives users an immediate means of making corrections to the requirements as incompleteness is exposed in the STD. For example, the STD of Figure 6 is disconnected, not every state has a path to every other state; some states have no incoming or outgoing transitions from/to other states Therefore, it can be argued that a disconnected STD is a STD that is incomplete. Consequently, a disconnected STD reveals a set of requirements that are incomplete. By correcting incompleteness in the STD, users can correct the corresponding incomplete requirements as well. The tool's Prolog-based automated reasoning will also help expose incompleteness, by looking for graph properties in the STD and RG that indicate a disconnected graph.

Incompleteness in the STD is displayed to users using the Transition View Matrix, shown in Figure 7. When examining the matrix of Figure 7, STD's incompleteness is evident by the fact that the "start" state along the left-hand column has no entries in its corresponding row (fifth row, from the top). This means the "start" state is not transitioning to another state.

By examining the matrix in the Figure 7, users can raise the following three questions:

1) "Can the restart state be combined with the start state?"

2) "How does an overheated motor start?" (as indicated by the dot lines and the question mark).

3) "How does a motor transition from a start state to a running state?" (as indicated by the question mark).

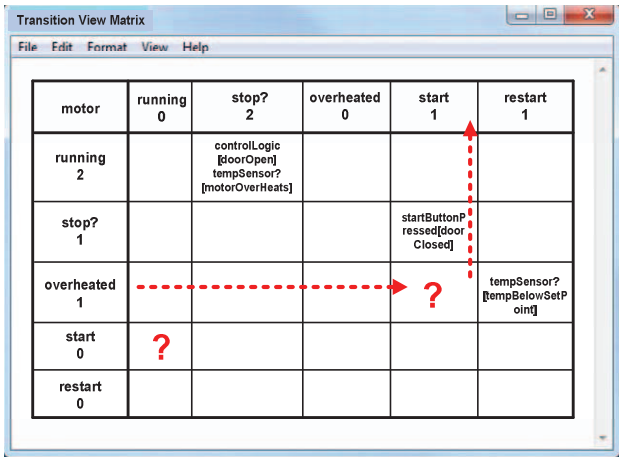

Figure 7. Transition View Matrix, showing an incomplete STD.

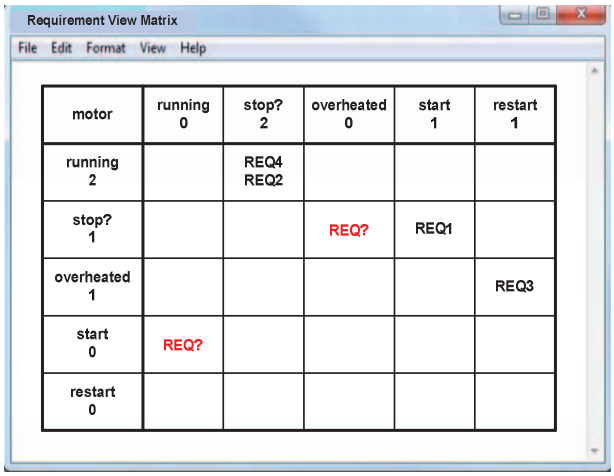

Figure 9. Requirements View Matrix Interface

In respond to these questions, users can delete or rename states, and/or specify the missing transitions needed to connect states to one another. Changes made to the matrix, propagate into changes to the Virtual Prototype's STD. Any correction made to the matrix can be easily traced back to the NL requirements, due to the one-to-one mapping that occur in the prototype creation phase.

The numbers associated with the state names are the number of incoming (top row), and outgoing (left-hand column) transitions. Users can also use these numbers to quickly identify incompleteness by looking for states that have a "0". The STD is one of the models that represent the Virtual Prototype. The other model is the Requirements Graph, shown in Figure 8.

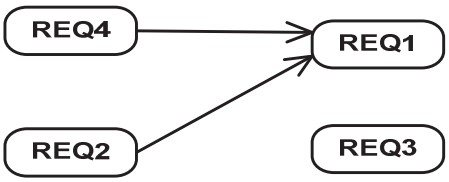

Figure 8. The Requirement Graph created from STD-BBs of Figure 3.

The tool creates the model in Figure 8 from the STD building blocks of Figure 5. As with the STD, the Requirements Graph is accessed by users, via its own Requirement View Matrix interface, shown in Figure 9.

Using the matrix, users can question if there are missing requirements by making note of any two states that should be transitioning into each other. For example, in Figure 9, we see that there is presently no requirements that describe how the motor goes from a "start?" state to a "running?" state. We have placed a "REQ?" in red in the cell adjacent to both states to indicate that perhaps a requirement is missing, and should be added. By using their respective matrices, both the STD and Requirements Graph can be corrected by either renaming some states, or adding more transitions to close any gaps in the STD, or more requirements to close existing gaps in the Requirements Graph. This equates to either changing the state names in the requirements, or adding more requirements. Figure 10 shows a corrected Transition View Matrix, after the three raised questions are addressed.

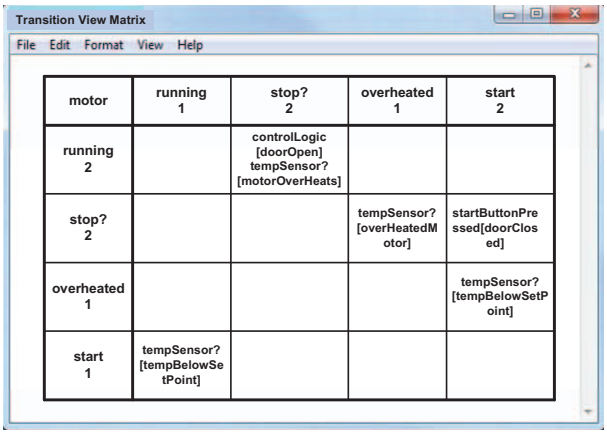

Figure 10. Revised Transition View, corresponding to a corrected STD.

The resulting corrections made to the Requirements View Matrix are shown in Figure 11. Corrections made to the matrices can be easily traced back to the NL requirements due to the one-to-one tracing between the two matrices and the set of NL requirements.

Thus, we have revised the original set of four requirements, by combining two states and adding two requirements. The revised set of requirements is now:

- REQ1: The motor shall start if the door is closed and the start button is pressed.
- REQ2: The running motor shall stop if it overheats
- REQ3: An overheated motor shall **start** when its temperature gets below the set point.

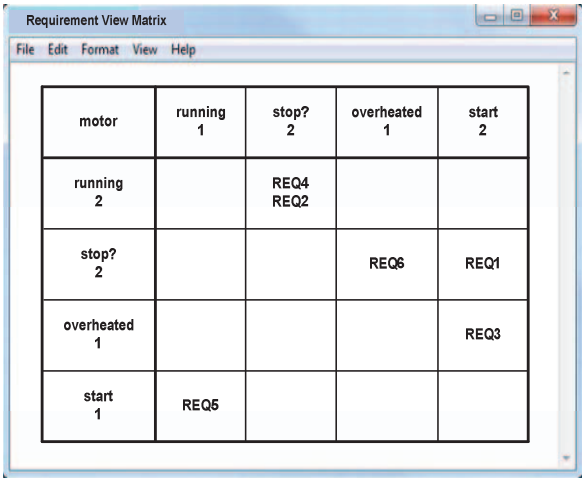

Figure 11. Revised Requirements View, corresponding to a corrected Requirements Graph.

- REQ4: The control logic shall stop the motor when the door is opened.
- **REQ5: When the motor starts it will assume a running state if its temperature is below the set point.**
- **REQ6: A stopped motor that is overheated will be put in an overheated state.**

This new set of requirements addresses the incompleteness present in the original set; the revisions are in bold. We are now ready to move into the next and final phase.

C. *Virtual Prototype Evaluation Phase*

1) *Path String Generation*

The completed Virtual Prototype can now be used to generate both STD path strings and Requirement Scenarios. Figure 12 shows two path strings from the completed STD.

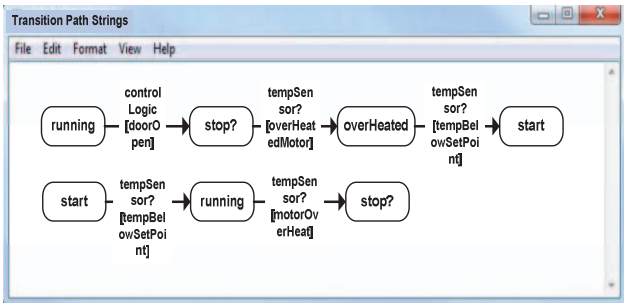

Figure 12. Two path strings generated by the VRP Tool from the completed Transition View Matrix in Figure 8

By examining a given path string, users can validate if that behavior is correct, and/or if there is an unexpected behavior that should be corrected in the requirements. For example, from the upper path string in Figure 12, we see that a running motor stops when the door is open, but the "stop?" state is exited if the motor is "overheated". This raises the question whether there is another way to exit a "stop?" state. In response, users can generate and examine other path strings to verify whether there is another way to exit a "stop?" state. Aside from generating another path string, the matrix representation allows users to quickly check for an alternative exit. In this case, the "stop?" has a outgoing number of 2, which quickly tells users of another exit. Further examination of the matrix reveals that the "stop?" state can transition to a "start" state via the "startButtonPressed[doorClosed]" transition.

Not only can the VRP tool be asked to generate all the possible path strings that can be derived from the Transition View Matrix, but the tool can also be asked to generate a set of path strings that satisfy one or more constraints. The ability to query the requirements is based on a concept developed by the authors called SQ Querying [22]. For example, referring again to the matrix in Figure 10, we could ask the tool to generate all the path strings that begin with a "stop?" state and end with a "start" state. That would generate all the ways that a stopped motor can be started. That would address the prior question we raised concerning the exit of a "stop?" state. The constraints can even be phrased as a question, such as: *"How many ways can a stopped motor start again?"* In this regard, users would be querying the requirements for a specific behavior. This is similar to users validating a physical prototype, by interacting with the prototype, while looking for certain behaviors.

2) *Requirement Scenario Generation*

With the Requirement View matrix filled in (Figure 11), the Requirement Graph is connected and can be evaluated by generating path traversals. Two such path traversals are shown in Figure 13. If we consider that each requirement is contributing to the overall functionality of the system, then by listing the requirements in the sequence that they can be enacted, we can literally read the various behaviors that the system will exhibit.

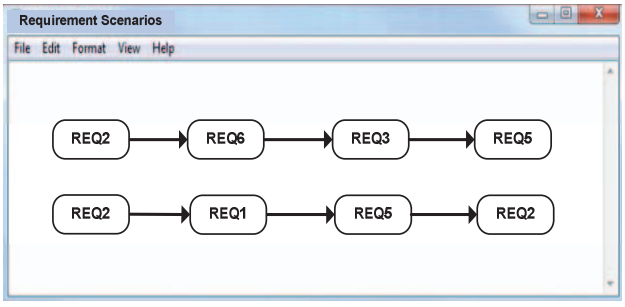

Figure 13. Two Requirement Scenarios generated by the VRP Tool from the completed Requirement View Matrix in Figure 11

We call these path-derived behaviors as Requirement Scenarios, because we are creating readable scenarios with the requirements themselves. Listing 2 shows the Requirement Scenario created from the path traversal in the upper sequence in Figure 13.

Listing 2. The top Requirements Scenario from Figure 13

1. REQ2: The running motor shall stop if it overheats
2. REQ6: A stopped motor that is overheated will be put in an overheated state.
3. REQ3: An overheated motor shall start when its temperature gets below the set point.
4. REQ5: When the motor starts it will assume a running state if its temperature is below the set point.

As we read the requirements in the order listed in Listing 2, we find that the described behavior can be described as an overheated motor scenario. The scenario appears reasonable, and all contingencies seem to be addressed. A second path traversal is made (lower string, Figure 13), with the resulting Requirement Scenario in Listing 3. From reading Listing 3, we notice that a motor that has over-heated is not placed in an "overheated" state, as in Listing 2. The Listing 3 scenario has no "overheated" state for the motor to wait in until it cools down. REQ5 will still prevent the motor from running if it is too hot, but users must decide if not going through an "overheated" state is a potential problem.

The Requirement Scenario of Listing 3 raises another question. If the motor overheats at about the time that the door is closed, and someone presses the start-button, will the motor try to immediately start again, since it will not spend time in an "overheated" state, until it cools down?

Listing 3. The lower Requirement Scenario from Figure 13

1. REQ2: The running motor shall stop if it overheats
2. REQ1: The motor shall start if the door is closed and the start button is pressed.
3. REQ5: When the motor starts it will assume a running state if its temperature is below the set point.
4. REQ2: The running motor shall stop if it overheats

Could this scenario damage the motor, if repeated often enough? These questions are raised by the fact that users can read the requirements in the order of execution, versus reading them in the order they were documented in, as would be the case with a traditional validation approach, such as inspection.

V. Related Work

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 [8], [4], [10], [11], [12]. Each of these techniques has its inherent strengths and weaknesses over others, so choosing a requirements validation technique should take into account the type of systems being built. Since our target is embedded systems, we decided to pursue a model-based requirements validation approach that provides a systematic way to validate functional requirements and is especially suitable for the logical analysis of behaviors of embedded system requirements [21]. Modeling has been accepted as a fundamental activity throughout requirements engineering processes [2, 4, 5]. Typically the models are "virtual" in the sense that they exist in the computer, as mathematical or logical representations of the requirements.

Some of the "virtual" models that have been used with embedded systems include Petri Nets [14], Message Sequence Charts [15], and model checking [16]. However, these methods tend to limit stakeholders' involvement, due to the high amount of technical on-boarding, and they are not necessarily requirements centric. Also, building models often requires NL translation and this translation process can be problematic due to the inherent incompleteness and ambiguities of NL [1, 3]. To address this problem, many researchers have proposed various modeling techniques including automated NL translation approaches [4, 5, 6, 7]. Automation can certainly reduce human errors and improve the translation process, but complete automation of this process is not possible because often NL requirements can be interpreted in multiple ways and thus human judgment is inevitable to lead correct/sensible interpretations.

We avoided the problems with automated translation, by using a Human-based translation technique that allows for stakeholders' involvement. This also adds another opportunity for stakeholders to essentially review the requirements, first-hand, during translation. At the same time, the VRP uses some automated reasoning to help catch errors during the translation process. Toward this aim, we are constantly trying to improve the reasoning engine.

An alternative to using virtual models is the use of physical prototypes, whose strength is in allowing stakeholders to interact with a partial representation of the system to be. Physical prototypes can be an effective means of getting stakeholders' feedback [13]. However, prototyping an embedded system is often done for the sake of validating hardware and software, and can occur considerably downstream from the requirements phase.

A Virtual Prototype could provide the benefits of prototyping early in the development phase, preferably at or near the requirements validation. A virtual prototype can also provide the flexibility to accommodate changing requirements. Some work has been done in this area [18], such as SCORES [17].

Overall, we believe that the area of using virtual prototypes for requirements validation is very promising. To this end, we centered our efforts around a virtual prototype as the means of modeling requirements for user-driven validation, particularly as performed by non-technical stakeholders.

VI. Conclusion and Future work

We have looked at a technique for validating requirements of embedded systems that virtually mimics the validation potential gained from a physical prototype. Our goal is to increase both technical and non-technical stakeholders' participation in the requirement validation of

embedded systems. In achieving this goal we have developed a three phase process that enables non-technical stakeholders to create a virtual prototype of the requirements, make corrections to that prototype, and evaluate the prototype via stakeholder interaction. The approach presented in this paper makes use of the following techniques and concepts:

- The concept of prototyping requirements as a Virtual Requirement Prototype (VRP).
- Transforming NL requirements into a VRP, while maintaining a direct one-to-one trace between each requirement and a building block of the VRP.
- A Graph representation of how requirements interrelate and interact with one another.
- The use of an adjacency matrix as a convenient user-interface to a VRP, allowing users to quickly examine and modify an inter STD and RG model.
- The generation of Requirement Scenarios as the means of analyzing the VRP's behaviors.

Two areas that will be developed further, are enhancing the method's ability to model embedded systems, and enhancing the NL to VRP transformation process to better accommodate the representation of temporal characteristics, for use with hard real-time systems. We will be also improving the reasoning engine that assists stakeholders, as in having the tool trigger a user alert whenever users enter erroneous data during the prototype's creation. Finally, the tool will be tried on several sets of reactive requirements that have been already been obtained by us through various sources.

ACKNOWLEDGMENT

This work was supported in part by NSF under Awards CNS-0855106 and CCF-1050343 to North Dakota State University.

VII. REFERENCES

[1] D. Berry, Ambiguity in natural language requirements documents, Lecture Notes in Computer Science, extended abstract, 2008.

[2] B. Nuseibeh and S. Easterbrook, Requirements engineering: A roadmap, International Conference on Software Engineering, pp. 35-41, 2000.

[3] D. Gause, User DRIVEN Design-The luxury that has become a necessity, International Conference on Requirements Engineering, Tutorial T7, 2000.

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

[5] E. Letier. J. Kramer, J. Magee, and S. Uchitel, Monitoring and control in scenario-based requirements Analysis, International Conference on Software Engineering, pp. 382-391, 2005.

[6] R. Alur, K. Etessami, and M. Yannakakis, Inference of message sequence charts, International Conference on Software Engineering,, pp. 304-313, 2000.

[7] D. Deeptimahanti and M. Babar, An automated tool for generating UML models from natural language requirements, Automated Software Engineering, pp. 680-682, 2009.

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

[9] I. Sommerville, Software Engineering, 8th Edition, Addison-Wesley, 2007

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

[11] 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, pp. 382-391, 2005.

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

[13] R.V. Buskirk and B.W. Moroney, Extending prototyping, IBM Systems Journal, 42(4), pp. 613-623, 2003.

[14] L.A. Cortés, P. Eles, Z. Peng, Verification of embedded systems using a petri net based representation, Proc. International Symposium on System Synthesis, pp. 149-155, 2000.

[15] L. Kof. Scenarios: Identifying missing objects and actions by means of computational linguistics. In 15th IEEE International Requirements Engineering Conference, pp. 121–130, 2007.

[16] R. Alur, C. Courcoubetis, and D. Dill, Model checking for real-time systems, Proc. Symposium on logic in computer science, pp. 414-425, 1990.

[17] A. Homrighausen, H. Six, and M. Winter, Round-trip prototyping for the validation of requirements specifications, Conference Draft for Requirements Engineering Foundation for Software Quality, 2001

[18] A. Ravid and D. Berry: A method for extracting and stating software requirements that a user interface prototype contains, Requirements Eng, Springer-Verlog, pp. 225-241, 2000

[19] D. Zowghi, and V. Gervasi, On the interplay between consistency, completeness, and correctness in requirements evolution, Information and Software Technology 45(14): 993-1009, November 2003.

[20] P. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goosens, Embedded software in real-time signal processing systems: Application and Architecture Trends, Proceedings of the IEEE, vol. 85, pp. 419–435, March 1997.

[21] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli, Design of embedded systems: Formal Models, Validation, and Synthesis, Proceedings of the IEEE, vol. 85, pp. 366–390, March 1997.

[22] D. Aceituna, H. Do, and S. Lee, SQ2E: An approach to requirements validation with scenario Question, Proceedings of the 17th Asia-Pacific Software Engineering Conference pp. 33-42, 2010.

[23] D. Aceituna, H. Do, and S. Lee, A human interactive approach to building requirements models, International Symposium on Software Reliability Engineering, fast abstract, 2010.

[24] F. Schneider, S.M. Easterbrook, J.R. Callahan and G.J. Holzmann, Validating Requirements for Fault Tolerant Systems using Model Checking, Third IEEE Conference on Requirements Engineering, pg. 4-13, 1998.