

# Architecture-Based Semantic Evolution: A Study of Remotely Controlled Embedded Systems

Lawrence Chung  
Department of Computer Science  
University of Texas, Dallas  
Richardson, TX  
chung@utdallas.edu

Nary Subramanian  
Applied Technology Division  
Anritsu Company  
Richardson, TX  
narayanan.subramanian@anritsu.com

## Abstract

*Evolution of a software system is a natural process. In many systems, evolution usually takes place during the maintenance phase of their lifecycles. Those systems that have reached their limit in evolution have usually reached their end of useful life and may have to be replaced. Their life expectancy can be made greater if evolution occurs during the working phase of their lifecycles. Such systems need to be designed to evolve, i.e., adaptable. Semantically adaptable systems are of particular interest to industry as such systems adapt themselves to environmental change with little or no intervention from their developers. Research in embedded systems is now becoming widespread but developing semantically adaptable embedded systems presents challenges of its own. Embedded systems usually have a restricted hardware configuration, hence techniques developed for other types of systems cannot be directly applied to embedded systems. This paper develops concepts and techniques for semantic adaptation of embedded systems, using remotely controlled embedded systems as an application. In this domain, an embedded system is connected to an external controller via a communication link such as ethernet, serial, radio frequency, etc., and receives commands from, and sends responses to, the external controller. Techniques for semantic evolution in this application domain give a glimpse of the complexity involved in tackling the problem of semantic evolution in embedded systems. The techniques developed in this paper are validated by applying them in a real embedded system - a test instrument used for testing cell phones.*

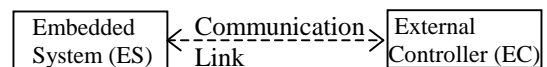
## 1. Introduction

Evolution of a software system is a natural process. In many systems, evolution takes place during the maintenance phase of their lifecycles. Those systems that have reached their limit in evolution have usually reached their end of useful life and may need replacement. Their

life expectancy can be made greater if evolution occurs during the working phase of their lifecycles. Such systems need to be designed to evolve, i.e., adaptable.

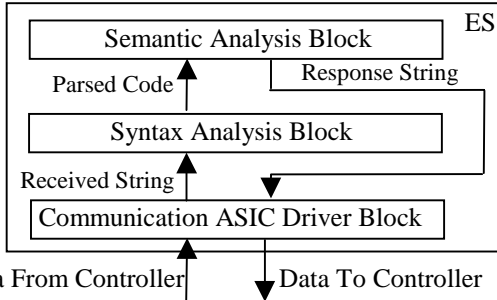
Research in embedded systems is now becoming widespread [1] but developing semantically adaptable embedded systems presents challenges of its own. Embedded systems are usually hardware-constrained systems running dedicated software [2]. For this reason, techniques for dealing with evolution that are developed for non-embedded systems [4] cannot be directly applied to embedded systems. For example, maintaining component libraries is usually ruled-out since embedded systems by and large do not have enough memory for storing the libraries.

In this paper, we develop concepts and techniques for semantic evolution of embedded systems, whose behavior change dynamically. For the purpose of illustration, we consider, as in Figure 1, the domain of remotely controlled embedded systems (RCES). The embedded system (ES) receives commands from, and responds to, an external controller (EC). The communication link between ES and EC could be any physical medium – ethernet, serial, radio frequency, etc. Figure 2 gives the functional blocks in a typical RCES. In this figure, the Communication ASIC (Application-Specific Integrated Circuit) Driver Block handles the hardware signals and the protocol associated with the physical interface. Usually such interfaces are connected to an ASIC block, which receives ASCII strings from the physical interface as commands sent by EC to ES, and sends them to the Syntax Analysis Block. The Syntax Analysis Block analyses the strings, and if syntactically correct, parses the strings and sends the parsed code to the Semantic Analysis Block, which takes appropriate actions for the input string (i.e., the command).



**Figure 1. Application domain for the problem**

The key advantage of this domain lies in simplicity. This domain lets scripts be executed automatically on EC,



**Figure 2. Generic architecture for embedded system**

and ES will execute all the commands of the scripts. This means of control by the remote controller also has other advantages [3, 11, 12].

The different techniques that we develop for tackling this problem of semantic evolution in embedded systems will lead to different architectural solutions for the problem. Software architecture consists of among other things, components, connections and constraints [5,6]. Thus architecture can be adaptable along any one, or more, of the three basic constituents. In this paper, we will develop different techniques for semantic adaptation and consider the effect of these techniques, but due to space limitations, on only one constituent of software architecture, viz., the component (considerations on other constituents can be found in [15]). All the architectures developed in this paper will be in a layered style.

Once software architectures have been developed, then comes the problem of finding the most efficient technique(s). We use the NFR Framework [7, 16], where *NFR* stands for non-functional requirements, for developing and comparing the various architectures. Using this framework, we determine the relative effectiveness of the different techniques.

Our discussion of the different techniques assumes the use of object-oriented technology. However, this does not preclude the use of these techniques in non-object-oriented environments. Many of the diagrams in this paper are depicted in UML [8], although any other notation with a similar modeling power can also be used. Also in the architectures the  $\Rightarrow$  has been used to indicate message passing between the layers of the architecture in the direction of the arrow.

Section 2 develops the concepts for semantic evolution. Section 3 discusses the application of the NFR Framework to this problem. Section 4 discusses the techniques for semantic evolution in the embedded system taken up for case study – the RCES. Section 5 validates the different techniques by implementing the designs in a commercial ES, and Section 6 gives the conclusion.

## 2. Semantic Evolution

Semantic evolution is a form of adaptation. Before we develop concepts for semantic evolution, we give a set of basic definitions for adaptation.

### 2.1 Adaptation Definition

Adaptation means change in the system to accommodate change in its environment. More specifically, adaptation of a software system ( $S$ ) is caused by change ( $\delta_E$ ) from an old environment ( $E$ ) to a new environment ( $E'$ ), and results in a new system ( $S'$ ) that ideally meets the needs of its new environment ( $E'$ ). Formally, adaptation can be viewed as a function:

$$\text{Adaptation: } E \times E' \times S \rightarrow S', \text{ where } \text{meet}(S', \text{need}(E')).$$

A system is *adaptable* if an adaptation function exists. *Adaptability* then refers to the ability of the system to make adaptation.

Adaptation involves three tasks:

1. ability to recognize  $\delta_E$ .
2. ability to determine the change  $\delta_S$  to be made to the system  $S$  according to  $\delta_E$ .
3. ability to effect the change in order to generate the new system  $S'$ .

The *meet* function above involves the two tasks of validation and verification, which confirm that the changed system ( $S'$ ) indeed meets the needs of the changed environment ( $E'$ ). The predicate *meet* is intended to take the notion of goal satisficing of the NFR framework, which assumes that development decisions usually contribute only partially (or against) a particular goal, rarely “accomplishing” or “satisfying” goals in a clear-cut sense. Consequently generated software is expected to satisfy NFRs within acceptable limits, rather than absolutely.

### 2.2 Semantic Evolution

In order to appreciate the definitions and symbols in this section, we again take the example of a cell phone. Let us assume that the cell phone can work in two standards – GSM and CDMA (so-called dual-mode phones) [13,14]. The inputs that it receives from the user are most likely the same for the two standards. That is, the input spaces are identical for the two standards. However, the behavior of the cell phone could be different for the two standards. For example, the phone connections could be made faster for one standard than for the other. Thus intuitively, the behavior could be related to non-functional aspects of the system, while inputs and outputs are related to the functional aspects of the system. Other behavior

related aspects include security, usability and throughput. After adaptation, however, the input spaces could change.

*Change* in the behavior ( $\delta_B$ ) is related to the behavior (B) of the system before and after adaptation when the input space (I) is the same. The change in the output space ( $\delta_O$ ) of the software system, and the change in the behavior ( $\delta_B$ ) are defined below.

$$\begin{aligned} [I \times S \rightarrow O \wedge I \times S' \rightarrow O'] &\rightarrow [\delta_O = O' - O]. \\ [I \times S \rightarrow B \wedge I \times S' \rightarrow B'] &\rightarrow [\delta_B = B' - B]. \end{aligned}$$

A software system evolves semantically (or *adapts* semantically) if

$$\text{for } \delta_E \neq 0, \delta_B = 0 \wedge \delta_O = 0.$$

That is, the system does not change its behavior even though the environment has changed. However, since this may not be possible to achieve all the time, using the concept of satisficing of the NFR framework, the following definition of semantic evolution will also be acceptable: for a semantically adaptable system one or more of the following holds true when  $\delta_E \neq 0$ :

1.  $\delta_B = 0$  and  $\delta_O = 0$ .
2.  $\delta_B \neq 0$  but  $\delta_O = 0$ .
3.  $\delta_B \neq 0$  but  $\delta_O \sim 0$ .

Equation 1 above states that the behavior of the system before and after semantic adaptation remains the same. Equation 2 above states that for a semantically adaptable system, the output space before and after adaptation remains the same. Equation 3 says that some difference in the output space is acceptable as long as the outputs are identical for key inputs (what is “key” depends on the particular application).

In this paper techniques for semantically adaptable systems conforming to equation 3 above will be developed. The input space for the domain of interest in this paper is the set of commands sent to the ES, while the output space is the set of responses from the ES. NFR ( $\delta_B = 0$ ) is satisfied if:

1. the commands suited to the adapted environment are accepted.
2. the time taken to execute/respond to the commands are less than 20ms for the RCES.

Likewise, the NFR ( $\delta_O = 0$ ) means the following: the responses to a command are the same before and after adaptation.

### 3. The NFR Framework

The NFR Framework [7,16] requires the following interleaving tasks, which are iterative:

1. Develop the NFR goals and their decomposition.
2. Develop architectural alternatives.
3. Develop design tradeoffs and rationale.
4. Develop goal criticalities.
5. Evaluation and Selection.

In the NFR Framework, each NFR is called an NFR softgoal (depicted by a cloud), while each design component is called a design softgoal (depicted by a dark cloud). The graph that the NFR Framework creates by the application of the five steps above is called the softgoal interdependency graph or SIG.

Each softgoal has a name following the convention

$$Type[Topic1, Topic2, \dots],$$

where *Type* is a non-functional aspect (e.g., adaptability) and *Topic* is a system to which the *Type* applies (e.g., RCES), and the refinement can take place along the *Type* or the *Topic*.

#### 3.1 Develop NFR Goals and their Decomposition

In this step, the semantic adaptation NFR is decomposed into its constituent NFRs. This decomposition will allow us to understand what semantic adaptation is all about and also to ensure that the designs that we come up with will be able to meet the requirements of the various sub-NFRs of the semantic adaptation NFR (see Figure 3).

The decomposition given in Figure 3 reads from the top. At the top are the three high level softgoals – Adaptation[RCES], Extensibility[RCES] and Speed[RCES]. Adaptation for RCES can be of different types – Syntactic Adaptation[RCES], Semantic Adaptation [RCES], Contextual Adaptation [RCES] and Quality Adaptation[RCES]. That we are concerned with one or more of these decompositions is indicated by a double arc, which stands for OR-decomposition. Quality Adaptation of RCES involves adaptation of NFRs for the system. As mentioned in Section 2, we are interested in the qualities of behavior and output, and there could be other qualities (or NFRs) as well. This consideration results in the two OR-decompositions of the softgoal Quality Adaptation[RCES]. Since in this paper we are concerned mainly with semantic adaptation, the softgoal Semantic Adaptation[RCES] is further OR-refined in terms of the three functions of RCES – Communication, Parsing (or Syntax Analysis) and Processing (or Semantic Analysis): Semantic Adaptation [Communication],

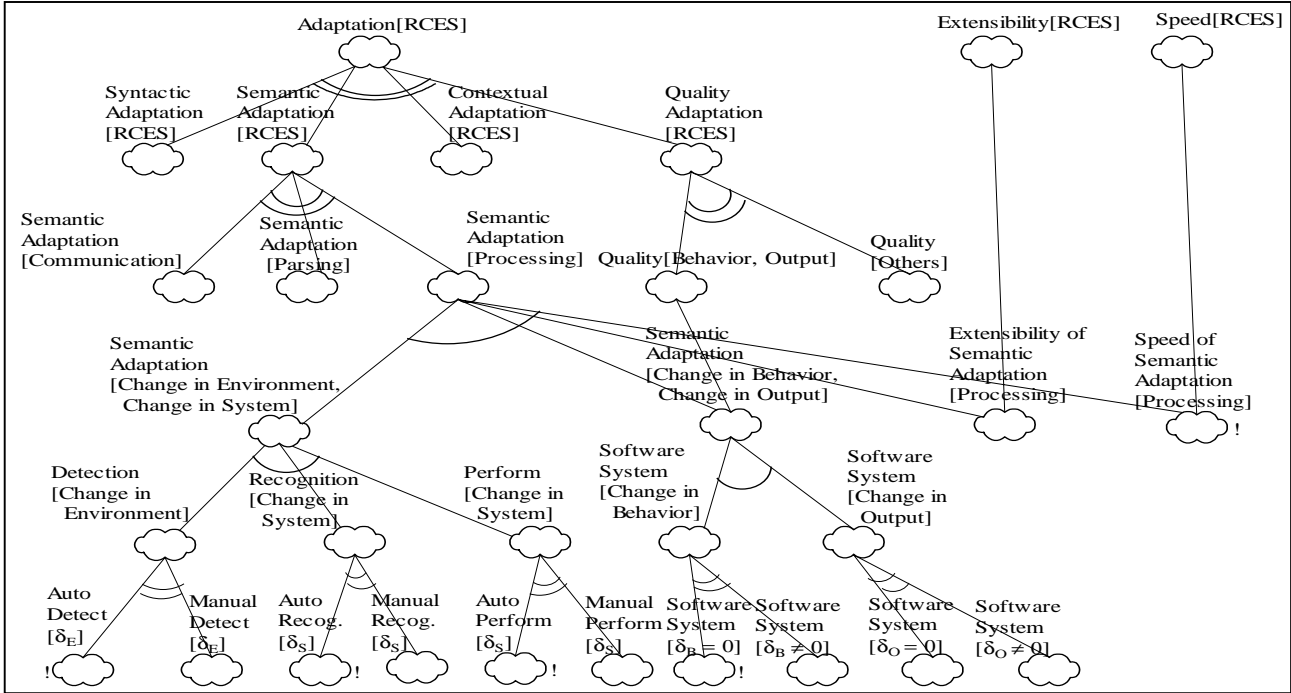


Figure 3. Softgoal hierarchy for semantic adaptation

Semantic Adaptation [Parsing] and Semantic Adaptation [Processing]. In this paper we focus on the semantic adaptation of the Processing component of RCES, so Semantic Adaptation[Processing] is AND-decomposed (indicated by the single arc) into four subgoals - Semantic Adaptation[Change in Environment, Change in System] (this follows from the definition of the requirements of adaptation in Section 2.1), Semantic Adaptation[Change in Behavior, Change in Output] (which again follows from the definition of semantic adaptation in Section 2.2), Extensibility of Semantic Adaptation[Processing] and Speed of Semantic Adaptation[Processing]. The AND-decomposition means that all the four softgoals have to be satisfied in order for the softgoal Semantic Adaptation[Processing] to be satisfied. Furthermore, it may be noted that three of the decomposed softgoals have two parents each – Semantic Adaptation[Change in Behavior, Change in Output] has two parents: Quality[Behavior, Output] and Semantic Adaptation [Processing], Extensibility of Semantic Adaptation[Processing] has Extensibility[RCES] and Semantic Adaptation[Processing] as parents, while Speed of Semantic Adaptation[Processing] has Speed[RCES] and Semantic Adaptation[Processing] as parents. For such multi-parent softgoals, satisficing of the softgoal satisfies both its parents.

Semantic Adaptation[Change in Environment, Change in System] is then AND-decomposed into Detection[Change in Environment], Recognition[Change in System] and Perform[Change in System], where the last

softgoal is for performing the change in the system. These decompositions again follow from the definitions in Section 2.1. The softgoal Semantic Adaptation[Change in Behavior, Change in Output] is further AND-decomposed into its constituents: Software System[Change in Behavior] and Software System[Change in Output].

Detection[Change in Environment] is OR-decomposed into the two ways whereby such a detection can take place- automatic and manual (we have used  $\delta_E$  for Change in Environment). The same is done for the softgoals Recognition[Change in System] and Perform[Change in System]. The softgoal Software System[Change in Behavior] is OR-decomposed into two softgoals- Software System[No Change in Behavior] and Software System[Change in Behavior] (where we have used  $\delta_B$  to indicate Change in Behavior), and the softgoal Software System[Change in Output] is also OR-decomposed in the same manner.

### 3.2 Develop Architectural Alternatives

Architectural alternatives are developed in Section 4.

### 3.3 Develop Design Tradeoffs and Rationale

As mentioned in Section 2.1, different architectural solutions satisfy various NFRs to different extent. We use the legend of Figure 4 in describing the different degrees of NFR satisficing. Then for each design softgoal

a rationale table is developed to indicate the extent to which that softgoal satisfies an NFR softgoal and why.

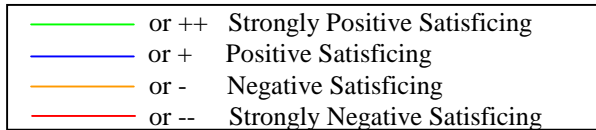


Figure 4. Symbols for different degrees of satisficing

### 3.4 Develop Goal Criticalities

For the particular application, different NFRs will have different criticalities. Criticalities are shown in the SIG using ‘!’ marks. In Figure 3, five NFRs are marked as critical:  $\delta_B = 0$ , Speed, Automatic  $\delta_E$  detection, Automatic  $\delta_S$  recognition, and Automatic  $\delta_s$ . The reason why these NFRs were chosen as critical is because of their relevance in practice. In the company where one of the authors works, fulfillment of these NFRs would give the greatest advantage for using semantic adaptation.

### 3.5 Evaluation and Selection

Here the SIG is constructed and the most suitable architecture for the application is selected. This step will be performed after implementation.

## 4. Techniques for Semantic Evolution

We have identified the following techniques for semantic adaptation in embedded systems:

1. Rework, Reload and Reboot (or 3Rs Technique)
2. Stored Data Technique
3. Rule Based Approach
4. Run-time Module Generation.

### 4.1 Rework, Reload and Reboot (3Rs) Technique

This is the technique that is currently widely used (see Figure 5). In this technique, for any change in environment, a new system that works in the new environment is developed (either from scratch or as a modification of the existing system – the rework phase) and is executed in the embedded system’s hardware (reload and reboot phase). This technique is efficient but very slow in terms of time for adaptation.

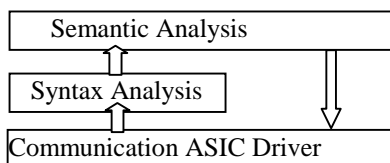


Figure 5. Architecture for the 3Rs technique

### 4.2 Stored Data Technique

In this technique, the data for possible adaptations are stored in ES in the beginning. At run-time, a state machine keeps track of current state of the system. Consider, for example, a cell phone measuring instrument that generates output signals for two different cell phone systems, say GSM and CDMA. Now if the instrument generates signals in response to an input command “MAKE CALL” (from EC), then depending on the state the instrument is currently in, the particular signals pertaining to that cell phone system will be generated. As in Figure 6, the transition between the two states GSM and CDMA occurs due to another input command “CHANGE SYSTEM”.

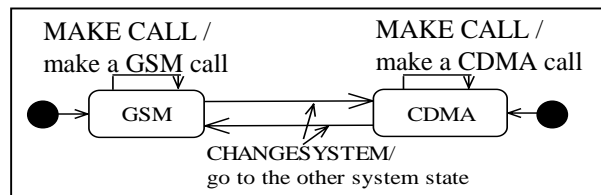


Figure 6. Statechart for the stored data technique

Thus in a system using this technique, all possible states are already captured in the state diagram for the software system. The software system can adapt to those new environments for which there are corresponding states in the system. There are two ways in which this adaptation to the new environment can occur – *manually* or *automatically*. In a manual setting, the change in environment is detected manually - an external user sends in a command like “CHANGE SYSTEM” and the system changes to the new state consistent with the new environment. Thus the detection of the environment change and need for a corresponding system change are both detected manually while the system change itself occurs automatically. In an automatic method, the system receives signals (continuing with the example of Figure 6) from a phone connected to it and, based on the signals received, the system changes (if necessary) its own internal state. Here the environment change detection, need for corresponding system change and the system change itself are all done automatically.

In an OO-system, if the class hierarchy is as shown in Figure 7 [13, 14], then each state may represent one of the leaf nodes in the hierarchy. In the class diagram of Figure 7, at the top of the hierarchy is the Wireless Protocol class from which are derived US Protocol and Non-US Protocol classes. Digital and Analog classes are derived from US Protocol class. IS-136 and CDMA (two digital standards) classes are then derived from the Digital class. AMPS is an analog standard class derived from Analog class. GSM and Japanese Cellular are two standards classes derived from Non-US Protocol class.

The architecture for a system using this technique is given in Figure 8. In this figure, the State Machine component controls the state that the system is in (the system can be in one of the pre-defined states 1,2,...,n). The State Machine component includes the State Checking and State Modification functions (or components). The response from the system also takes place via the State Machine component.

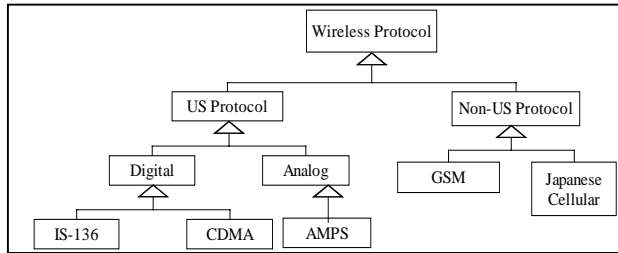


Figure 7. Class diagram for stored data technique

### 4.3 Rule-Based Approach

Just about any software system follows a set of rules. The rules could be either embedded in the executable code or could be data (or algorithms) that are separate from the code [10]. A system could adapt semantically by modification of one or more rules of this set. Figure 9 shows some examples of the rule-based approach. For the RCES, the use of this rule-based approach is pretty easy, as we can tie one or more commands to a rule. Thus, for Example 1 of Figure 9, if the command “CUTOFF\_LEVEL X” referred to the current rule, and if by sending the command “CUTOFF\_LEVEL Y” the

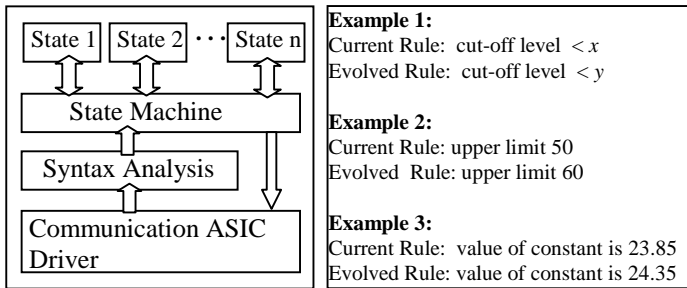


Figure 8. Architecture for the stored data technique

Figure 9. Examples for rule-based approach

current rule changes to the evolved rule, the needed evolution has been achieved.

Figure 10a gives the architecture for this approach. In this figure the Command Control component executes the inputs and generates responses, if necessary. All the rules are present in the Rules component. A command to change the rule will cause the Command Control component to call Rules Modification component to change one of the existing rules in the Rules component

and thus affect the behavior of the system in the future. Likewise, a command to execute a system function will cause the Rules Checking and Execution component to check if a rule exists for the system function and if there is then that rule is followed (else the default action takes place). Here adaptation takes place by changing the contents of the Rules Component.

### 4.4 Run-time Module Generation

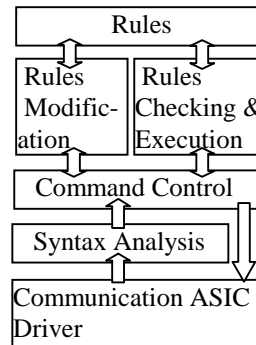


Figure 10a. Architecture for rule-based approach

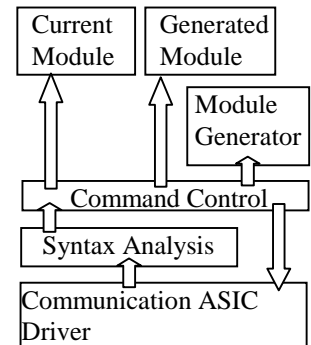


Figure 10b. Architecture for run-time MG technique

This is a very powerful technique that allows the system to change its behavior in many different ways. Here new modules are generated at run-time to enable the system to adapt to the change in environment. Thus, for example, if a software system currently displays all text in English and if the text language had to be changed to Japanese, then a new display module could be generated that displays all text in Japanese. This new module may be a peer of the existing English module. Again in this application domain, module generation (MG) may be done with the help of commands from outside. Figure 11 shows the class diagram for this technique.

In Figure 11, Display is a template class and its parameterized element is Language. In the original system, only one language display class called English Display was instantiated from the template class. Due to adaptation requirements, at run-time the Japanese Display class was instantiated to handle displays in Japanese.

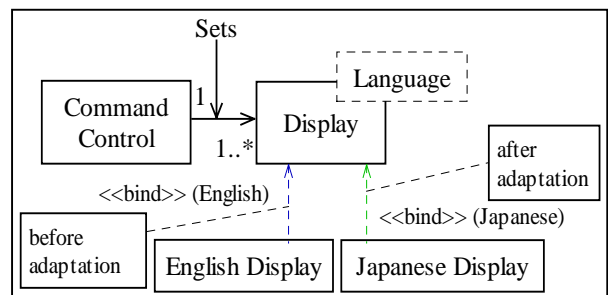


Figure 11. Class diagram for module generation

The generic architecture for this technique is given in Figure 10b. Here the Module Generator is called to create the Generated Module in response to an external command. The pre-determined external commands also give the limit to which new modules can be generated. If due to memory limitation no further new modules can be generated, the system informs the user of this problem and the new module will not be generated.

## 5. Validation of the Techniques

### 5.1 Validation Approach

In order to validate the techniques we took the following steps:

1. Implemented the architectures in an ES for a chosen problem.
2. Tested the implementation with the specific environment change for the problem chosen.
3. Ensured that the implementations adapted themselves for the environment change.
4. Checked to see how the various implementations fulfilled the requirements of the critical NFRs (marked by ‘!’ in the SIGs).
5. Timed the speed of adaptation (which is one of the critical NFRs).

The various techniques for semantic evolution mentioned in Section 4 were implemented in a test instrument that runs on a Motorola 68K processor, and has for external communication a IEEE488 port (the advantage with this interface is that accurate timing measurements are possible with the aid of a PC-based tool). In order to make the differences between the techniques clear, all the techniques are implemented for a pre-defined environment change. This will also let the times for adaptation indicate the relative speeds of adaptation for the different techniques. The time taken to execute in this application domain is the time taken to execute an input string.

### 5.2 Problem for Implementing Adaptation

The environment change example is taken from the user-interface domain, which we call the level-changing problem. This permits an easy appreciation of the problem and the solutions. However, the techniques are applicable to any other environment change as well. Let a test equipment for a cell phone be connected to a cell phone under test. The test equipment can generate signals that the cell phone receives and the test equipment can receive the signals from the cell phone as well. One of the tests [14] that the cell phone manufacturer needs to be perform

is to find out the level at which the cell phone drops a call. In such a test the cell phone connected to the test equipment is brought into a conversation state (that is, the cell phone makes a call with the test equipment) and then the test equipment drops its level step-by-step until the cell phone drops the call. This step-by-step reduction could be in steps of 1dB (the levels are usually expressed in dBm and dB which are log values of the corresponding power values). Thus the software in the test equipment would accept level settings of integer values (in dBm units). However, let us suppose that for a future generation of cell phones a finer step size is required – say 0.5dB steps. Then the test equipment should adapt to this change in level settings – from integer level settings to floating point level settings. This environment change will be used to illustrate the techniques discussed in Section 4.

Let there be a parameter that takes different integer values. Let the value of the parameter be set by the input command “`PARA_VALUE n`”, where  $n$  is the integer value that the parameter takes. The effect of this command on the Syntax Analysis Block of Figure 2 is to generate a code corresponding to the command “`PARA_VALUE`”, say  $c$ , and send  $c$  and  $n$  to the Semantic Analysis Block. The Semantic Analysis Block, in response to code  $c$ , sets the corresponding parameter,  $parameter\_c$  (where  $parameter\_c$  could be the output level as described above). Let the  $parameter\_c$  be an object. Then the Semantic Analysis Block may use a function such as  $parameter\_c.Set(n)$  to set the value of  $parameter\_c$ . Another function that may be required of the object  $parameter\_c$  is the  $Get()$  function which lets EC get the value of the parameter using a command like “`PARA_VALUE?`”. This is depicted in Figure 12.

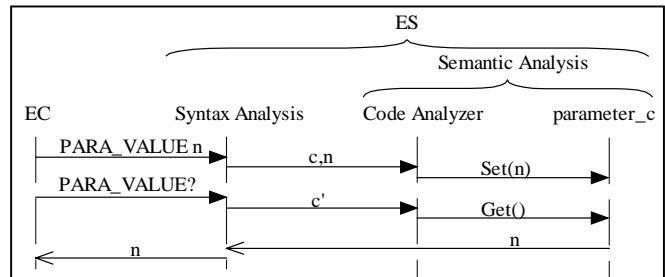


Figure 12. Sequence diagram for level-changing problem

The environment change occurs by suddenly switching over to floating point values. That is, the command, “`PARA_VALUE f`” is sent, where  $f$  is a floating point (FP) value that the parameter should take. In response to this command the Syntax Analysis Block will send the parameters  $c$  and  $f$  to the Semantic Analysis Block. The requirement is that the ES (or the Semantic Analysis Block) should accept this FP value. The different techniques for semantic adaptation react to this

environment change differently and the differences will be discussed in the implementation.

### 5.3 Implementation Using the 3Rs Technique

In the 3Rs technique, before the environment change, the code is changed to adapt the system to the new environment. Thus a new object called *parameter\_f* would be created that stores FP values. Then the Semantic Analysis Block, in response to code *c* from the Syntax Analysis Block will call the Set( ) function of *parameter\_f*.

As can be expected, for each change in environment, the code has to be changed, the new executable loaded into ES's memory and the ES is re-started. This is a slow process of adaptation, with adaptation time in the order of minutes. However, this is a sure method in that the system can be made to suit any change of environment.

This technique does not meet the requirements of four critical NFRs, viz., Automatic  $\delta E$  detection, Automatic  $\delta S$  recognition, Automatic  $\delta S$  and the Speed of Semantic Adaptation. However, it does meet the fifth NFR, viz., Software System [ $\delta B = 0$ ], as it fulfils both the conditions for satisfying the NFR (mentioned in Section 2.2) – this was confirmed by means of timing measurement and acceptance of commands as shown in Table 1 (where ET is execution time and RT is response time).

**Table 1.  $\delta_B$  for 3Rs technique implementation**

State	Command	Performance
Before Modification (accepts integer values)	PARA_VALUE 100	Command Accepted
Before Modification (accepts integer values)	PARA_VALUE?	300 $\mu$ s ET, 9 ms RT
After Modification (accepts FP values)	PARA_VALUE 10.5	Command Accepted
After Modification (accepts FP values)	PARA_VALUE?	1.926ms ET, 11ms RT

### 5.4 Implementation of the Stored Data Technique

In the implementation of this technique, both the objects *parameter\_c* and *parameter\_f*, corresponding to the integer parameter and FP parameter, respectively are already present in the system. If the parameter sent is a FP parameter (detected by the '.') then the value of *parameter\_f* is set, else the value of *parameter\_c* is set. A flag is maintained for the last parameter value type, so that in response to PARA\_VALUE? the correct object's Get( ) function is called.

As can be expected, the adaptation using this technique is very fast (discussed below); however, all the different states will have to be present in the code, which means that the environment changes will have to be foreseen; this also means more memory requirement.

As mentioned earlier, this technique can detect environment change automatically or manually. Also this technique recognizes the need for the system change automatically and performs the system change automatically. This technique also fulfills the NFR of no change in behavior and its results with respect to this NFR are similar to those of the 3R's technique. Adaptation times for this technique were in microseconds for integer parameters and in milliseconds for FP parameters.

### 5.5 Implementation of the Rule-Based Approach

In this technique, a rule is set to indicate that the next parameter is a FP parameter. This may be set by a command such as "NEXT\_PARA FLOAT". This command sets a rule (a variable) to indicate that the next parameter is a FP parameter. When the Syntax Analysis Block sends the code *c*, the Semantic Analysis Block first checks the rule corresponding to this code and then calls the Set( ) function of the correct object. However, we implemented this technique with several different rules. In the implementation we controlled the different rules included and measured the time taken to execute each of the rules. As can be expected, the time taken to check the rules makes this technique slightly slower than the Stored Data technique. Regarding change in behavior, this technique also meets the conditions like the 3Rs Technique does. Thus this technique meets all the critical NFRs except that of Automatic  $\delta_E$  detection. The adaptation times for this technique were in microseconds for integer parameters and in milliseconds for FP parameters. This technique is also a fast adaptation technique. However, the rules cannot be developed during run-time; all the possible rules will have to be implemented beforehand.

### 5.6 Implementation of the Run-time Module Generation Technique

The currently available techniques do not allow an easy implementation of this technique. Firstly, the run-time generation of such objects will require a compiler to be present in the ES that will dynamically compile the code – called just-in-time compilation ([9] discusses this concept for the Java language). However, such compilers are not available for all embedded operating systems. Also, these compilers will occupy memory. Furthermore, dynamic compilation loses the advantages of performance optimizations available with static compilation. However, if these disadvantages could be overcome, then this technique would be a powerful technique.

In order to implement this technique we used a different approach – we modified the binary executable code at run-time. The size of an object like *parameter\_c* is

about 25 bytes. We also knew the addresses of the different functions of the objects from the map file. What we did is to change the binary code of the object at run-time. Thus if the Set( ) function for *parameter\_c* object started at memory location 0x123456, we overwrote the bytes of this function with the Set( ) function for the *parameter\_f* object. This required overwriting 16 bytes of memory. We also had to overwrite some bytes prior to setting the value (such as replacing *atoi* by *atof*) and this required 46 bytes of memory to be overwritten. Thus when the FP value was received, it was set correctly. Likewise, for reading the set value, we had to change additional bytes of memory (30 bytes) and we were able to retrieve the set values correctly. In order to change memory we used a special command “MEMWRITE *addr*, *new\_value*” where *addr* is the address to be overwritten and the *new\_value* is the new byte that should be written to *addr*. EC used a script to overwrite memory and change the system. This will take a longer time (2.073s to execute MEMWRITE for 92 bytes of memory) than the rule-based approach, but is fast and versatile. However, a deep knowledge of the software addresses will be required. Also overwriting wrong addresses could have disastrous results. Moreover, there is one major difference between what we did and what is required: when we overwrote the integer function, then we could no longer use the integer functionality until we restored the original data again – while in a true module generation technique both types of data will be available for access at the same time.

This technique meets all the critical NFRs except automatic  $\delta_E$  detection. When we tested the implementation the behavior did not change. Also the adaptation time for this technique (in our implementation, the adaptation time was 2.073 seconds) is much faster than the 3Rs Technique.

### 5.7 SIG Based Comparison of the Techniques

In this section we perform the last phase of the NFR Framework – that of evaluation and selection of architectures (Section 3.5). We now have all the necessary information to draw the SIGs. A partial design tradeoffs and rationale table for the 3Rs Technique is given in Table 2.

Based on the complete version of Table 2, the SIG for the 3Rs Technique can be drawn and this SIG is given in Figure 13. Similarly, more interesting SIGs can be drawn for the other techniques.

Figure 14 gives the combined SIG for all the techniques put together (in the interests of clarity the lines from the techniques have not been extended all the way to the softgoals they connect). The lines emanating from the different techniques connect to the corresponding leaf

**Table 2. Partial rationale table for 3Rs technique**

Softgoal	Degree of Satisficing	Rationale
$\delta_B = 0$	+	Ensured during modification
$\delta_O = 0$	++	In this application domain repeatability is very high
$\delta_O \neq 0$	-	Domain characteristics
automatic detection [ $\delta_E$ ]	--	No such capacity exists
manual detection [ $\delta_E$ ]	++	By design
Extensibility	++	Can be modified to any extent
Speed	-	By validation – Section 5.3

softgoals in the upper part of the SIG, from left to right. An architecture having the maximum green lines emanating from it is the better one – however, there is a rider to this rule – the green lines should also connect to the most number of the softgoals determined to be critical. Thus while the Stored Data Technique and the Run-time Module Generation Technique both have six green lines emanating from it, three of the green lines of the Stored Data Technique fulfill critical NFRs while only two of the green lines from the Run-time Module Generation Technique fulfill critical NFRs. Thus by the NFR Framework it should be concluded that the Stored Data Technique is more appropriate with respect to the softgoal decomposition used.

## 6. Conclusion

In this paper we have attempted to address the important problem of semantic adaptation (or evolution) in embedded systems. Semantic Adaptation is a very important NFR for embedded systems and techniques to satisfy this NFR will be very useful to the industry. This paper is among the first investigations into adaptability in embedded systems on an architectural basis.

We first defined adaptation and then extended the definition to semantic adaptation. We then show how to rationalize the development of adaptable software architecture using the NFR Framework [7, 16] and we provide not only the methodology to perform this rationalization but also a decomposition of the adaptation NFR. We then developed the different techniques for semantic evolution in embedded systems in the domain of remotely controlled embedded systems, which we took upon as a case study. The techniques include:

1. 3Rs Technique
2. Stored Data Technique
3. Rule Based Approach
4. Run-time Module Generation Technique

In this application domain, the embedded system receives commands from and sends responses to, an external controller over a communication link such as ethernet, serial, etc. Each of the techniques produced

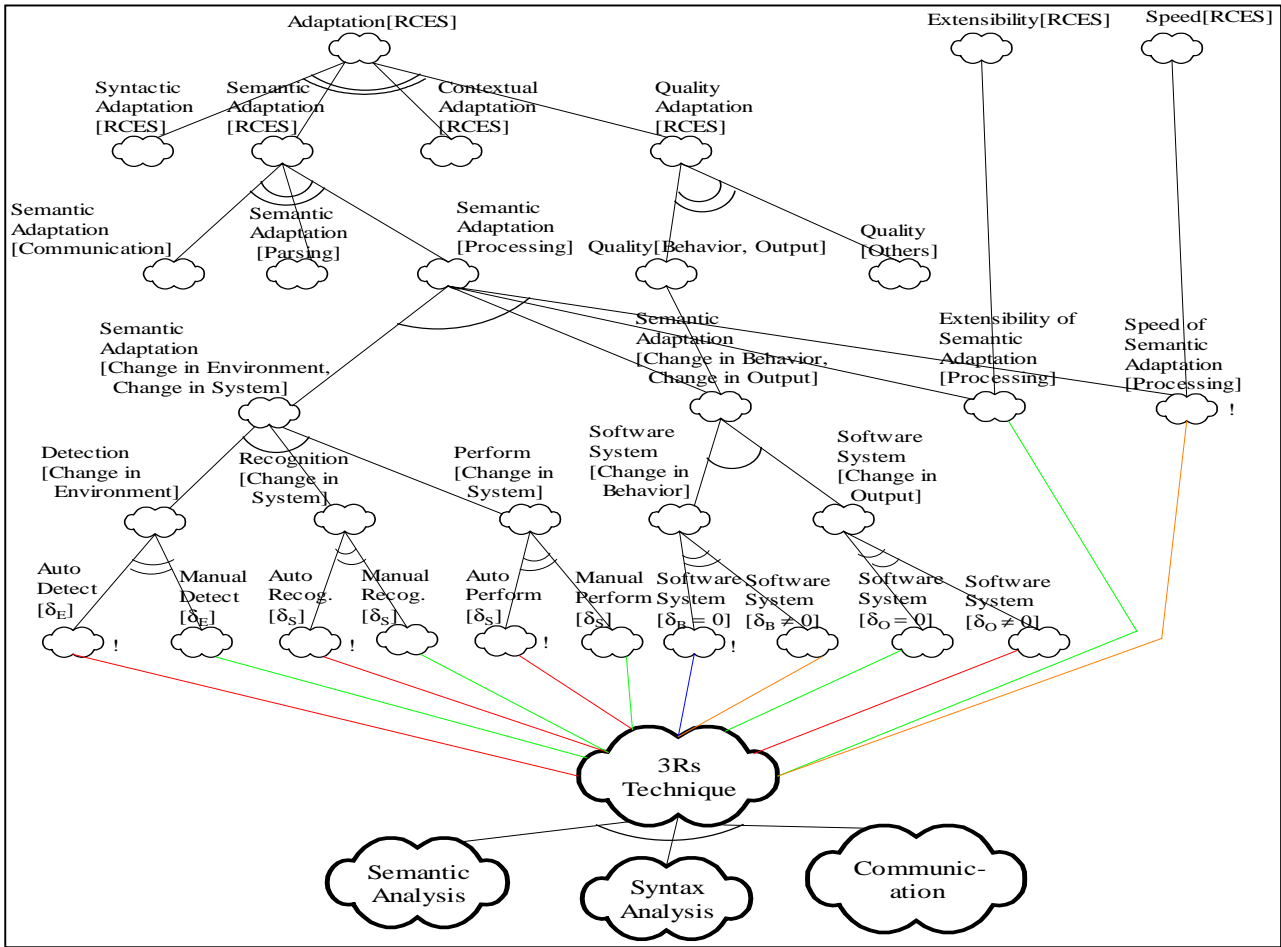


Figure 13. SIG for the 3Rs technique

different architectures for semantic adaptation for this application. The codes that ensued from the different architectures were implemented in a real embedded system (a test equipment connected to a PC by an IEEE488 cable) and the techniques were validated. The time for adaptation for the different techniques was also measured. Then the various architectures were compared using the NFR Framework. As can be expected different architectures scored differently for the given decomposition of the semantic adaptation NFR. It is our opinion that by using this application domain as a sub-domain in other applications, these techniques can easily be extended to these other applications [11,12].

There are several areas for further research. The techniques discussed here are by no means exhaustive – more work needs to be done to find better techniques suited for embedded systems. Also extensible techniques such as run-time module generation techniques have to be studied further. Extension of concepts like just-in-time compilation to embedded systems will have to be considered. Also of interest will be the development of a

better mathematical model for semantic adaptation. We understand that the techniques discussed in this paper are only a beginning to achieving the goal of fully automatic semantic adaptation in embedded systems.

## Acknowledgements

The authors would like to thank the colleagues of one of the authors in Anritsu Company, particularly Mr. Pete Johnson, for their valuable suggestions and comments.

The authors would also like to thank the anonymous referees of our complete original paper for their valuable and constructive suggestions and comments.

## References

- [1] E. A. Lee, "What's Ahead for Embedded Software?", *Computer*, Sep. 2000, pp. 18 – 26.
- [2] P. A. Laplante, *Real-Time Systems Design and Analysis*, IEEE Press, Piscataway, New Jersey, 1992.

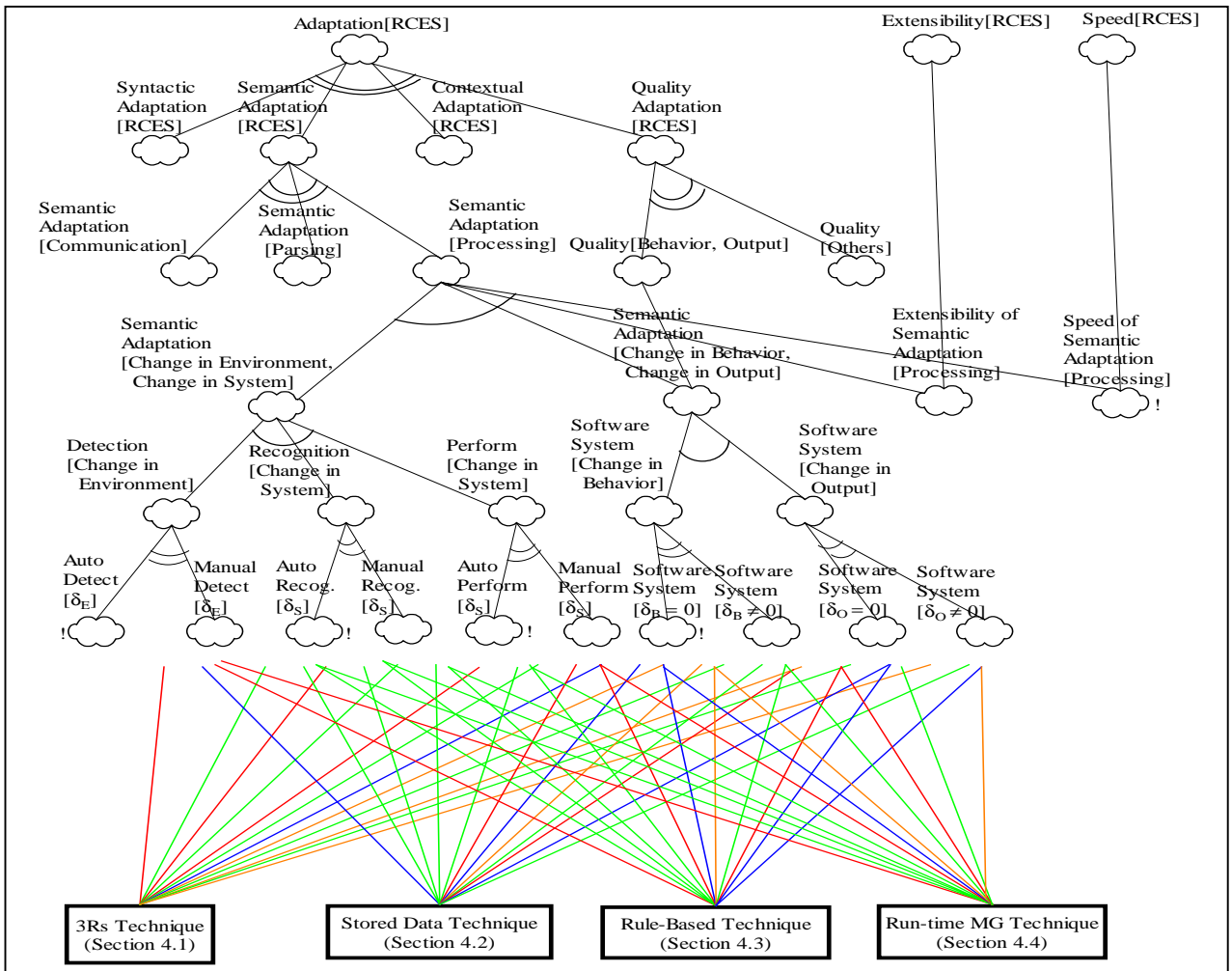


Figure 14. Combined SIG (partial) for all techniques

[3] N. Subramanian and L.Chung, "Architecture-Driven Embedded Systems Adaptation for Supporting Vocabulary Evolution", *ISPSE 2000*, Nov., 2000, Kanazawa, Japan, IEEE Computer Press, pp. 143 – 152.

[4] P. Oreizy, M. M. Gorlick, R. N. Taylor, D.Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software", *IEEE Intelligent Sys.*, May/June 1999, pp. 54 – 62.

[5] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[6] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, SEI Series in Software Engineering, Addison-Wesley, 1998.

[7] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, "Non-Functional Requirements in Software Engineering", Kluwer Academic Publishers, Boston, 2000.

[8] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[9] J. Hoskin, "Robust Embedded Java Devices Must Meet Special Requirements", *Embedded Systems Development*, 3(10), Oct. 2000, pp. 44 – 48.

[10] P. Johnson, "Rule-Based ELINT for Realtime Systems", *Texas Instruments Technical Journal*, 9(4), July-Aug. 1992, pp. 58 – 71.

[11] N. Subramanian, "A Novel Approach To System Design: Out-In Methodology", *Wireless Symposium/Portable by Design Conference*, San Jose, Feb. 2000.

[12] N. Subramanian and L. Chung, "Testable Embedded System Firmware Development: The Out-In Methodology", *Computer Standards & Interfaces Journal*, 22(5), Dec. 2000, pp. 337- 352 .

[13] K. Nakatsugawa, "Trends in Digital Mobile Communications and Related Measuring Instruments", *Anritsu Technical Review*, No. 17, Feb. 1996, pp. 48 – 55.

[14] A. Miceli, *Wireless Technician's Handbook*, Artech House, Boston, 2000.

[15] L. Chung and N. Subramanian, "Architecture-Based Semantic Evolution of Embedded Systems", Working Memo, 2001.

[16] J. Mylopoulos, L. Chung, S. S. Y. Liao, H. Wang, E. Yu, "Exploring Alternatives During Requirements Analysis", *IEEE Software*, Jan./Feb. 2001, pp. 2 – 6.