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 several 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 have to be replaced. However, there are several systems where evolution occurs during the working phase of their lifecycles. Such systems were designed to evolve or in other words, be adaptable. Semantically adaptable systems are of particular interest to industry as such systems adapt themselves to environment change with little or no intervention from their developing organization. Research in embedded systems is now becoming popular [1] and developing semantically adaptable embedded systems presents challenges of its own. Embedded systems usually have a restricted hardware configuration as well and several techniques applicable to "normal" systems cannot be directly transferred to embedded systems. This paper considers semantic evolution as applicable to embedded systems and develops the concepts and techniques for semantic adaptation in embedded systems. However, the field of embedded systems being vast, this paper concentrates on those embedded systems that can be remotely controlled (as opposed to remote controlled). In this application domain the embedded system is connected to an external controller by a communication link such as ethernet, serial, radio frequency, etc., and receives commands from (and sends responses to) the external controller via the communication link. 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 several 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 have to be replaced. However, there are several systems where evolution occurs during the working phase of their lifecycles. Such systems were designed to evolve or in other words, be adaptable.

Embedded systems are usually hardware-constrained systems running dedicated software [2]. Software running in the embedded systems is usually optimized for the underlying hardware and the OS used (if any). However, the software for the embedded system has properties just like any other software, and in particular, it evolves. Due to the constrained characteristics of embedded systems, several techniques for dealing with evolution that are applicable to non-embedded systems cannot be directly applied to embedded systems as well. For example, libraries of components are usually ruled-out since several embedded systems do not have enough memory for storing the libraries.

An example of embedded system is the cell phone. The hardware for the cell phone consists of the circuitry for receiving and transmitting radio signals, circuits for working on the electrical signals converted from radio signals, and a microprocessor for controlling the working of different pieces of hardware. The software is stored in a memory (such as FLASH or DRAM) and runs on the microprocessor. The software tells the microprocessor what action to take in different situations. Yet another embedded system is the base station used in a cell phone system. The base station sends signals to all the cell phones in its service area. An equipment that tests the cell phones in the factory floor emulates the base station and this equipment is yet another example of an embedded system.

Evolution in adaptable systems can occur at different levels of abstraction. At the lowest level, or Level 0, there is no evolution at all. The hardware and software configurations are fixed. The software system runs in the prescribed environment efficiently. For a different environment the software system fails. At a higher level, or Level 1, the software can tolerate some change in environment. [3] gives a problem at this level. At the next level, or Level 2, software can tolerate large changes in environment; in fact, software's behavior can also change. Techniques to develop software for this level (albeit, for non-embedded systems) are discussed in [4]. At the highest level, or Level 3, there can be both hardware and software changes in systems to adjust to virtually any change in environment. Large scale distributed systems attempt to reach Level 3 adaptation.

At higher levels of abstraction the system evolves dynamically. At these levels the behavior (or the semantics) of the system changes dynamically. While achieving dynamic evolution is difficult enough in normal systems, the difficulty becomes compounded when such dynamic evolution has to be enforced in embedded systems. An example of the requirement for such evolution can be found in cell phones – a cell phone that is used for one wireless standard (say GSM) may be required to work for two different wireless standards (say GSM and CDMA) [13, 14]. In such a case the makers of the cell phone could profit by making the software of the cell phone dynamically evolvable.

In this paper, we attempt to develop techniques for embedded systems to satisfy Level 2 adaptation. In order to validate the concepts and techniques that we develop, we will concentrate on one important problem at this level: semantic evolution in embedded systems. In order to understand this problem, we further restrict our attention to an application domain given in Figure 1 – the domain of remotely controlled embedded systems (as opposed to remote-controlled embedded systems). The embedded system (*ES*) receives commands from and responds to the commands from an external controller (*EC*). The communication link between the ES and EC could be any physical medium – ethernet, serial, radio frequency, etc. Figure 2 gives the functional blocks in a typical embedded system in this application domain. In this figure, the Communication ASIC Driver Block handles the hardware signals and the protocol associated with the physical interface. Usually such interfaces are connected to an ASIC (Application Specific Integrated Circuit). This block receives ASCII strings from the physical interface (these strings are the commands sent by the external controller to the embedded system) 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 the appropriate actions for the input string (the input command).

Figure 1. Application Domain for the Problem.

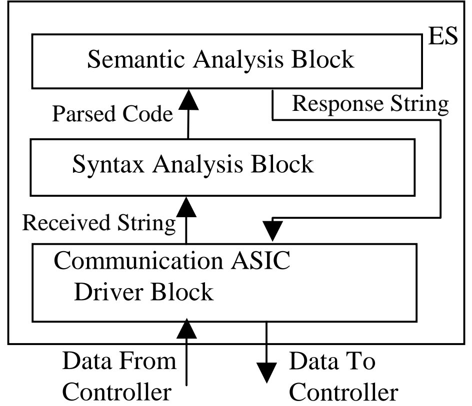

Figure 2. Functional Blocks in the Embedded System (ES) with their Interconnections.

The basic advantage of this domain is that commands can be sent from the external controller (which could be a PC) and the embedded system receives these commands over the communication link, parses the commands and takes actions based on these commands. This lets scripts be executed automatically on the external controller and the embedded system will execute all the commands of the scripts. This means of control by the remote controller also has other advantages [3, 11, 12]. The semantic adaptation in this application domain concerns itself with the adaptation of the Semantic Analysis Block of Figure 2 (though the adaptation of this block may require adaptation of the other blocks in that figure).

The different techniques that we develop for tackling this problem of semantic evolution in embedded systems will lead to different architectural solutions for this problem. We know [5,6] that software architecture consists of among others, components, connections and constraints. Thus architecture can be adaptable along any one of the three basic constituents – one, two or all three of components, connections and constraints can be adaptable. In this paper we will develop different techniques for semantic adaptation and consider the effect of those techniques on the three constituents of software architecture. All the architectures developed in this paper will be in the layered style.

Once software architectures have been developed, then comes the problem of finding the most efficient technique(s). In order to determine the best architectures(s) we use the NFR Framework [7, 15], where *NFR* stands for non-functional requirements, for comparing the various architectures that we come up with. Using this framework we are able to determine the relative effectiveness of the different techniques.

In the discussion of the techniques for semantic evolution in this paper it has been assumed that object-oriented technology has been used. However, this does not preclude the use of these techniques in non-object-oriented environments.

In this paper, the terms "semantic evolution" and "semantic adaptation" are used interchangeably. Many of the software diagrams in this paper use the notation borrowed from UML [8] although any other notation with a similar modeling power can also be used. Also in the architectures the ⇨ 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 systems taken up for case study – the remotely controlled embedded systems. Section 5 validates the different techniques by implementing the designs in a commercial embedded system, and Section 6 gives the conclusion. There are also two appendices – Appendix A gives the softgoal interdependency graphs while Appendix B gives the validation timings.

2. SEMANTIC EVOLUTION

Semantic evolution is a form of adaptation. Before we develop concepts for semantic evolution, we give the 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 (δ_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:

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

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 δ_E
2. ability to determine the change δ_S to be made to the system S according to δ_E
3. ability to effect the change in order to generate the new system *S'*.

These can be written as functions in the following way:

$$EnvChangeRecognition : E' - E \rightarrow \delta_E$$
$$SysChangeRecognition : \delta_E \; x \; S \rightarrow \delta_S$$
$$SysChange : \delta_S \; x \; S \rightarrow S', \text{ where } meet(S', need(E')).$$

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.

Figure 3 explains the relationship between the various symbols described above.

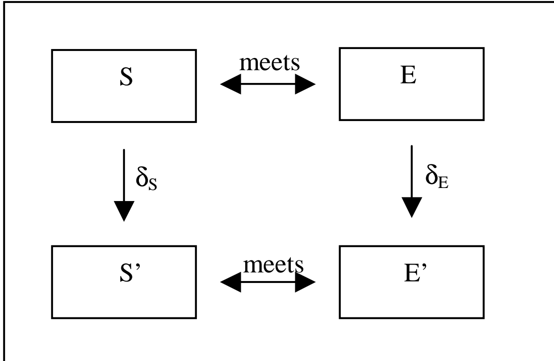

Figure 3. Explanation of Symbols in the Definition of Adaptation.

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 the two standards that we mentioned in the Introduction – GSM and CDMA (so-called dual-mode phones). 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 one standard the phone connections could be made faster than for the other standard, for example. 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 time. Also it is not necessary that the input space should be the same after adaptation.

Change in the behavior (δ_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 (δ_O) of the software system, and the change in the behavior (δ_B) are defined below.

If $I \; x \; S \rightarrow O$, and
$I \; x \; S' \rightarrow O'$, then

$$\delta_O = O' - O.$$

Also, if $I \times S \rightarrow B$, and
$I \times S' \rightarrow B'$, then

$$\delta_B = B' - B.$$

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

for $\delta_E \neq 0$, $\delta_B = 0$ and $\delta_O = 0$.

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. Before that, we will need definitions for the input and output spaces for the problem domain of interest to us in this paper (as explained in the Introduction). This is described in the next section.

2.3 Input and Output Spaces

For the problem domain of interest in this paper, the model of the embedded system as given in Figure 4 will be used for illustrating the concept of the input and output spaces.

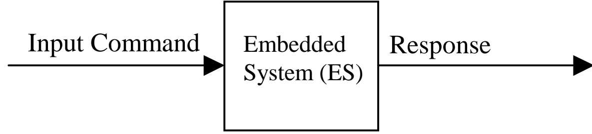

Figure 4. Input and Output Spaces for this Application Domain.

Input space is the set of legal commands for the embedded system. The *output space* is the set of responses expected from the system for legal commands.

3. APPLICATION OF THE NFR FRAMEWORK

As mentioned in the Introduction the various techniques to be developed in Section 4 will be compared using the NFR Framework [7, 15]. As per the NFR Framework, the following steps are required to complete the softgoal interdependency graph (hereafter, *SIG*) and evaluate the architectures:

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

Each of the above steps will be developed below.

3.1 Develop the Softgoal Hierarchy for the NFR Semantic Adaptation

In this step, the NFR semantic adaptation 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 NFR semantic adaptation. Each NFR in the decomposition is a softgoal that is satisficed (defined in Section 2.1) to different degrees by the designs we develop. The decomposition for semantic adaptation is given in Figure 5.

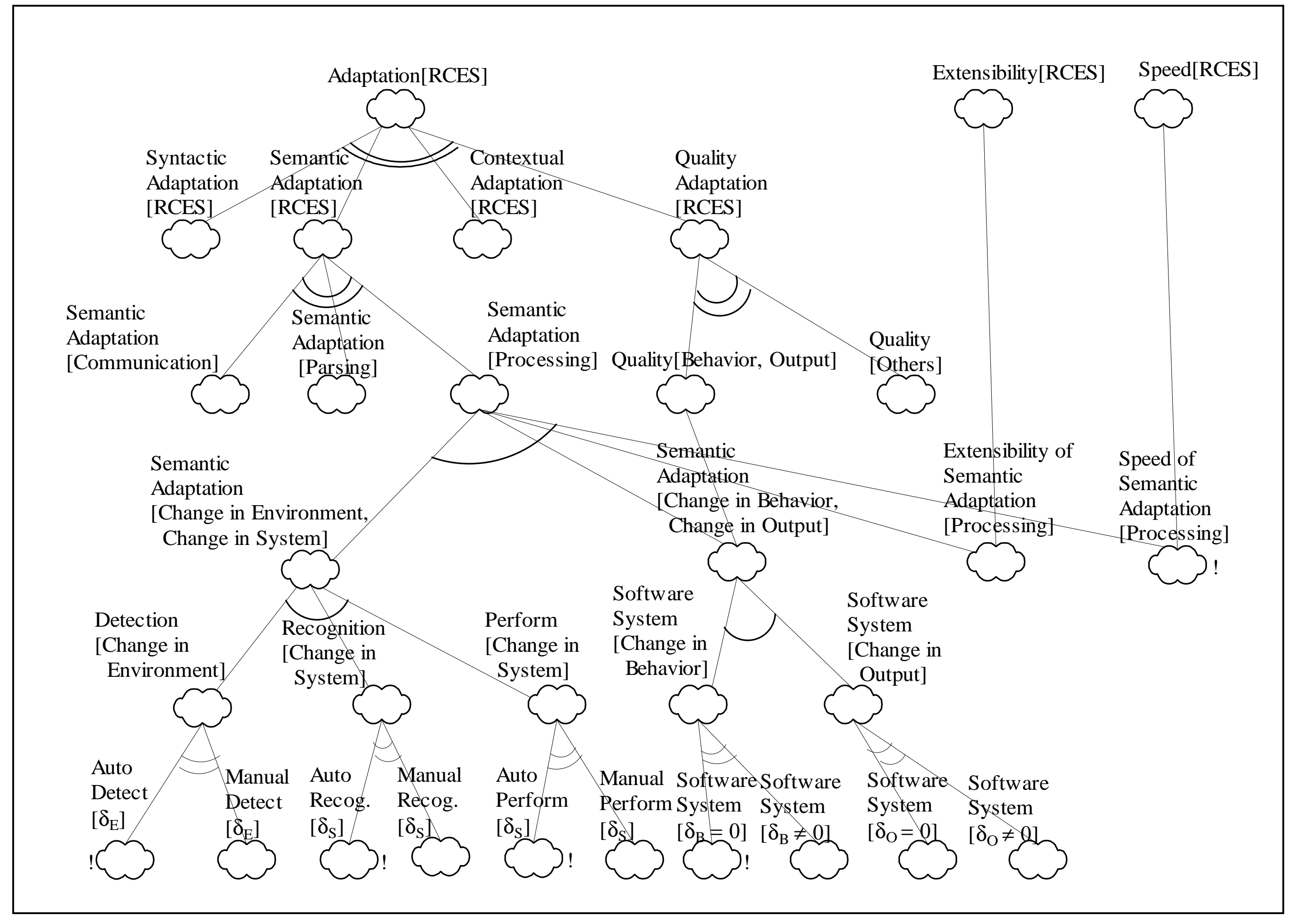

Figure 5. Softgoal Hierarchy for Semantic Adaptation

All softgoals in the decomposition given in Figure 5 (depicted as clouds) are named in the following convention:

Type[Topic1, Topic2, …],

where *Type* is an NFR and *Topic* is a system or another NFR[1] to which the *Type* applies.

[1] In [7, 15] it has been mentioned that *Topic* is a functional item – we deviate from this custom in this paper where *Topic* could be another NFR as well.

The decomposition given in Figure 5 reads from the top. At the top are the three high level softgoals – Adaptation[RCES], Extensibility[RCES] and Speed[RCES], where RCES stands for Remotely Controlled Embedded System. 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 the double arc, which stands for the OR-decomposition. Quality Adaptation of RCES involves adaptation of NFRs for the system. As mentioned in Section 2, we will be interested in the qualities of behavior and output (which stands for whether the output changes or not, and not for the actual output from the RCES), while there could be other qualities (or NFRs as well). This consideration results in the two OR-decompositions of the softgoal Quality Adaptation[RCES]. In this paper we are only concerned with semantic adaptation and so the softgoal Semantic Adaptation[RCES] is further OR-refined into the three functions of an RCES – Communication, Parsing (or Syntax Analysis) and Processing (or Semantic Analysis), which gives us the three soft-subgoals: Semantic Adaptation[Communication], 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, as defined in Section 2.1), Semantic Adaptation[Change in Behavior, Change in Output] (which again follows from the definition of semantic adaptation, as defined 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 satisficed in order for the softgoal Semantic Adaptation[Processing] to be satisficed. Further 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, satificing of the softgoal satisfices 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 means 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 that such a detection can take place- automatic and manual (we have used δ_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 δ_B to indicate Change in Behavior), and the softgoal Software System[Change in Output] is also OR-decomposed in the same manner.

The NFRs Software System[Change in Behavior] and Software System[Change in Output] have been defined in Section 5.1 for the application domain taken for the case study.

3.2 Develop Architectural Alternatives

The architectural alternatives will be developed in section 4.

3.3 Develop Design Tradeoffs and Rationale

As mentioned in Section 2.1, different architectural solutions satisfice various NFRs to different extent. We use the legend of Figure 6 in describing the different degrees of NFR satisficing.

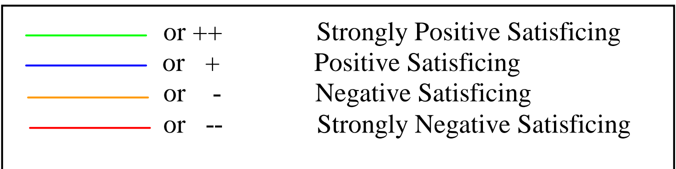

Figure 6. Symbols for Different Degrees of Satisficing.

For each architecture developed in Section 4, we will indicate the design tradeoffs and the rationale for the degrees to which the architectures satisfice the various softgoals of Figure 5.

3.4 Develop Goal Criticalities

For the particular application, different NFRs will have different criticalities. Criticalities in the NFR framework are shown in the SIG using '!' marks. In Figure 5, five NFRs are marked as critical: $\delta_B = 0$, Speed, Automatic δ_E detection, Automatic δ_S recognition, and Automatic δ_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

In this step the SIG is constructed and the most suitable architecture for the application is selected from the SIG. This step will be performed after the implementation phase.

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 (which we would like to call as the 3Rs technique) that is currently widely used. 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. An architecture that uses this technique is given in Figure 7. This is a high level architecture and each of the components in this architecture could have further sub-components. Here the components, connections and constraints may be changed as needed to meet the requirements of the new environment.

4.1.1 SIG for the 3Rs Technique

In order to develop the softgoal-interdependency graph for this technique we have to know the degree to which this architecture fulfills the various softgoals of the NFR decomposition given in Figure 5. Table 1 gives the rationale behind deciding the degree of satisficing of the various softgoals and Figure 8 gives the SIG for this technique. In this SIG the clouds in normal lines represent the softgoal requirements while the clouds in dark (or bold lines) represent the design elements (the architectural elements). The colored lines represent the degree of satisficing of the various softgoals and follow the legend of Figure 6. In this SIG

the extent to which the various software components satisfice the different softgoals were determined with the help of the domain experts.

Table 1. Softgoal Satisficing by the 3Rs Technique.

Softgoal	Degree of Satisficing	Rationale
$\delta_B = 0$	+	Can be ensured during modification
$\delta_B \neq 0$	-	Domain Expert
$\delta_O = 0$	++	In this application domain outputs are strings and repeatability is very high
$\delta_O \neq 0$	-	Domain characteristics
automatic detection [δ_E]	--	As no such capacity exists
manual detection [δ_E]	++	By design
automatic recognition[δ_S]	--	As no such capacity exists
manual recognition[δ_S]	++	By design
automatically perform[δ_S]	--	As no such capacity exists
manually perform[δ_S]	++	By design
Extensibility	++	Can be modified to any extent
Speed	-	By validation – Section 5.3

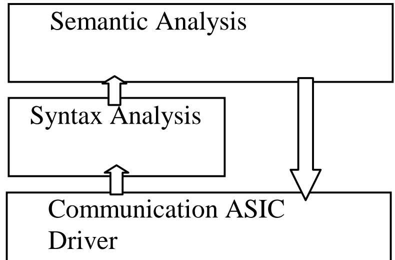

Figure 7. Architecture for the 3Rs Technique.

4.2 Stored Data Technique

In this technique the data for possible adaptations are stored in the embedded system in the beginning. At run-time, a state machine keeps track of current state of the system. Thus, in a cell phone measuring instrument that generates output signals for two different cell phone systems, say GSM and CDMA, and if the instrument generates signals in response to an input command "MAKE CALL" (from the external controller), then depending on the state the instrument is currently in, the signals corresponding to that cell phone system will be generated. This is indicated by the statechart given in Figure 9. As can be seen in that figure, the transition between the two states GSM and CDMA occurs due to another input command "CHANGE SYSTEM".

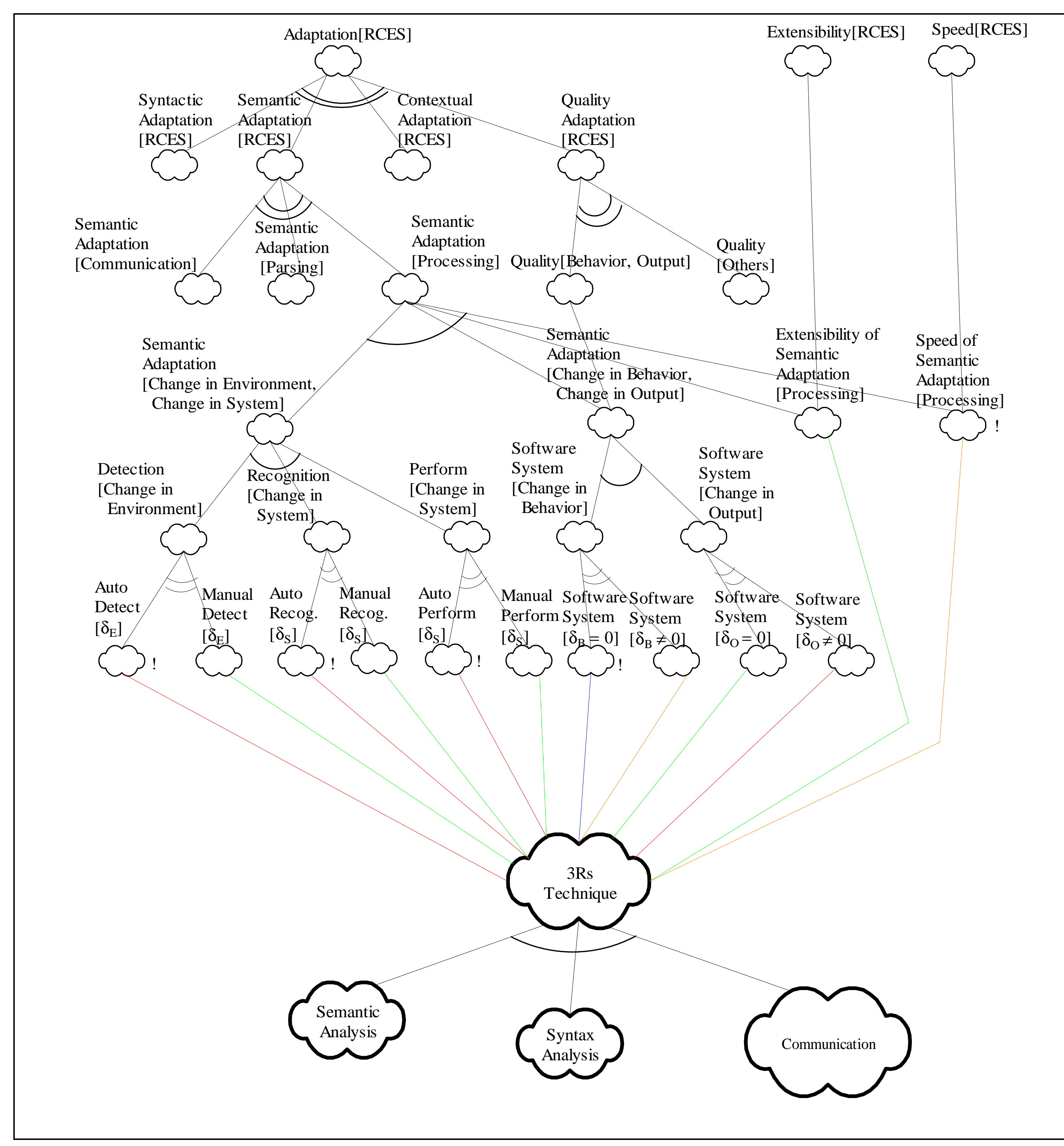

Figure 8. SIG for the 3Rs Technique.

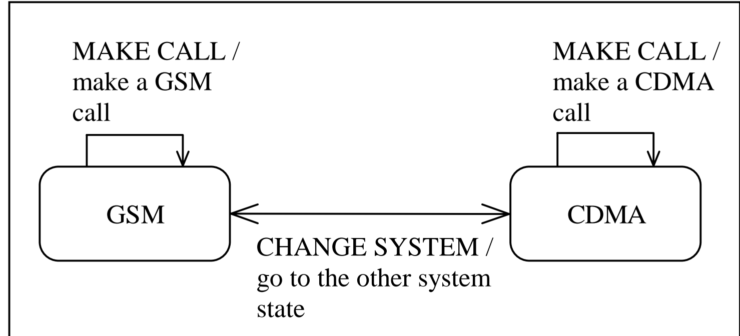

Figure 9. Statechart Diagram explaining the Stored Data Technique.

Thus in a system using this technique, all possible states are already present 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 the manual method, 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 the automatic method, the system receives signals (continuing using the example of Figure 9) 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 10 [13, 14], then each state may represent one of the leaf nodes in the hierarchy. In the class diagram of Figure 10, 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.

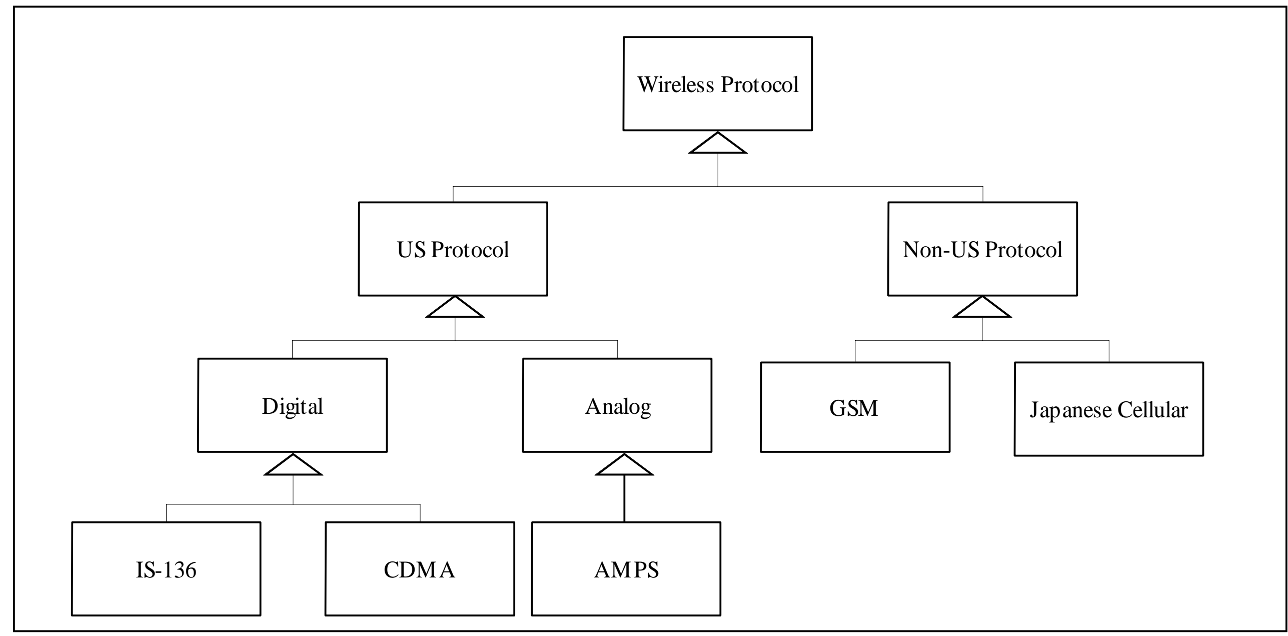

Figure 10. Class Diagram for the Stored Data Technique.

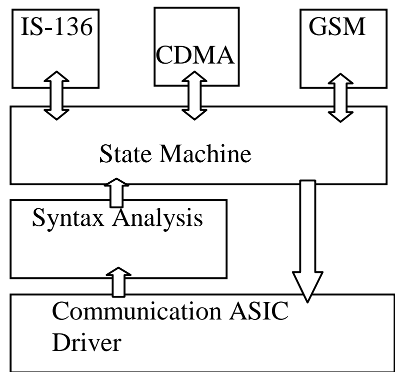

Figure 11. Architecture for the Class Diagram of Figure 10 using Stored Data Technique.

An architecture based on the stored data technique for this class diagram is shown in Figure 11. In this architecture there are three systems – IS-136, CDMA and GSM. A state machine keeps track of the current system and changes between systems based on an external command (there are such systems commercially available [13]). More generally, the architecture for a system using this technique is given in Figure 12. 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.

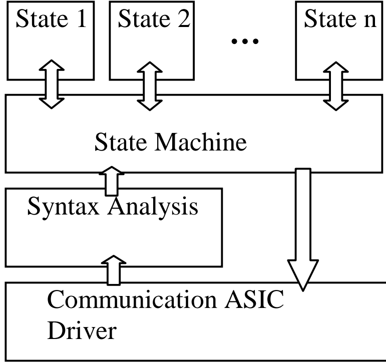

Figure 12. Architecture for the Stored Data Technique (Component Adaptation).

In the architecture of Figure 12, adaptation is achieved through components. Another way of adaptation is by having an adaptable connector. This architecture is given in Figure 13. Here the connector connects to the one component of interest (indicated by the darkened block arrow between the State Machine component and State 1 component) at the particular state.

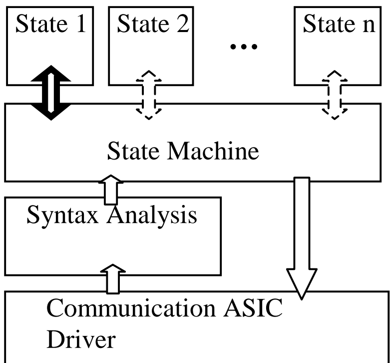

Figure 13. Architecture for the Stored Data Technique (Connector Adaptation).

Yet another way to achieve adaptation is by adapting the constraints on the connections between components. Thus while Figure 12 and Figure 13 place two different constraints (but fixed constraints) on the interconnections between the components, in the constraint adaptation technique, the interconnections can be changed as required. This is indicated in Figure 14.

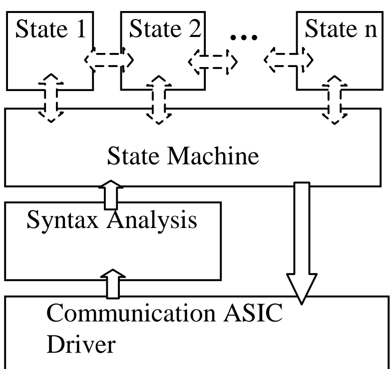

Figure 14. Architecture for the Stored Data Technique (Constraint Adaptation).

Each of the different types of adaptation has different advantages with respect to each other. Thus component adaptation is very fast; connector adaptation lets new components be added to the system easily; constraint adaptation lets the correct component for a particular state be found very fast.

4.2.1 SIG for the Stored Data Technique

Table A1 gives the rationale behind deciding the degree of satisficing of the various softgoals and Figure A1 gives the SIG for this technique. In this SIG the clouds in normal lines represent the softgoal requirements while the clouds in dark (or bold lines) represent the design elements (the architectural elements). The colored lines represent the degree of satisficing of the various softgoals and follow the legend of Figure 6. In this SIG also the extent to which the various software components satisfice the different softgoals were determined with the help of the domain experts.

4.3 Rule-Based Approach

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. For example, when a cell phone is being tested using a test equipment, one of the tests [14] that is done is to first make a call from the cell phone (the test equipment can emulate a base station), and then decrease the level of the signal being output by the test equipment and find the level when the cell phone drops the call. One way in which the software in the cell phone could decide call dropping is to check the level of the signal received continuously and then when the level goes below a preset value, drop the call. While the level at which to drop the call may be x for the current generation of base stations (for the same standard, say GSM), for the next generation of base stations it may be y. However, it would be extremely helpful if the cut-off level for the cell phones could be changed without any change in its software. In the rule-based approach, the level at which to cut-off is made a rule. Whenever, the level changes the cell phone software checks for the rule and detects if the level is below the cut-off rule. If yes, then the cell phone drops the call. In order to adapt to the newer generation of base stations, the cell phones will have to modify their rules to change the cut-off level to y. Figure 15 shows some examples of the rule-based approach.

Example 1:
Current Rule: cut-off level $< x$
Evolved Rule: cut-off level $< y$

Example 2:
Current Rule: upper limit 50
Evolved Rule: upper limit 60

Example 3:
Current Rule: value of constant is 23.85
Evolved Rule: value of constant is 24.35

Example 4:
Current Rule: minimum value -10
Evolved Rule: mininum value -20

Figure 15. Examples for Rule-Based Approach.

For the application domain considered in this paper, 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 15, if the command "CUTOFF_LEVEL X" referred to the current rule, and if by sending the command "CUTOFF_LEVEL Y" the current rule changes to the evolved rule, the needed evolution has been achieved.

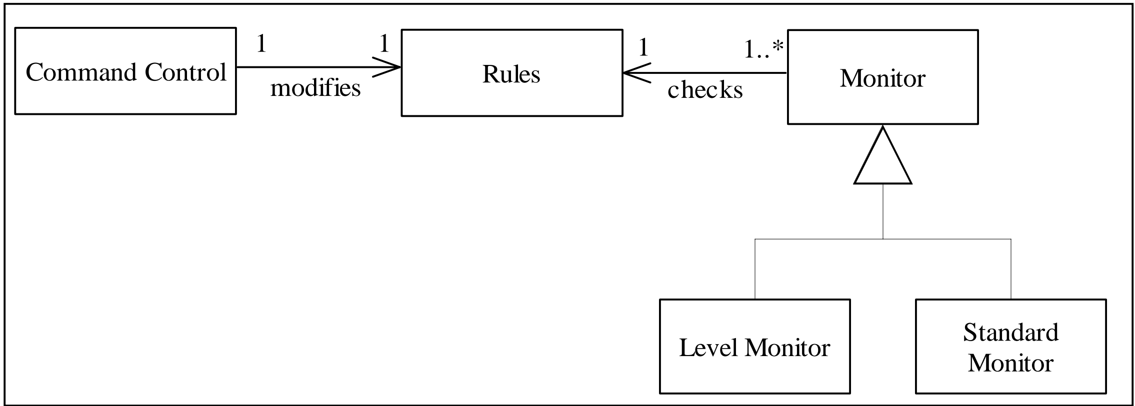

Figure 16. Class Diagram for the Rule-Based Approach.

The class diagram for the rule-based approach is given in Figure 16. In this diagram the class diagram for only the Semantic Analysis Block of Figure 2 is given. Here the Command Control class receives the codes for the input commands from the Syntax Analysis Block. The code causes the Command Control class to set the appropriate rule in the Rules class. Subsequently, whenever the input level to the cell phone changes, the Level Monitor class checks the Rules class for the cut-off level and takes appropriate action based on the rule. Likewise, the Standard Monitor is another class that monitors the wireless standard of the signals received by the cell phone. When the standard changes, the Standard Monitor class checks for any rules associated with this change in the Rules class and takes appropriate action based on the rule (if any).

Figure 17 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. Thus the adaptation given in Figure 17 is component-based adaptation.

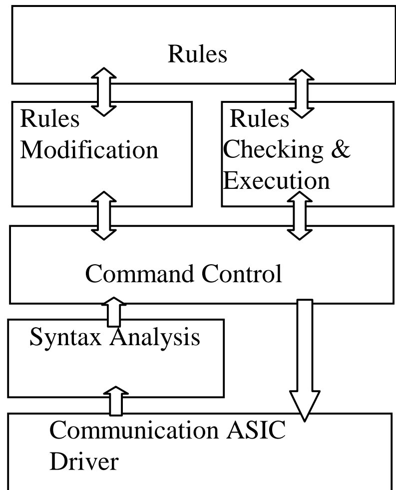

Figure 17. Architecture for the Rule-Based Approach (Component Adaptation).

For a connector-based adaptation for the rule-based technique, all the rules exist in the Rules class of Figure 16. What an external command for adaptation informs the system is which of the available rules to use. Here the connector between the Command Control component and Rules Component points to the correct rule for the current situation. Each of the rules is capable of modifying, checking and executing themselves. This is depicted in the architecture of Figure 18.

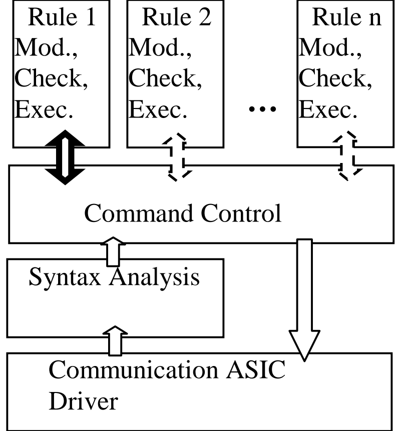

Figure 18. Architecture for the Rule-Based Approach (Connector Adaptation).

For a constraint-based adaptation using the rule-based technique, consider an example: before the cell phone drops the call, it may have to inform the user first. Upon adaptation, the constraints may be to first inform the user, log the time of call drop to a file and then drop the call. Thus the rules for dropping the call may inform the actions to be done before the call is dropped. However, the order in which the actions occur has changed. The architecture of Figure 19 shows the architecture for constraint adaptation.

In this case the environment change detection can only be done manually (as rules reflect physical or business limitations, usually), while the need for system change can be detected manually or automatically

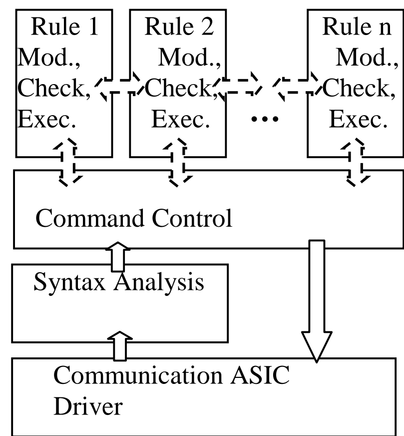

Figure 19. Architecture for the Rule-Based Approach (Constraint Adaptation).

while the system change itself can be done automatically. One of the major disadvantages of this technique is the fact that the rules have to be decided in advance. Subsequently, the rules can only be modified. The extent of adaptation is limited to the extent to which the rules permit adaptation.

4.3.1 SIG for Rule-Based Approach

Table A2 gives the rationale behind deciding the degree of satisficing of the various softgoals and Figure A2 gives the SIG for this technique. In this SIG the clouds in normal lines represent the softgoal requirements while the clouds in dark (or bold lines) represent the design elements (the architectural elements). The colored lines represent the degree of satisficing of the various softgoals and follow the legend of Figure 6. Again the extent to which the various software components satisfice the different softgoals were determined with the help of the domain experts.

4.4 Run-time Module Generation

This is a very powerful technique that allows system behavior to be changed 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 may be done with the help of commands from outside. Figure 20 shows the class diagram before and after adaptation in this technique. In this figure,

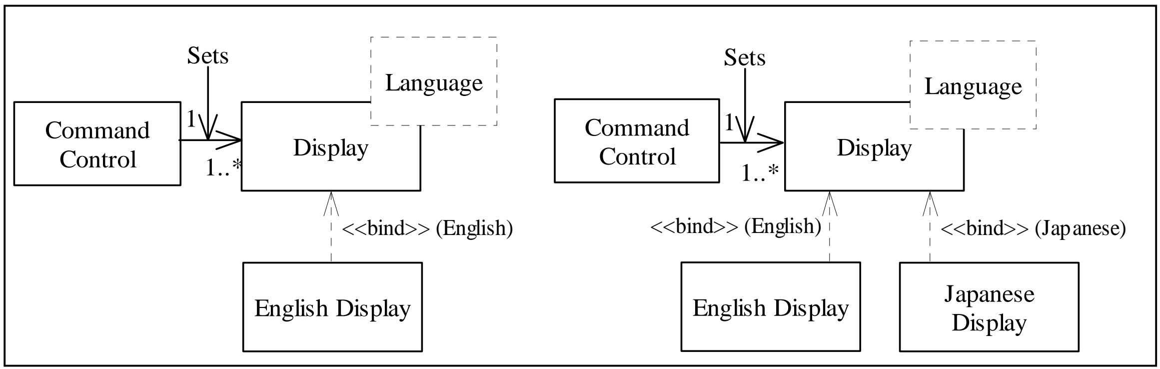

Figure 20. Class Diagram for Module Generation (diagram on left is before module generation and the diagram on the right is after module generation).

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 21 shows an example of this evolution with two architectures for the above example. In Figure 21a, the system has only the English display module. The Command Control module sends items to be displayed directly to the English Display module. However, upon receiving the command "JAPANESE DISPLAY" the Command Control module will instruct the Module Generator module to create a peer of the English Display module. Figure 21b shows the resulting system with the Japanese Display module and subsequent texts are displayed in Japanese by this module.

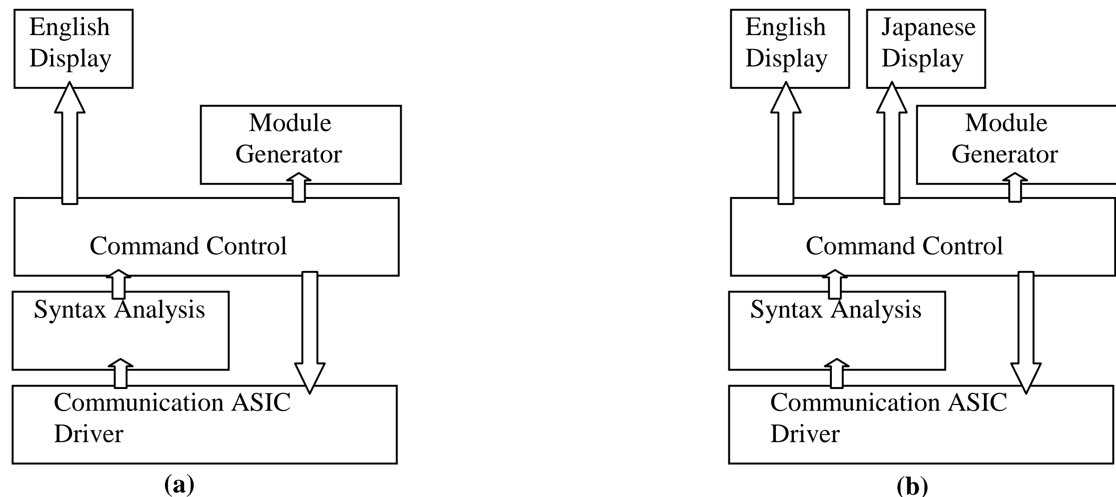

Figure 21. Example for Run-time Module Generation Technique.

The generic architecture for this technique is given in Figure 22. This architecture shows component-based adaptation technique. 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.

For connector-based adaptation of this technique, new connections are developed between components at run-time. This will be required if in the above example the matter displayed on the LCD should also be sent to a speaker. Then there will be new connection required between the Display class and the Speaker class. In this case the Module Generator class is capable of creating additional connections as well. This would mean making classes on either side of this new connection aware of each other.

Adaptation based on constraints is achieved similarly, with new constraints added whenever necessary.

Here the change in environment is detected manually, the need for a system change is detected manually or automatically and the system change itself is done automatically. One of the major disadvantages with this system is the additional memory required to add components. In memory-constrained embedded systems this technique could lead to problems. Also the implementation of this technique is not easy. There are very few methods of implementation for this technique – more on implementation of this technique is discussed in Section 5. However, the advantages are, at least theoretically, this technique gives the broadest range of adaptation to environment changes. This lets systems grow at run-time.

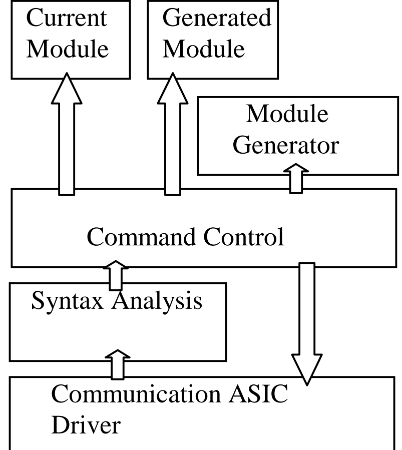

Figure 22. Architecture for Run-time Module Generation Technique (Component Adaptation).

4.4.1 SIG for Run-Time Module Generation Technique

Table A3 gives the rationale behind deciding the degree of satisficing of the various softgoals and Figure A3 gives the SIG for this technique. In this SIG the clouds in normal lines represent the softgoal requirements while the clouds in dark (or bold lines) represent the design elements (the architectural elements). The colored lines represent the degree of satisficing of the various softgoals and follow the legend of Figure 6. Again the extent to which the various software components satisfice the different softgoals were determined with the help of the domain experts.

5. VALIDATION OF THE TECHNIQUES FOR SEMANTIC EVOLUTION

5.1 Validation Approach

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

1. Implemented the architectures in an embedded system for a particular 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 timing measurements for adaptation indicate the relative speeds of adaptation for the different techniques. Also only the component- based adaptation was implemented for the different techniques.

In order to have a common understanding on the critical NFR ($\delta_B = 0$), we mean that the behavior is not changed if

1. the commands suited to the adapted environment are accepted
2. the time taken to execute the commands fall within satisficing limits before and after adaptation. In the

domain of interest in this paper, response times within 20ms is usually acceptable (this is based on the experience of one of the authors with various test instruments that execute commands sent from an external controller, usually a PC).

Likewise, the NFR ($\delta_O = 0$) means the following: if a command that produces an output, produces the expected outputs before and after adaptation, then there was no change in the outputs. Thus if a command "LEVEL?" produces an output in dBm before adaptation, and produces an output in different units or does not produce an output at all (assuming that this command was valid in the systems before and after adaptation) then there was a change in the output. It should be noted that if the command "LEVEL?" produced an output but in a different unit, then command has been accepted (i.e., $\delta_B = 0$, assuming condition 2 above is also satisfied) but $\delta_O \neq 0$. If, however, the command "LEVEL?" does not produce any output at all after adaptation, then this command has not been accepted, and $\delta_B \neq 0$ and $\delta_O \neq 0$.

The time taken to execute in this application domain is the time taken to execute an input string. This time includes two components – the time to parse the string (the time taken by the lower two blocks of Figure 2) and the time to actually execute the parsed code (the time taken by the Semantic Analysis block of Figure 2).

5.2 Problem for Implementing Adaptation

The environment change example is taken from the user-interface domain. 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 would like to be perform on the cell phone 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 (that is, the implementation uses object-oriented technology). 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 the external world get the value of the parameter using a command like "PARA_VALUE?". This situation is depicted in Figure 23, where the sequence diagram for the above interactions is given. 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 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 embedded system (or the Semantic Analysis Block) should accept this floating point value. The different techniques for semantic adaptation react to this environment change differently and the differences will be discussed in the implementation.

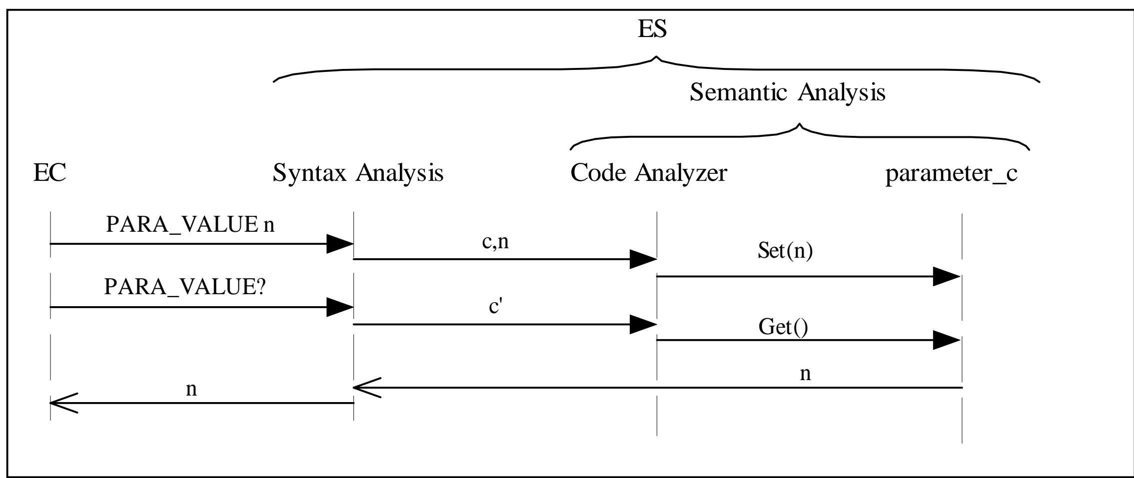

Figure 23 . Sequence Diagram for the Interactions in the Example Problem .

5.3 Implementation Using the 3Rs Technique

In the 3Rs technique, before the environment change occurs the code is changed to adapt the system to the new environment. Thus a new object called *parameter_f* would be created that stores floating point 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 the embedded system's memory and the embedded system has to be re-started. This is a slow process of adaptation, with adaptation time in the order of minutes. However, this is 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 δE detection, Automatic δS recognition, Automatic δS and the Speed of Semantic Adaptation. However, it does meet the fifth NFR, viz., Software System [δB = 0], as it fulfils both the conditions for satisfying the NFR (mentioned in Section 5.1). Table 2 shows this fact:

Table 2. Behavior Change in the 3Rs Technique Implementation.

State	Command	Performance
Before Modification (accepts integer values)	PARA_VALUE 100	Command Accepted
Before Modification (accepts integer values)	PARA_VALUE?	300 microseconds (execution time), 9 ms (response time)
Before Modification (accepts floating point values)	PARA_VALUE 10.5	Command Accepted
Before Modification (accepts floating point values)	PARA_VALUE?	1.926ms (execution time), 11ms (response time)

5.4 Implementation Using the Stored Data Technique

In the implementation of this technique, both the objects *parameter_c* and *parameter_f*, corresponding to the integer parameter and floating parameter, respectively, are already present in the system. If the parameter sent is a floating point parameter (detected by the '.') then the value of *parameter_f* is set, else the value of *parameter_c* is set. A flag is maintained to keep track of the last parameter value type, so that in response to PARA_VALUE? the correct object's Get( ) function is called.

Figure 24 shows the statechart diagram for this implementation. 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 Table 2. Adaptation times for this technique are given in Table B1. Here the times are given for automatic environment change detection. Had environment change detection been manual, we would have to send commands like "NEXT_PARA INT" before changing over to integer system and the command "NEXT_PARA FLOAT" before changing over to floating point system. As can be seen from Table B1, the adaptation is very fast.

5.5 Implementation Using the Rule-Based Approach

In this technique, a rule is set to indicate that the next parameter is a floating-point 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 floating point 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 rules as given in Table 3.
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 δ_E detection. The adaptation times for this technique are given in Table B2. In order to understand some of the entries in this table refer to Figure 25. Figure 25 shows the script that the external controller will execute for the environment change using this technique. For each command the corresponding effect in the embedded system is also indicated in that figure. The time taken to execute this script is about 120ms. 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.

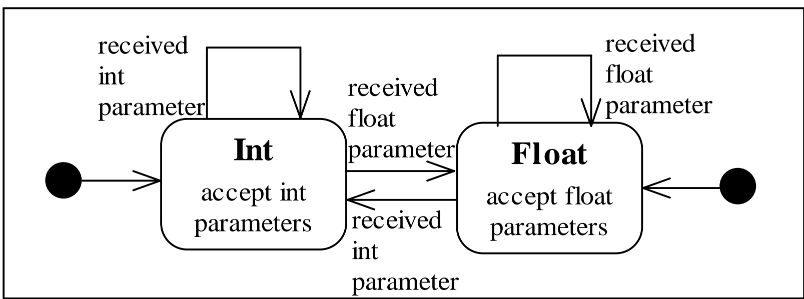

Figure 24. Statechart Diagram for the Stored Data Technique Implementation .

Table 3. Rules Used to Implement Rule-Based Approach.

Rule	Description	Command to Set Controlled Values	Command to Set Rule (include this rule into consideration)	Command to Unset Rule (exclude this rule from consideration)
Rule 1	Controls if parameter type has to be checked	NEXT_PARA INT and NEXT_PARA FLOAT	INCLUDE RULE1	EXCLUDE RULE1
Rule 2	Controls if upper limit value for the parameter has to be checked	UPPER_LIMIT n	INCLUDE RULE2	EXCLUDE RULE2
Rule 3	Controls if the constant value as mentioned in Figure 15 has to be checked and if so whether the constant value has to be added or subtracted	CONSTANT_VALUE f CONSTANT ADD CONSTANT MINUS	INCLUDE RULE3	EXCLUDE RULE3
Rule 4	Controls if the minimum values have to be checked	MIN_VALUE n	INCLUDE RULE4	EXCLUDE RULE4

EC	ES
Send(NEXT_PARA INT)	Sets para = int
Send(INCLUDE RULE1)	Sets rule 1 or R1
Send(PARA_VALUE 10)	*parameter_c*.Set(10) called
Send(PARA_VALUE?)	Based on R1, *parameter_c*.Get( ) called
Read( ) - receives 10	
Send(NEXT_PARA FLOAT)	Sets para = float
Send(UPPER_LIMIT 200)	Sets u_limit = 200
Send(INCLUDE RULE2)	Sets rule 2 or R2
Send(PARA_VALUE 20.0)	Based on R1 & R2, *parameter_f*.Set(20.0) called
Send(PARA_VALUE?)	Based on R1, *parameter_f*.Get( ) called
Read( ) – receives 20.0	

Figure 25. Script Running on External Controller (EC) and the Corresponding Actions taken by the Embedded System (ES) for the Rule-Based Approach

5.6 Implementation of the Run-time Module Generation Technique

Ideally speaking, implementation of this technique should work like this: a template class is created as shown in Figure 26. Based on the need, new classes of the type required are created dynamically from this template.

```
template <class C> class parameter
{
      public:
            void Set(C value) {val = value;};
             C     Get(void)     {return val; };

      private:
             C   val;
 };
```

Figure 26. A C++ Class Template for the Ideal Implementation of the Run-time Module Generation Technique .

Thus if an integer parameter has to be stored a class parameter_int is created from this template and the *parameter_c* object will be an instance of the parameter_int class. Likewise, if upon an external command like "CREATE_PARA FLOAT" a class parameter_float could be created from the template of Figure 26, then the *parameter_f* object could be instantiated from the class parameter_float. However, the currently available techniques do not allow this ideal situation to be realized easily. Firstly, the run-time generation of such objects will require a compiler to be present in the embedded system that will dynamically compile the code – called the 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. Also, dynamic compilation loses the advantages of performance optimizations available with static compilation. However, if these disadvantages could be overcome, then this technique is 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 the object instantiated from the above template is about 25 bytes. We also knew the addresses of the different functions from the map file. What we did is to change the binary code of the class 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 floating point 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*. The external controller used a script like in Figure 25 to overwrite memory and change the system. This will take a longer time (2.073 s 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 δ_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 do this step. Figure A4 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 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, 15] and we provide not only the methodology to perform this rationalization but also a decomposition of the adaptation NFR that is required by the Framework. 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 that we came up with included

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 (and sends responses) to an external controller over a communication link such as ethernet, serial, etc. Each of the techniques produced different architectures for semantic adaptation for this application. The codes that ensued from the different architectures were implemented in a real embedded system in this domain (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 of further research still open. 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 the run-time module generation techniques have to be studied further. Extension of concepts like the 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.

REFERENCES

1. E. A. Lee, "What's Ahead for Embedded Software?", *Computer*, September 2000, pp. 18 – 26.

2. P. A. Laplante, *Real-Time Systems Design and Analysis*, IEEE Press, Piscataway, New Jersey, 1992.

3. N. Subramanian and L.Chung, "Architecture-Driven Embedded Systems Adaptation for Supporting Vocabulary Evolution", *ISPSE* 2000, November, 2000, Kanazawa, Japan.

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 Systems*, 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*, Volume 3, Number 10, October 2000, pp. 44 – 48.

10. P. Johnson, "Rule-Based ELINT for Realtime Systems", *Texas Instruments Technical Journal*, Vol. 9, No. 4, July-August 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 and Interfaces Journal,* Vol. 22, Issue 5, December 2000, pp. 337 - 352 .

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

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

15. J. Mylopoulos, L. Chung, S. S. Y. Liao, H. Wang, E. Yu, "Exploring Alternatives During Requirements Analysis", *IEEE Software*, January/February 2001, pp. 2 – 6.

APPENDIX - A

SOFTGOAL INTERDEPENDENCY GRAPHS

This appendix gives the following data:

1. Softgoal satisficing tables (also called correlation tables)
2. Softgoal interdependency graphs

for the following techniques:

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

Table A1. Softgoal Satisficing by the Stored Data Technique.

Softgoal	Degree of Satisficing	Rationale
$\delta_B = 0$	+	Some change in behavior may exist in some states
$\delta_B \neq 0$	-	Only for a few states this is true
$\delta_O = 0$	++	In this application domain outputs are strings and repeatability is very high
$\delta_O \neq 0$	--	Domain characteristics
automatic detection [δ_E]	+	Is possible in some states
manual detection [δ_E]	++	Is possible in all states
automatic recognition[δ_S]	++	Can recognize new state to be in for the environment change
manual recognition[δ_S]	++	Is possible in all states
automatically perform[δ_S]	++	Can change state by itself
manually perform[δ_S]	--	By design
Extensibility	+	Can be modified to some extent
Speed	++	By validation – Section 5.4

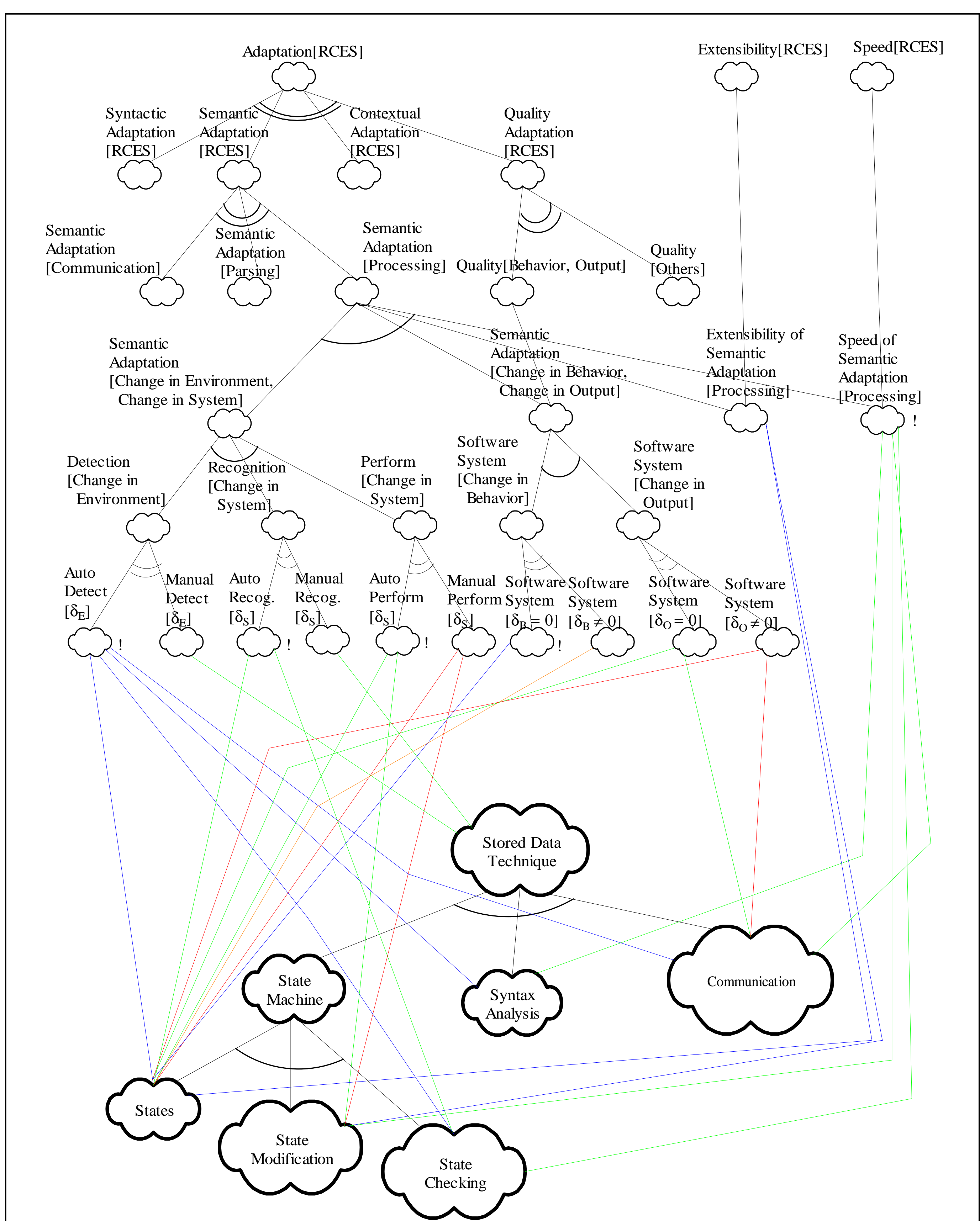

Figure A1. SIG for the Stored Data Technique.

Table A2. Softgoal Satisficing by the Rule-Based Approach.

Softgoal	Degree of Satisficing	Rationale
$\delta_B = 0$	+	Change of rules could affect behavior
$\delta_B \neq 0$	-	Some changes in rules could cause behavior change
$\delta_O = 0$	++	In this application domain outputs are strings and repeatability is very high
$\delta_O \neq 0$	--	Domain characteristics
automatic detection [δ_E]	--	Change in rules can only done by user
manual detection [δ_E]	++	Domain characteristics
automatic recognition[δ_S]	++	Need to change a rule can be recognized automatically
manual recognition[δ_S]	++	Domain characteristics
automatically perform[δ_S]	++	Can automatically change the rules
manually perform[δ_S]	--	By design
Extensibility	+	Can be modified to some extent
Speed	+	By validation – Section 5.5

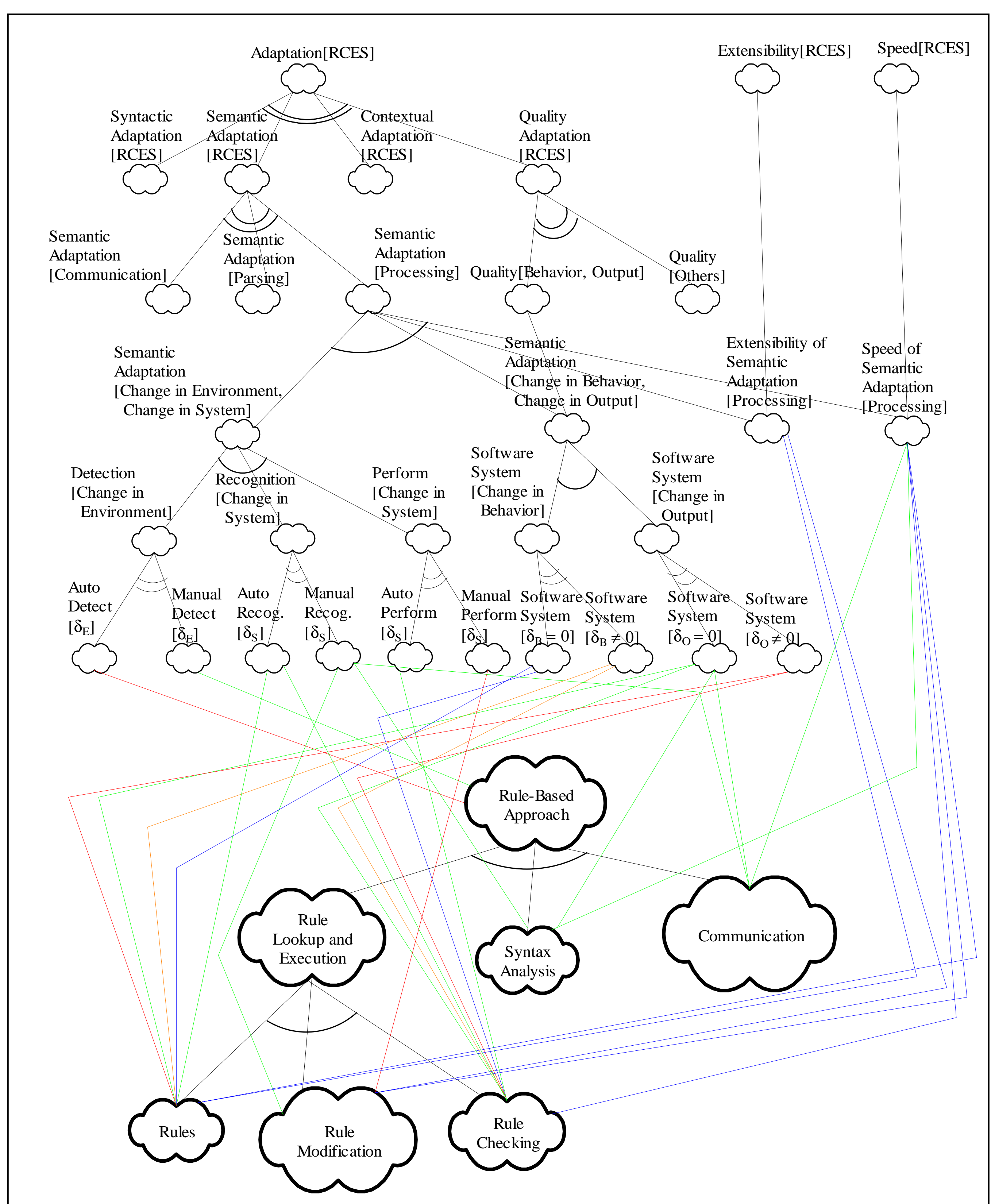

Figure A2. SIG for the Rule-Based Approach.

Table A3. Softgoal Satisficing by the Run-Time Module Generation Technique.

Softgoal	Degree of Satisficing	Rationale
$\delta_B = 0$	+	Addition of new modules could affect behavior
$\delta_B \neq 0$	-	Addition of new modules could cause behavior change
$\delta_O = 0$	++	In this application domain outputs are strings and repeatability is very high
$\delta_O \neq 0$	--	Domain characteristics
automatic detection [δ_E]	--	Need for new modules can only be detected by user
manual detection [δ_E]	++	Domain characteristics
automatic recognition[δ_S]	++	Need to create new modules can be recognized automatically
manual recognition[δ_S]	++	Domain characteristics
automatically perform[δ_S]	++	Can automatically generate new modules
manually perform[δ_S]	--	By design
Extensibility	++	Can be modified to a large extent
Speed	-	By validation – Section 5.6

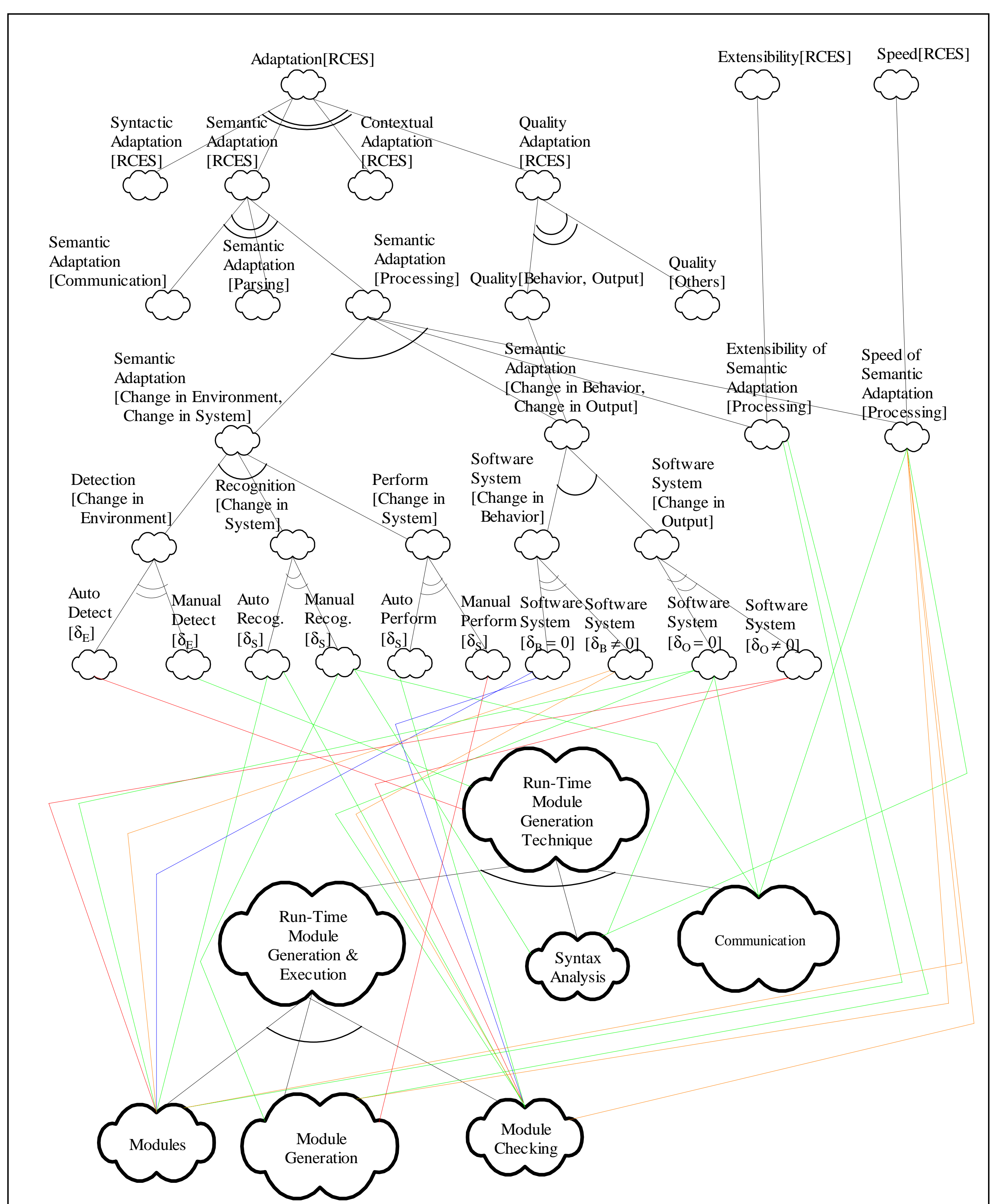

Figure A3. SIG for the Run-time Module Generation Technique.

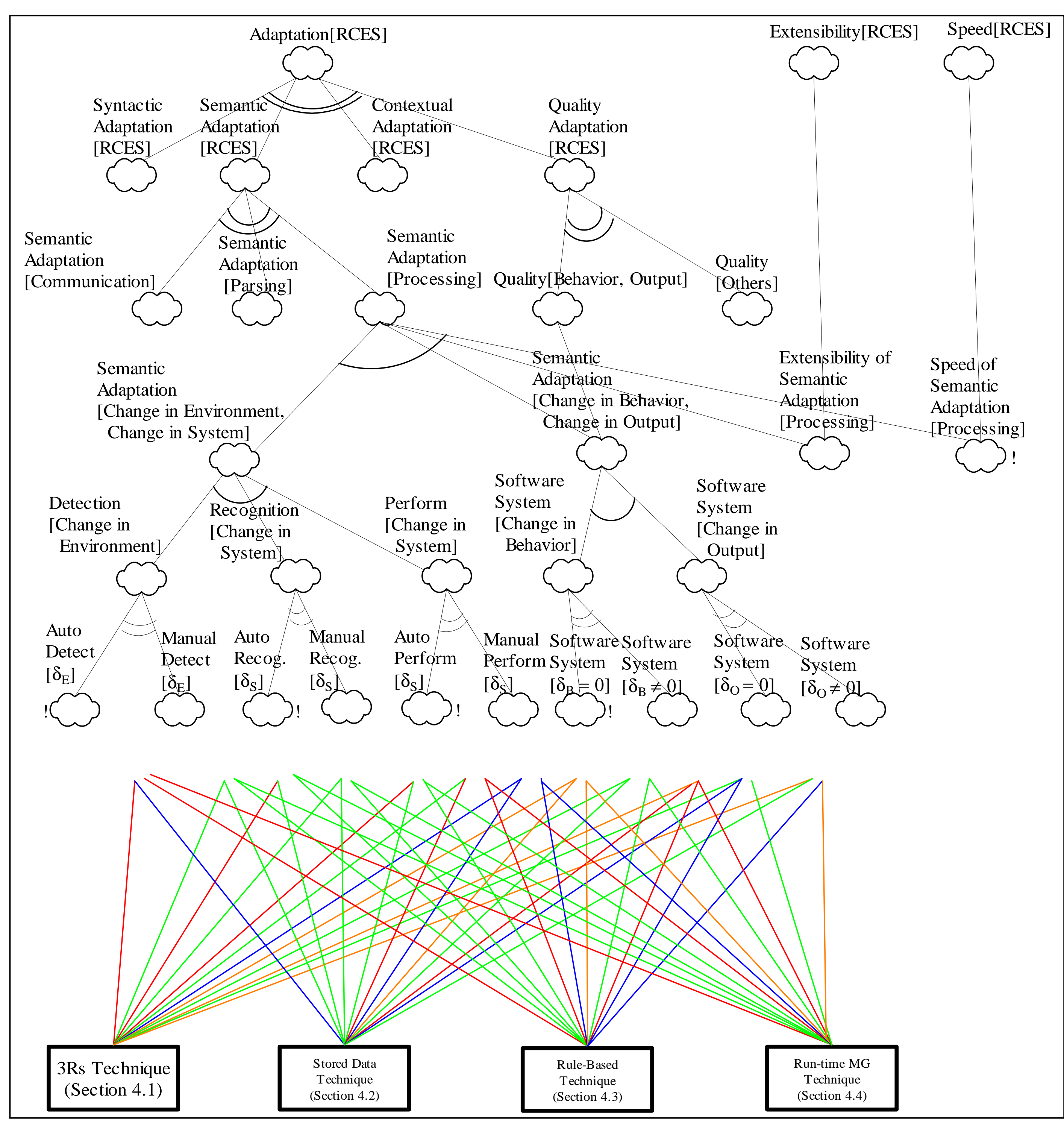

Figure A4. Combined SIG (partial) for All Techniques.

APPENDIX - B

VALIDATION TIMINGS

This appendix gives the timings for adaptation for the following techniques:

1. Stored Data Technique
2. Rule-Based Approach.

Table B1. Adaptation Times for Stored Data Technique Implementation.

Command	Parameter Type	$\delta_E \neq 0$?	Time for δ_E detection	Time for δ_S recognition	Time for δ_S and/or setting value
PARA_VALUE 100	Integer	Yes	140µs	124µs	167µs
PARA_VALUE 2000	Integer	No	142µs	Not Applicable	175µs
PARA_VALUE 30000	Integer	No	144µs	Not Applicable	180µs
PARA_VALUE 400000	Integer	No	147µs	Not Applicable	185µs
PARA_VALUE 10.05	Float	Yes	136µs	124µs	1.610ms
PARA_VALUE 396.89	Float	No	137µs	Not Applicable	2.033ms
PARA_VALUE 108476.998	Float	No	145µs	Not Applicable	3.542ms
PARA_VALUE 7849432.8897	Float	No	147µs	Not Applicable	3.870ms
PARA_VALUE 100	Integer	Yes	140µs	124µs	167µs

Table B2. Adaptation Times for Rule-Based Approach Implementation (the Initial State is INT)

Command Sent	Parameter Type	$\delta_E \neq 0$?	Time for δ_E detection (Rule 1 detect time)	Time for δ_S recog-nition	Rule 2 detect time	Rule 3 detect time	Rule 4 detect time	Time for δ_S or setting value
NEXT_PARA INT								
INCLUDE RULE1								
UPPER_LIMIT 50								
INCLUDE RULE 2								
PARA_VALUE 100	INT	NO	140µs		160µs			Not set
PARA_VALUE 45	INT	NO	140µs		154µs			132µs
PARA_VALUE? Read: 45								
CONSTANT_VALUE 23.85								
CONSTANT ADD								
INCLUDE RULE3								
NEXT_PARA FLOAT								
PARA_VALUE 33.99	FLOAT	YES	136µs	124µs	1.893ms	168µs		134µs
PARA_VALUE? Read: 57.84								
MIN_VALUE -10								
INCLUDE RULE4								
NEXT_PARA INT								
PARA_VALUE -5	INT	YES	140µs	124µs	147µs	169µs	122µs	128µs
PARA_VALUE? Read: 18								
CONSTANT_VALUE 24.35								
CONSTANT MINUS								
PARA_VALUE 35	INT	YES	140µs	124µs	151µs	168µs	122µs	134µs
PARA_VALUE? Read: 11								