Preliminary Design of ArchE: A Software Architecture Design Assistant

Felix Bachmann
Len Bass
Mark Klein

September 2003

TECHNICAL REPORT
CMU/SEI-2003-TR-021
ESC-TR-2003-021

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Preliminary Design of ArchE: A Software Architecture Design Assistant

CMU/SEI-2003-TR-021
ESC-TR-2003-021

Felix Bachmann
Len Bass
Mark Klein

September 2003

Architecture Tradeoff Analysis Initiative

Unlimited distribution subject to the copyright.

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Christos Scondras
Chief of Programs, XPK

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2004 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

Abstract vii

1 Introduction 1

2 Selection of an Expert System Platform 3
- 2.1 Theoretical Assumptions 3
- 2.2 About Expert Systems 3
  - 2.2.1 Logic Programming 4
  - 2.2.2 Rule-Based Systems 4
  - 2.2.3 Logic Programming Vs. Rule-Based Systems 4
  - 2.2.4 Inference Nets 4
  - 2.2.5 Frames 4
  - 2.2.6 Induction-Based Systems 5
  - 2.2.7 Neural Nets 5
- 2.3 ArchE as an Expert System 5

3 Responsibilities 9
- 3.1 Decomposition of Responsibilities 9
- 3.2 Role of Responsibilities 10

4 Solving Multiple Scenarios 11
- 4.1 Two Scenarios in the Context of the Same Reasoning Framework 11
- 4.2 Two Scenarios in the Context of Two Different Reasoning Frameworks 14
  - 4.2.1 Interaction Among Reasoning Frameworks 18

5 ArchE Operation 19
- 5.1 Key ArchE Data Concepts 19
- 5.2 Basic Activities of ArchE 20
  - 5.2.1 Step 1: Acquire Requirements 20
  - 5.2.2 Step 2: Refine Scenarios 21
  - 5.2.3 Step 3: Choose Reasoning Framework 21
  - 5.2.4 Step 4: Build Quality Attribute Models 22

5.2.5 Step 5: Build Design 23

6 Interaction Between Key Concepts and ArchE Activities 25

7 User Interactions with ArchE 27

7.1 Designer's Interactions with ArchE 27

7.1.1 Basic Interactions 27

7.1.2 Acquire Requirements 28

7.1.3 Refine Scenarios 28

7.1.4 Choose Reasoning Framework 28

7.1.5 Build Quality Attribute Models 29

7.1.6 Build Design 29

7.2 System Maintainer's Interactions with ArchE 30

8 Conclusions 31

Appendix: Detailed Description of ArchE 33

References 53

List of Figures

Figure 1: Overall Flow of ArchE....................6

Figure 2: Key Concepts of ArchE and Their Relationships 19

Figure 3: Blackboard Architecture of ArchE 34

List of Tables

Table 1: Key Concepts and How Activities Access Them....................................25

Abstract

This report presents a procedure for moving from a set of quality attribute scenarios to an architecture design that satisfies those scenarios. This procedure is embodied in a preliminary design for an architecture design assistant named ArchE (Architecture Expert), which will be implemented on a rule-based platform. This report includes the theory and rationale precipitating the design of ArchE and then describes this design in detail.

1 Introduction

In previous reports, we developed a theory of how architecture design decisions can be related to quality attribute achievement [Bachmann 02, Bachmann 03]. We focused on two quality attributes—modifiability and performance—and for each, we described how to derive a design fragment that supports the achievement of a single quality attribute scenario. In this report, we extend this work as follows:

- We describe how the theory developed previously can be applied to two scenarios, either for the same quality or different qualities.
- We present a preliminary design for an architecture assistant that will help a designer generate a software architecture design to satisfy requirements expressed as concrete quality attribute scenarios. We call this design assistant ArchE, which stands for Architecture Expert.

The motivation for our theory comes from our experiences in working with customers on software architecture design and evaluation. To utilize our theory fully, an architect must be an expert in many different quality attributes, and such expertise is rare. Furthermore, to apply the method resulting from our emerging theory, the architect must manage many relationships among the various attribute models. Both of these demands present problems of scale—in knowledge and in level of detail. As an attempt to deal with such problems, we are embarking on the construction of tool support in the form of an expert system. ArchE (in our vision) will have knowledge of all the quality attributes and will remember and report on the relationship among the attributes as a design progresses. This work represents our first attempt at designing such a system, and our goals for the initial version are modest:

- Demonstrate that constructing such an assistant is feasible.
- Demonstrate that quality attribute models can be integrated together to enable the semi-automatic achievement of a satisfactory design.

In other words, we are constructing ArchE because we believe that it will be useful, but its utility cannot be validated until it is constructed. Consequently, many unknowns exist, in both the details of ArchE's design and its ultimate utility. The promise that such an expert assistant holds, however, is so great that our investment in ArchE is more than justified.

We begin this report with our theoretical assumptions, discuss the reasons for our choice of an expert system platform, and present an overview of ArchE. In Sections 3 and 4, we discuss two extensions to our theory developed since our last report: the role of responsibilities and

how to manage multiple scenarios. In Sections 5, 6, and 7, we discuss ArchE *per se*, including key concepts and user interactions. The appendix contains a module decomposition view of ArchE.

2 Selection of an Expert System Platform

2.1 Theoretical Assumptions

Our approach is driven by four axioms:

1. Quality attribute requirements (including requirements for reuse, time to market, interoperability, performance, and modifiability) exhibit the most dominant influence on architecture design. They do so by exerting requirements on responsibilities that can be satisfied only by appropriately allocating responsibilities to architectural elements and properly assigning properties to responsibilities.
2. Given a quality attribute model (such as performance or modifiability) that satisfies particular quality attribute requirements, an associated set of architectural decisions can be inferred from the model.
3. Interactions between several quality attribute models can be identified by using responsibilities as the "communication glue" among the models.
4. An architecture flows from consistent quality attribute models and the architectural decisions inferred from them.

We explored some of these axioms in prior reports [Bachmann 02, Bachmann 03]. We explore others in Section 3.

2.2 About Expert Systems

As stated in the introduction, we are constructing an expert system for three reasons:

1. Being an expert in multiple quality attributes is very difficult for a single person.
2. Using our theory involves managing too many details without intelligent tool support.
3. An expert system that produces high-quality designs will provide validation to our theory.

We could have chosen any of the existing expert system technologies described in sections 2.2.1 - 2.2.7 [Giarrantano 98], but chose instead to construct our own, as discussed in Section 2.3.

2.2.1 Logic Programming

A logic programming language such as Prolog is based on the architect knowing the desired conclusion and developing a chain of reasoning that begins with known facts and yields this conclusion. This process is called backward-chaining rules. A theorem prover is an application of this type of expert system.

2.2.2 Rule-Based Systems

A rule-based system supports a collection of "if-then" rules that operate from a "fact" base. If the "if condition" is true based on the current facts, the "then" portion is eligible for execution—a process known as forward-chaining rules. Such a system must know how to choose which of multiple eligible rules to execute in the case of a conflict.

2.2.3 Logic Programming Vs. Rule-Based Systems

Both logic programming and rule-based systems can be applied to similar problems. If several hypotheses can be generated, logic programming can determine if a chain of reasoning exists for one of them. This process equates to generating the hypothesis through forward application of the rules. Some problems, though, lend themselves more naturally to one style over the other.

2.2.4 Inference Nets

An inference net is an expert system that relies on probabilistic reasoning and Bayesian statistics to determine likely transitions among nodes in a net representing semantic knowledge of a domain. The creation of a semantic net depends on a taxonomy of knowledge for the considered domain.

2.2.5 Frames

A frame represents limited knowledge about a narrow subject and is analogous to a record structure in a high-level language. Frames are good for managing mathematical-based knowledge in which no variations are allowed in the elements of frame. They do not represent variable knowledge well, however. For example, a frame describing an elephant might say that the elephant has four legs, but this limited knowledge excludes an elephant that has only three legs because of an accident.

2.2.6 Induction-Based Systems

This type of expert system learns by example. The system automatically generates rules with associated probabilities from the examples that it sees.

2.2.7 Neural Nets

A neural net is a particular type of induction-based expert system in which a body of knowledge is organized as a net and the probabilities of transition among the nodes are unknown. The examples given to the system result in a definition of the probabilities, which provides subsequent input.

2.3 ArchE as an Expert System

We chose to construct ArchE as a rule-based system with the quality attribute models viewed as frames. We selected Jess (a Java rule-based shell) as our initial platform because it is generally available, is implemented in Java, and allows a sophisticated user interface to be developed outside the expert system shell [Friedman-Hill 03]. We did not choose logic programming as a vehicle because we want to create a design, not justify one. A group at the University of Tandil has been working on expert support of the software architecture design process based on logic programming [Pace 03]. Both our effort and the University of Tandil's efforts are still too incomplete to allow a comparison of the results achieved. Also, Jaczone AB has introduced a commercial product called Waypointer that provides expert support for the Rational Unified Process but has no special provisions beyond that process to support architecture design.

We also ruled out induction-based systems since our theory states that we have the basis for deriving a design rather than understanding how to design by providing examples. Neural net systems were ruled out for similar reasons. Our theory is not built around the assignment of probabilities to rules. Some of the rules may themselves involve probabilities, such as those in a modifiability model, but the choice of rules is not determined probabilistically.

We visualize the interactions between ArchE and the designer as highly interactive, with the designer both providing information necessary for ArchE to proceed and specifying portions of the design. One model for ArchE is that of tax preparation software. That is, the software has a great deal of knowledge about the structure of the tax code and has the ability to check for consistency, but the information on which the tax return is based and the correctness of the resulting return depend on the information provided by the user. ArchE has knowledge of quality attributes and how to relate quality requirements to architecture design, but has no knowledge of the semantics of the system being designed. Consequently, without assistance,

ArchE cannot perform operations such as dividing a single responsibility into multiple smaller ones.

Figure 1 shows the overall flow of ArchE. Embedded in this figure are the following three concepts:

- quality attribute scenarios. We previously introduced a six-part formal structure for quality attribute scenarios involving a stimulus, a stimulus source, an environment, an artifact being stimulated, a response, and a response measure [Bachmann 02].
- reasoning frameworks. A reasoning framework is a body of knowledge about a particular quality attribute. A reasoning framework includes methods for calculating a response measure (the dependent parameter), given a collection of independent parameters. It also includes architectural tactics that enable independent parameters to be adjusted to affect (and control) the dependent parameter's value [Bachmann 03].
- responsibilities. A responsibility is an activity undertaken by the software being designed. ArchE uses responsibilities as a means of expressing functional requirements, as an integral portion of quality attribute scenarios, and as a means of integrating the models produced by various quality-attribute reasoning frameworks. For a discussion of the responsibilities used in object-oriented design, see the work of Wirfs-Brock and McKean [Wirfs-Brock 03]. Section 3 of this report discusses how responsibilities are used to integrate the models produced by various reasoning frameworks.

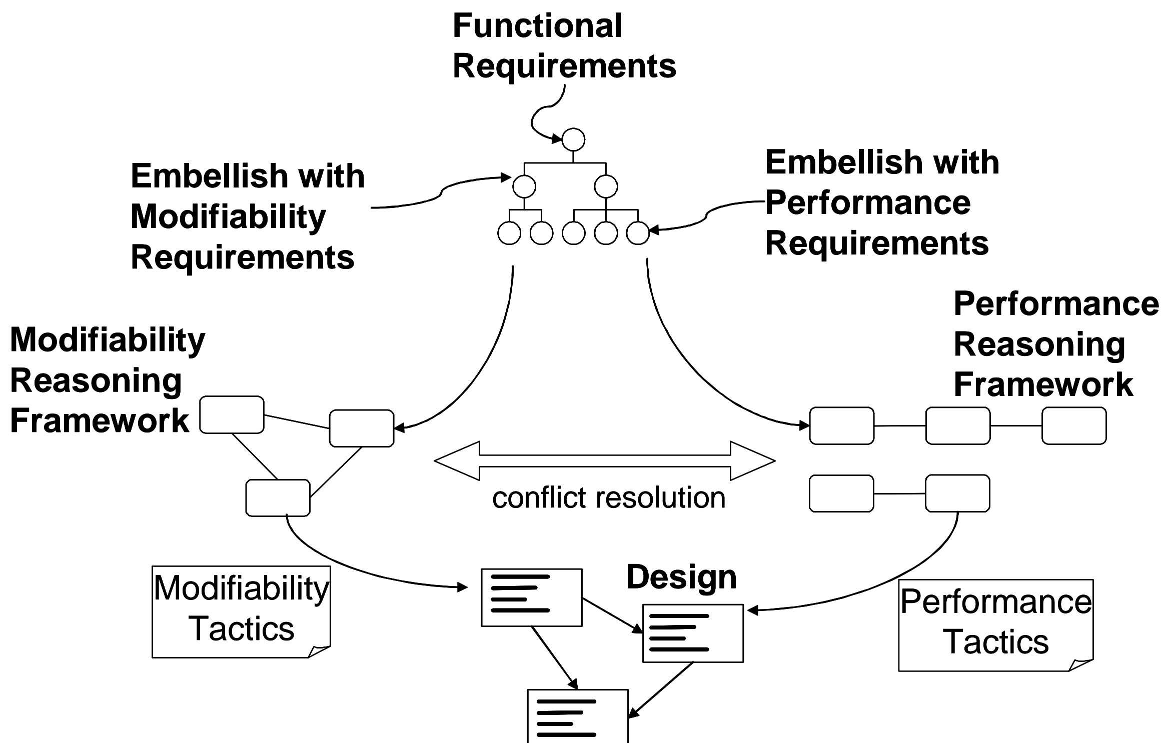

Figure 1: Overall Flow of ArchE

Within ArchE, functional requirements are represented as a responsibility graph and are embellished with quality attribute requirements in the form of quality attribute scenarios. These scenarios are categorized based on the quality attribute they specify. For each quality attribute, reasoning frameworks exist that convert the scenarios into a quality-attribute-specific model. Each model represents a design that satisfies the requirements specified for that quality attribute. The reasoning frameworks resolve conflicts among different quality-attribute-specific models to create an overall model that satisfies all the quality attribute requirements. This overall model is then converted into an architectural representation of the design.

3 Responsibilities

In this report, we discuss two extensions to our theory beyond those covered in our previous technical reports [Bachmann 02, Bachmann 03]: the role of responsibilities and how to manage multiple scenarios. This section focuses on responsibilities.

In our previous reports, we associated each quality attribute scenario with a quality-attribute reasoning framework. We discussed the process whereby a concrete scenario was determined to be an instance of a general scenario and how missing fields in the scenario were filled in. The process resulted in a concrete scenario that could be analyzed by a reasoning framework.

We used the reasoning framework to analyze a model to determine whether it could achieve the desired response-measure value. Then, after using the framework to specify the model's parameters, we extracted the responsibilities from the scenario to generate a design fragment that would achieve the response-measure value.

A responsibility as defined by Wirfs-Brock [Wirfs-Brock 03] is a property of a module in the module viewtype [Clements 03]. The responsibility consists of a brief textual description of the role(s) of the module within the system. A scenario, then, implicitly contains descriptions of roles that must be assumed by some module (or combination of modules) in a system achieving that scenario. One fundamental problem in generating a design for a software system is to define responsibilities and allocate each one to a module. The goal in generating the design is to define the responsibilities and their allocation in a way that satisfies both the functional and the quality attribute requirements for the system.

3.1 Decomposition of Responsibilities

Responsibilities can be decomposed. For example, the initial responsibility "halt garage door when obstacle exists" might be decomposed into "detect obstacle when one exists" and "halt door if obstacle detected." This decomposition results in one relationship among the responsibilities—containment of one responsibility by another. More generally, responsibilities exist in multiple relationships to each other, such as *precedes* and *intersects*:

- Responsibility A *precedes* responsibility B if, whenever A and B are in a sequence of responsibilities, A must be executed before B. This sequence can occur when A generates

data that B must have or when B provides needed services for A. For example, an obstacle must be detected before the garage door is halted.

- Responsibility A *intersects* responsibility B if the subresponsibilities of A have an intersection with those of B. This intersection can occur if A *contains* B totally (i.e., B is one of the subresponsibilities of A) or if A *overlaps* B (i.e., A and B have some subresponsibilities in common, but neither contains the other). For example, halting the garage door *contains* the subresponsibility of detecting an obstacle. Examples of overlap arise when modifiability tactics such as “abstract common services” are applied.

3.2 Role of Responsibilities

In previous reports, we viewed responsibilities as playing a passive role. That is, they were extracted from a scenario and assigned to particular modules of a design fragment. With multiple scenarios, responsibilities take on a more active role as follows:

- Responsibilities are used to connect the models analyzed by different reasoning frameworks. Applying tactics results in the addition of new responsibilities that must be treated by every reasoning framework relevant to any scenario. For example, application of the tactic “apply scheduling strategy” results in the responsibility “schedule units of concurrency.” The modifiability reasoning framework must assign this new responsibility to a module.
- The relationship among responsibilities may provide important cues for the reasoning framework. For example, overlapping responsibilities within two fixed-priority-scheduling scenarios may indicate a need for synchronization. If one responsibility is contained in another, the modifiability reasoning framework uses this information to guide its decomposition into modules.
- Properties of responsibilities include the information necessary to create a design. For example, the fixed-priority-scheduling reasoning framework assigns responsibilities to units of concurrency. The modifiability reasoning framework assigns responsibilities to modules. When modules are assigned to units of concurrency (e.g., processes), responsibilities are used to link the two.

Responsibilities can come from scenarios, requirements not embodied in scenarios, design specifications, or the application of a tactic. In cases where the new responsibilities affect the action of the reasoning frameworks, the relationship with the existing responsibilities must be determined. In some cases, such as the application of particular tactics, the relationship can be determined automatically. In other cases, such as the specification of a commercial off-the-shelf (COTS) component, the relationship might require manual determination.

4 Solving Multiple Scenarios

In a prior report, we addressed the problem of moving from a single scenario within the context of a reasoning framework to a design fragment [Bachmann 03]. Now we discuss the problem of moving from multiple scenarios to a design fragment that will satisfy their response-measure requirements. We divide our discussion into two parts: (1) two scenarios in the context of the same reasoning framework and (2) two scenarios in the context of different reasoning frameworks.

4.1 Two Scenarios in the Context of the Same Reasoning Framework

We begin by considering two scenarios. Dealing with multiple scenarios in the context of a single reasoning framework does not differ fundamentally from dealing with one scenario in a single reasoning framework. We use the same garage door example as in our previous reports [Bachmann 02, Bachmann 03]:

- A garage door must detect an obstacle and halt within .1 seconds.
- The garage door must respond to a user request to lower the door within .3 seconds.

Both are deadline scenarios; the response measure is latency, and the scenario provides a deadline that acts as an upper bound on the allowable latency. For the purpose of this example, we assume that the relevant reasoning framework is a fixed-priority-scheduling reasoning framework. In the context of ArchE, we use heuristics that enable a choice between a fixed-priority-scheduling reasoning framework and a cyclic-executive reasoning framework. Both enable the treatment of deadline scenarios.

To achieve our ultimate vision of using reasoning frameworks as a key element of design determinants, we will need many reasoning frameworks (at least one for each of the possible quality attributes) and a process for choosing which among them will provide the context for a scenario's solution. For now, since we have only performance and modifiability reasoning frameworks, we ignore this problem.

We perform these basic steps, based on the concepts in our previous report [Bachmann 03]:

1. Detect responsibilities and their relationships.
2. Determine the reasoning framework.

3. Determine the initial hypothesis for values of free parameters.
4. Determine and apply tactics.

Step 1: Detect responsibilities and their relationships.

The designer converts scenarios into a standard six-part form, from which responsibilities are derived. The obstacle-detection scenario yields two responsibilities: "detect obstacle" and "halt door." "Detect obstacle" must precede "halt door." The "respond to a user request scenario" yields two responsibilities: "detect request to lower the door" and "lower door." "Detect request to lower the door" must precede "lower door."

Implicitly, all other responsibilities of the garage door are included because the purpose of the system is to manage the garage door. Thus, this step provides an initial decomposition of the overall responsibility "manage the garage door."

This decomposition yields the following five responsibilities:

1. Detect obstacle.
2. Halt door.
3. Detect user request to lower the door.
4. Lower door.
5. Do everything else involved in managing the garage door.

Precedence relationships occur between some of the responsibilities. An overlap also might exist between the responsibilities "detect obstacle" and "detect user request to lower the door." This overlap is not of interest to the fixed-priority-scheduling reasoning framework, however, since from its perspective, each responsibility can be implemented individually. This overlap may be of interest from the modifiability reasoning framework's perspective, but if modifiability is important in detecting sensor input, a modifiability scenario should exist to identify this overlap.

If the overlap of responsibilities is, in fact, identified during a different part of the process, the fixed-priority-scheduling derivation must synchronize responsibilities derived from multiple scenarios. We do not concern ourselves with this complication here.

A contains relationship exists between "manage garage door" and the five responsibilities enumerated above. This relationship has no impact on subsequent activities, however. A decomposition of a responsibility has impact to the reasoning frameworks only if the parent responsibility that was decomposed occurs in a scenario.

Step 2: Determine the reasoning framework.

We assume that both scenarios are to be analyzed in the context of a fixed-priority reasoning framework. As we said, this procedure must become much richer to support realistic design exercises.

Step 3: Determine the initial hypothesis for values of free parameters.

The designer typically estimates the execution time for each set of responsibilities. These estimates comprise the first hypothesis to be tested and include (without much concern for reality)

- detect obstacle - 10ms
- halt door - 15ms
- detect user request to lower the door - 25ms
- lower door - 50ms
- do everything else - 300ms

Step 4: Determine and apply tactics.

At this point, the process maps directly to Step 5 in our report about deriving architectural tactics [Bachmann 03]. Since this example is hypothetical, we present the hypothetical result of applying the tactic. Notice, however, that the presence of multiple scenarios enters into the testing of the hypothesis. With multiple scenarios, several deadlines must be tested. Each time a tactic is applied, one of the parameters changes, resulting in another hypothesis that can be tested.

Stop the steps below as soon as the deadlines are satisfied with the current parameter values:

1. Apply tactics for managing demand.
   a. Reduce computational overhead.
      Currently no overhead is included in the timings.
   b. Reduce execution time.
      The execution times cannot be reduced appreciably.
2. Apply tactics for arbitrating demand.
   a. Increase logical concurrency.
      Three different sequences can be performed concurrently: responsibilities 1 and 2, 3 and 4, and 5. The reasoning framework knows about these concurrent sequences because of the relationships among the responsibilities. A precedence relationship exists between 1 and 2; hence they cannot be performed concurrently and similarly with 3 and 4. No other relationships exist among the responsibilities. The concur-

rent sequences become the units of concurrency output by the reasoning framework.

b. Determine the appropriate scheduling policy.
   Since we are within the fixed-priority-scheduling reasoning framework, applying this tactic means assigning priorities to the units of concurrency. Assigning a high priority to responsibilities 1 and 2, a medium priority to responsibilities 3 and 4, and a low priority to responsibility 5 satisfies the scenarios.

When we apply the tactic "increase logical concurrency," we add the new responsibility "schedule units of concurrency" to our list. This responsibility has an execution time and no relationship to the other responsibilities. Its execution time is factored into the determination of whether the deadlines are satisfied with the current set of parameters. Adding this responsibility is an adjunct of applying the tactic without regard to its use in any particular problem.

Each reasoning framework uses its own strategy when considering how to assign parameters so that a quality attribute model can be created and then solved to determine whether a scenario is achieved. In the worst case, all possible combinations can be tested, which is clearly not efficient, and optimizations of the strategy for assigning parameters are likely. For the fixed-priority-scheduling case, Rate Monotonic Analysis provides a method for assigning priorities based on computation times and deadlines.

Deriving a consistent model for two scenarios in the content of a single reasoning framework is, as we have seen, not substantially different from the problem of deriving a consistent model for a single scenario.

4.2 Two Scenarios in the Context of Two Different Reasoning Frameworks

Now we consider two scenarios in the context of two different reasoning frameworks—a deadline scenario and a modifiability scenario. This example illustrates the interactions among different reasoning frameworks. Basically, the reasoning frameworks need to operate on their respective scenarios in parallel with responsibilities that act as the communication mechanism among the frameworks. To illustrate this interaction, we use the following two scenarios (again with the general problem of managing a garage door):

- The garage door must detect an obstacle and halt within .1 seconds.
- Produce a new product based on a different processor within one person-day.

We go through the same steps as before and an additional one that iterates through the reasoning frameworks. Although iteration among the frameworks is not really a substitute for a parallel solution, it is a close approximation. In parallel operation, however, a hypothesis is generated for all relevant reasoning frameworks. If it is rejected by one of them, a new hy-

pothesis is generated that all reasoning frameworks must consider. With iteration, failed hypotheses should not be reconsidered, as this action causes complications. Our steps here include the following:

1. Detect responsibilities and their relationships.
2. Determine the reasoning frameworks.
3. Iterate through the reasoning frameworks, following steps 4 and 5 for each, until all scenarios are either satisfied or determined as not satisfiable.
4. Determine the initial hypothesis for values of free parameters.
5. Determine and apply tactics.

Step 1: Detect responsibilities and their relationships.

The designer converts the scenarios into a standard six-part form, from which responsibilities are derived. The obstacle-detection scenario yields two responsibilities: "detect obstacle" and "halt door." "Detect obstacle" must precede "halt door." The change-processor scenario yields processor responsibilities.

Implicitly, all the other responsibilities of the garage door are present since the system's purpose is to manage the garage door. Thus, this step provides an initial decomposition of the overall responsibility "manage the garage door."

This decomposition yields the following four responsibilities:

1. Detect obstacle.
2. Halt door.
3. processor responsibilities
4. everything else

Precedence relationships occur between two of the responsibilities—"detect obstacle" must precede "halt door."

For our example, the designer must specify additional dependencies among the responsibilities. In our report about deriving architectural tactics, we rationalized using a compiler and runtime library as two of three intermediaries [Bachmann 03]. In this example, we are concerned only with dependencies on unique features of the processor. Thus, the designer must specify which of the responsibilities—"detect obstacle," "halt door," and "everything else"—is dependent on unique features of the processor. For purposes of this example, we assume that only "everything else" is dependent on the unique features of the processor.

A contains relationship exists between "manage garage door" and the four responsibilities enumerated above. This relationship has no impact on subsequent activities, however. A de-

composition of a responsibility has impact on the reasoning frameworks only if the parent responsibility that was decomposed occurs in a scenario.

Step 2: Determine the reasoning frameworks.

We choose the fixed-priority-scheduling reasoning framework as in Section 4.1 to solve the obstacle-detection scenario. We choose the modifiability reasoning framework to solve the change-processor scenario.

Step 3: Iterate through the reasoning frameworks, following steps 4 and 5 for each, until all scenarios are either satisfied or determined as not satisfiable.

We begin with the fixed-priority-scheduling reasoning framework, which yields, similar to Step 4 on page 13, two units of concurrency and an additional responsibility of "schedule units of concurrency." Next we invoke the modifiability reasoning framework.

Step 4: Determine the initial hypothesis for values of free parameters for the modifiability reasoning framework.

The initial hypothesis for values of the free parameters within the modifiability reasoning framework is that all responsibilities detected in Step 1 are assigned to their own module.

The modifiability model requires knowledge of the dependencies among the responsibilities. The initial dependencies are the sequencing dependencies introduced by the obstacle-detection scenario. The application of tactics transforms the responsibilities, causing them to be modified, but each tactic determines how the dependencies are allocated among the transformed responsibilities.

The modifiability reasoning framework uses a cost model to determine if a particular assignment of parameters satisfies the scenario. Since cost models do not exist for the level of architectural decisions discussed here, we must create a cost model to apply our theory. Developing such a model will require validation with actual data, and that is beyond our scope here. We assume the existence of such a model and will use a crude version of it in our implementation of ArchE that combines "dead reckoning" and designer input.

Step 5: Determine and apply tactics for the modifiability reasoning framework.

The process is the same as that described in our report about deriving architectural tactics [Bachmann 03]. Since this example is hypothetical, we present the hypothetical result of applying the tactic.

Stop the steps below as soon as the cost of changing the processor with the current parameter values is below the constraint specified in the scenario. We iterate through these steps only once, although multiple iterations are likely in reality.

1. Apply tactics for localizing expected modifications.

   The expected modification (changing the processor) is already localized.

2. Apply tactics for restricting the visibility of responsibilities.

   These tactics are not relevant since nothing can be done about the visibility of the processor's responsibilities.

3. Apply tactics for preventing the ripple effect.

   a. Break the dependency chain.

      We initially have two dependencies—the sequencing dependency between "detect obstacle" and "halt door" and the semantic dependency between "everything else" and "processor responsibilities." We cannot break the sequencing dependency, but we can introduce an intermediary between "everything else" and "processor responsibilities." This intermediary (whose purpose is to mask processor-unique features) decomposes the "everything else" responsibility into "dependent on processor-unique features" and "not dependent on processor-unique features." Furthermore, no dependency exists between "not dependent on processor unique features" and the intermediary. A dependency might occur, however, between the two responsibilities of the decomposed "everything else" responsibility. For those dependencies that cannot be automatically inferred, a best guess, a worst-case guess, or designer input is necessary.

   b. Make the data self identifying.

      This tactic is not relevant.

   c. Limit communication paths.

      This tactic is not relevant.

We assume that introducing the intermediary to break the dependency will reduce the cost sufficiently to satisfy the cost constraint. If not, additional iterations through the tactics may introduce additional refinements in the responsibilities. Our point here, however, is to examine the interaction among the reasoning frameworks.

Notice that this use of the modifiability reasoning framework introduced a new responsibility ("the intermediary masks processor-dependent features") and decomposed existing responsibilities. The introduction of new responsibilities is a signal that the fixed-priority reasoning framework must reconsider its actions since the new responsibility is not assigned to a unit of concurrency with a priority. Decomposed responsibilities also may trigger action from the fixed-priority reasoning framework if the responsibility being decomposed contributes to a scenario, preventing the framework from accurately determining if the scenario still can be satisfied.

Step 6: Return to the fixed-priority-scheduling reasoning framework.

Since we have only one scenario being solved by the fixed-priority-scheduling reasoning framework, this scenario must be reexamined by that framework. In our example, this action results in the assignment of the new responsibility "mask processor-dependent features" to a unit of concurrency. None of the responsibilities involved in the obstacle-detection scenario is affected, so that scenario still should be satisfied.

On the other hand, if we had affected one of the responsibilities involved in a deadline scenario, satisfying the deadline through the application of tactics might not be possible. In this case, the modifiability reasoning framework must produce a new hypothesis. This situation shows the distinction between iteration among the reasoning frameworks and parallel operation of the reasoning frameworks mentioned on page 14.

4.2.1 Interaction Among Reasoning Frameworks

In this example, we see some of the possible interactions among reasoning frameworks:

- New responsibilities not previously treated by the reasoning framework are introduced.
- Responsibilities (or their parameters) that were previously satisfied by the reasoning framework are modified.

Each reasoning framework must keep track of the tactics it has tested so that if it is re-invoked, those tactics will not be tested further. Furthermore, cycles through reasoning frameworks must be recognized. If such a cycle is detected (e.g., using a criteria of a maximum of N iterations among the reasoning frameworks), the designer is informed of failure together with the results of the several best attempts. The designer then has the option of selecting one of the options, changing the scenarios, or choosing another activity.

5 ArchE Operation

Now we turn our attention from theoretical discussion to how we visualize ArchE operating. We discuss the key ArchE concepts, the basic activities of ArchE, and the interaction between them. In the appendix, we present the architecture of ArchE, a collection of modules, and rules for these modules that can act as an initial step toward an implementation of ArchE.

5.1 Key ArchE Data Concepts

The key concepts include scenarios, responsibilities, QualityAttributeModel, and design. We are not concerned here with representation but rather with informational content. Figure 2 shows the key concepts and their relationships.

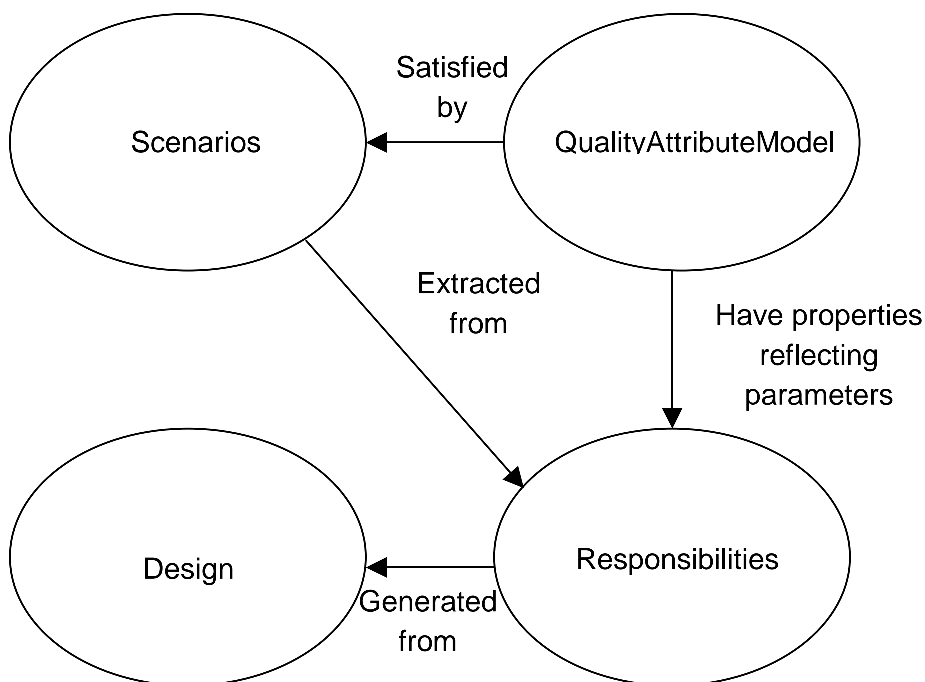

Figure 2: Key Concepts of ArchE and Their Relationships

Briefly, the informational content within the key concepts include

- scenarios—the quality scenario requirements for the system. Scenarios may have been refined into their constituent portions and may have been associated with reasoning frameworks.
- responsibilities—all the responsibilities identified within the system. They are linked to their source (i.e., scenario, requirements, designer-specified portion of the design, or tactic), include the relationship among the requirements as shown in Figure 2, and have parameters that include allocation to architectural elements and properties needed by the various reasoning frameworks, such as execution time.
- QualityAttributeModel—A quality attribute model is a fully instantiated instance from a reasoning framework—in other words, the independent parameters for the reasoning frameworks. These parameters are linked to their source (i.e., scenario, designer specifications, results of the application of a particular tactic), as well as to the responsibilities (if appropriate) to which they pertain.
- design—an enumeration of architectural elements, their properties, and their relationships

5.2 Basic Activities of ArchE

The basic activities of ArchE are based on the steps described in our theory, with the addition of activities to acquire requirements and actually build a design.

We describe the activities and how they interact with the key concepts. The section headings (acquire requirements, refine scenarios, choose reasoning framework, build quality attribute models, and build design) correspond to activity names within ArchE that we subsequently call ArchE goals and implement as rule-based modules.

5.2.1 Step 1: Acquire Requirements

ArchE's first step is to acquire the requirements for the system being designed. These include quality scenario requirements as well as functional requirements. ArchE treats these two types of requirements differently:

- ArchE translates the functional requirements into responsibilities and saves the latter in the responsibility concept. Because functional requirements, other than the derived responsibilities, are not treated further by ArchE, they are not one of the key concepts enumerated in Section 5.1. They are saved and referenced for traceability purposes from the responsibility concept.
- Quality scenarios, on the other hand, are a key concept because they help drive the actions of ArchE. The processing of the scenarios and the extraction of responsibilities from them is described in Section 5.2.2. Each scenario is entered into the scenario concept.

ArchE gathers a third type of requirement—constraints on the design—from the designer's specification during the Build Design activity, which is discussed in Section 5.2.5.

5.2.2 Step 2: Refine Scenarios

Once a raw scenario has been acquired, it must be refined into the component parts of a concrete scenario. That is, the six portions of a concrete scenario must be identified. ArchE enters into a dialogue with the designer to identify these parts that include a stimulus, a source of stimulus, an artifact, an environment, a response, and a response measure.

For one scenario of our example, the values are

- stimulus—obstacle detected
- source of stimulus—from external to the garage door
- artifact—garage door system
- environment—while the door is descending
- response—halt the descent of the door
- response measure—within .1 seconds

From these parameters, ArchE can identify two key responsibilities—"detect obstacle" and "halt the descent of the door"—and can achieve them in that order (at least in this example). The responsibilities and their relationships are represented in the responsibility concept. The scenario is marked as refined.

During the refinement, each scenario becomes a concrete instance of a general scenario. The designer specifies the quality attribute for which the scenario is a requirement, which allows ArchE to use a form-based approach to refine the scenario. In some cases, this choice of quality attribute dictates which reasoning framework ArchE uses to process the scenario because only one such framework exists for that quality attribute. In other cases, specifying the quality attribute limits the reasoning framework to one of several but does not dictate specifically which one to use. In our current plans for ArchE, specifying a quality scenario as a modifiability scenario will dictate the reasoning framework, whereas specifying a quality scenario as a performance one will only limit the possible reasoning frameworks (because we have a single reasoning framework for modifiability and two for performance).

5.2.3 Step 3: Choose Reasoning Framework

ArchE determines that the scenario is to be solved with a fixed-priority-scheduling reasoning framework. For the purposes of this report, the three reasoning frameworks include the fixed-priority-scheduling framework, the cyclic-executive reasoning framework, and the modifi-

ability framework. We expect ArchE to be extensible in reasoning frameworks, which introduces the problem of deciding which reasoning framework to use for particular scenarios. However, we do not deal with that problem here.

For our example scenario, ArchE chooses the fixed-priority-scheduling framework. ArchE adds a value to the scenario concept indicating that this scenario has been attached to a reasoning framework.

5.2.4 Step 4: Build Quality Attribute Models

A quality attribute model is a fully instantiated instance from a reasoning framework. In a previous report, we discussed how parameters for a model are chosen—some being bound by the scenario and others free to be chosen through the use of architectural tactics [Bachmann 03]. Once all the parameters have been bound and the response measure has been satisfied, the resulting parameters account for the values of the quality attribute model.

For our example scenario, the final values of the parameters for a scheduling model are

- arrival period—sporadic
- execution time—35 msec.
- priority—high
- processors—one

The designer may specify decisions that constrain the choice of parameters. For example, if the designer chooses an operating system that has a maximum of 32 threads, the fixed-priority-scheduling reasoning framework must respect that constraint. In a normal flow, a quality attribute model contributes to the design of the system. If a constraint is specified, it must be translated back into quality-attribute-model terms. In this report, we do not describe how this task is accomplished.

If ArchE is simultaneously solving multiple scenarios, it generates multiple quality attribute models (one for each scenario) that are consistent. That is, for those responsibilities involved in multiple models, the parameters associated with the models are consistent. For example, a single responsibility cannot have more than one priority in a scheduling model, regardless of the number of scenarios in which it participates. If ArchE is unable to generate a consistent quality attribute model, it reports failure to the designer.

The key concept QualityAttributeModel holds all the information associated with the quality attribute models.

5.2.5 Step 5: Build Design

Once a model is created, ArchE constructs a design. A model consists of parameters associated with quality attributes. A design consists of architectural elements and their properties. For example, the fixed-priority-scheduling reasoning framework generates parameters that consist of responsibilities and their priority within a priority-scheduling discipline. The modifiability reasoning framework generates parameters that consist of responsibilities, their assignments to modules, and the dependencies (and their type) among the modules. To convert these parameters into a design, the modules must be assigned to either threads or processes, which, in turn, have scheduling priorities. The types of dependencies must be converted into information flow and interfaces among the modules.

For our example scenario, the design consists of two processes with different priorities. Within one process is a module that has the responsibilities "detect obstacle" and "halt door," and within the other process is a module that has the responsibility "everything else."

Design is one of the key concepts, and it describes the system from an architectural point of view. The QualityAttributeModel concept describes the parameters derived from the various reasoning frameworks, and the design concept describes them from the architectural perspective. In many cases, this information might be the same, but in others, some transformations will occur (e.g., from scheduling priorities to threads or from dependencies to information flow).

6 Interaction Between Key Concepts and ArchE Activities

Table 1 summarizes the key concepts and the interaction between them and ArchE activities.

Table 1: Key Concepts and How Activities Access Them

Concept	Accessed by activity	Action	When
Scenario	Acquire Requirement	Raw scenarios input into ArchE	At initial entry or on request from designer
	Refine Scenario	Raw scenarios refined into six-part form Quality attribute identified	After acquiring requirements based on selection by the Planner Module
	Build Quality Attribute Models	Test whether current model satisfies scenario.	Must be done subsequent to changing any responsibility derived either from the scenario itself or responsibilities derived by applying tactics
	Refine Scenario	Modify existing scenario.	Designer modifies existing scenario at any point (likely after being told that some scenario is not satisfied by current model).
Responsibility	Acquire Requirement	Derived from specification formalism	On initialization or at designer's initiative
	Refine Scenario	Abstracted from refined scenarios	After a scenario is type checked

Table 1: Key Concepts and How Activities Access Them (cont'd.)

Concept	Accessed by activity	Action	When
Responsibility (cont'd)	Build Quality Attribute Models	Adds or refines responsibilities based on tactics used Also retrieves responsibilities in order to build models Accesses responsibilities through a "view" mechanism that allows each instance of a model builder to see only those relationships and parameters that are important to it	When a model has been constructed to satisfy a scenario; may involve interaction with designer (e.g., for abstract common services, the designer must specify which services)
	Build Design	Retrieves responsibilities from designs specified by the designer; also specifies constraints or values for the parameters of particular responsibilities	At the designer's initiative, a portion of the design may be specified. This portion may include new responsibilities.
Design	Build Design	Constructs design from existing model and displays it to the designer Designer also may specify portions (or all) of the design.	Either when a complete model has been constructed or at the designer's initiative
QualityAttribute-Model	Build Quality Attribute Models	Saves parameters of quality attribute models	When models are being constructed
	Build Design	Retrieves parameters of models as a portion of building the architecture	When design is being displayed or exported

7 User Interactions with ArchE

We describe the interactions of two types of users with ArchE—the designer who is using ArchE to help construct a design and the system maintainer who is extending ArchE by adding a new reasoning framework or additional rules within an existing reasoning framework.

7.1 Designer's Interactions with ArchE

We divide our treatment of how the designer interacts with ArchE into the five activities we discussed in Section 5. We begin with a discussion of a base set of interactions that are possible within any of the other activities.

7.1.1 Basic Interactions

At any point during the preparation of a design, the designer has the option to save the current state of the design for future work; restore the design from a previously saved state; inspect the current state of the design and the rationale for particular portions of the design; specify aspects (or all) of the design; and import, export, or modify existing scenarios or requirements.

Specifying design aspects allows the designer to perform multiple types of activities that include the following:

- Give a meaningful name to a set of requirements. As we saw in Section 5.2, ArchE treats responsibilities by understanding their decompositions and relationships. Sets of responsibilities should have meaningful names such as "sensor manager" rather than "responsibilities both in obstacle and user-input detection." The designer can assign these names.
- Specify values of parameters. The designer can do this for any of the reasoning frameworks. Parameters such as execution time for responsibilities, cost of modification of responsibilities, and so forth must be specified for the Build Quality Attribute Model activity, but the designer may specify them at any time.
- Specify constraints. Constraints on the design (such as "use Windows CE") manifest themselves as names of sets of responsibilities and as limits on parameters values. Some of these parameters are manifested during quality-attribute-model building. Windows CE, for example, allows a maximum of 32 threads. The designer specifies this limitation,

which the Build Quality Attribute Model activity must respect. Another example of a constraint occurs when processes can have only a single thread. Such constraints are manifested both during the building of the design and the building of the model by the reasoning frameworks.

7.1.2 Acquire Requirements

Requirements may come into ArchE from a variety of sources and in a variety of forms. The designer during this activity specifies the types of requirement sources and the forms in which the requirements arrive. Requirements may come in textual form (either from the designer or from some external source) or in stylized form (such as state diagrams). In any case, during the Acquire Requirements activity, the designer controls the input of requirements.

The designer also inputs raw quality attribute scenarios during this activity. Again, they may come from a variety of sources including direct interaction with ArchE, a text file, or an external program.

7.1.3 Refine Scenarios

Requirements must be refined into the six-part stylized form required by ArchE. The interaction with the designer during the Refine Scenarios activity turns the raw requirements coming from the Acquire Requirements activity into six-part scenarios. The quality attribute type of the scenario also is specified as one of a list of allowable quality attributes.

7.1.4 Choose Reasoning Framework

The designer can specify the reasoning framework to solve each scenario. Initially, the only reasoning frameworks included are modifiability, cyclic executive, and fixed-priority scheduling. ArchE uses a set of rules to choose the reasoning framework, although the designer may assist ArchE in making this choice when ArchE cannot decide unambiguously.

7.1.5 Build Quality Attribute Models

Three types of interaction occur between ArchE and the designer during the Build Quality Attribute Model activity: specifying parameters, assisting ArchE to choose tactics, and reporting the inability of ArchE to satisfy particular scenarios.

- Parameters within a quality attribute model are either bound or free. Parameters can be bound by the current state of the design or by the designer's specification during this activity.
- The quality-attribute-model builder explores possible tactics to generate a solution. This exploration may involve the designer. For example, the modifiability model builder may ask, "Are there common services that can be abstracted between these two sets of responsibilities. If yes, what are they?" This question requires the designer to specify a set of common services by name and by responsibilities and, possibly, to rename the original two sets of responsibilities.
- If ArchE is unable to generate a solution for a particular scenario, the designer is informed and the reasons for this inability are presented. This presentation may be an enumeration of the tactics attempted and the results of each parameterization, or it may take some other form. In any case, the designer can then modify one of the scenarios (maybe the one that failed, maybe another one) and ArchE will attempt to generate the design again. The designer likely will relax the response measure but may specify one of the parameters or change the stimulus.

7.1.6 Build Design

The Build Design activity is invoked at ArchE's initiative when all scenarios are satisfied or at the designer's initiative. In either case, ArchE must then create the design from the existing model. The model may not be complete since the designer may have requested an examination of the design when a scenario is only partially processed. For example, a modifiability scenario may have introduced new responsibilities, but the designer requests an inspection of the design before the Build Quality Attribute Models activity has completed its work on the fixed-priority-scheduling model.

The Build Design activity can fail only when a constraint is specified. If no constraint exists, the design should be achievable. If a design does fail, ArchE specifies a restriction on parameter values and retries the design. The designer becomes involved only when ArchE cannot determine the restrictions.

The designer also may specify a portion of the design or make choices among design alternatives for a number of reasons:

- A constraint exists to use existing assets such as legacy or COTS software.

- ArchE is unaware of certain quality attribute considerations. A security or reliability reason may exist for making a particular decision, but ArchE does not have the appropriate reasoning frameworks to generate that design decision.
- The model solvers within ArchE are inaccurate for some reason. For example, we use a crude cost model to evaluate modifiability hypotheses. The designer may recognize that ArchE made a mistake and rectify it.

7.2 System Maintainer's Interactions with ArchE

The system maintainer interacts with ArchE to extend and modify three different types of entities:

1. rules. The system maintainer modifies and adds rules. The current development environment maintains rules in Rational Rose and uses a Visual Basic for Applications script to generate the rules in the syntax of the selected rule engine—Jess.
2. reasoning frameworks. Additional reasoning frameworks will be added to ArchE. Adding a reasoning framework involves modifying the modules that implement the Choose Reasoning Framework activity; creating a model builder for that reasoning framework; integrating the model building into the modules that implement the Build Quality Attribute Models activity; and adding the rules pertinent to these two modules.
3. import/export capabilities. Additional import/export capabilities will be added for different types of information in the key ArchE concepts, including design information, relationships between items in the concepts (e.g., scenarios), and new items (e.g., business goals).

8 Conclusions

Our ultimate goal is to enable the design of systems with predictable quality attribute behavior. We claim that one method of achieving this goal is to have a collection of parallel and interacting reasoning frameworks that are each specialized for a single quality attribute but that work together to enable the creation of a design. In this report and prior ones, we presented evidence to support this claim.

In this report, we

- elaborated our theory to cover multiple scenarios with different types of response measures
- presented initial thoughts for an expert system to support the application of this theory

The elaboration of the theory reveals two different types of scope problems: the number of reasoning frameworks that must interact and the depth of detail necessary to use them. For both reasons, we argue that some type of intelligent tool support is necessary to make our theory operational.

Consequently, in addition to expanding the theory, we need to construct a prototype intelligent support tool to demonstrate that the theory will, in fact, lead to predictable designs. Hence, we have focused on the design of such a tool.

We hope to follow this path in the future:

- We construct our prototype tool—ArchE—and use it to demonstrate that for systems whose quality attribute requirements can be restricted to modifiability and real-time performance, the tool supports a designer in a meaningful way and helps a designer achieve a design as good or better than would have been possible without the tool. We, as of yet, have not considered how to measure the quality of the design produced by ArchE.
- We demonstrate that ArchE is useful with systems whose quality attribute requirements are broader than modifiability and real-time performance but that include those attributes. Thus, although ArchE initially will not support the design of a system with predictable security behavior, for example, it will support a system design where one quality attribute concern is security, and the other is modifiability or real-time performance.
- We extend the number and accuracy of the reasoning frameworks that are incorporated into the tool. The number of quality-attribute reasoning frameworks that exist for archi-

tecture design is limited. Even for modifiability—one of the quality attributes most commonly seen in requirements—we felt the need to create a cost model that reflects architectural and not system-level decisions. Consequently, extending the number of reasoning frameworks is a long-term goal.

We believe that with ArchE, we can effectively demonstrate the potential of our theory and stimulate the quality-attribute-research communities to focus on creating reasoning frameworks that support the design of systems with predictable quality attribute behavior.

Appendix: Detailed Description of ArchE

Structure of ArchE

In this section, we present the internal structure of ArchE. ArchE will be constructed on top of the Jess platform [Friedman-Hill 03]. Rule-based systems, as we have discussed, effectively allow the type of complex interactions that characterize ArchE.

On the other hand, exactly because of the complex interactions that it permits, this type of platform can be very difficult to use when managing the flow of control. Jess supports the concept of *modules* and *focus* to manage this flow. Assigning the focus to a particular module restricts the executable rules to those in the current module with the exception of *auto-focus* rules. Those rules can be executed regardless of the current focus; we intend to use them only to handle exceptions.

In the structuring of ArchE, we intend to allow experimentation with a variety of different control strategies while ensuring that this experimentation does not cause extreme side effects. Jess maintains its data structures in a FactBase and uses a RuleBase to maintain the rules that operate on the FactBase. Figure 3 shows the basic architecture of ArchE, which is a blackboard architecture where the data repository is the FactBase, the rule engine is provided by Jess, and each rule set is triggered by data in the repository and places new data into the data set [Clements 03]. We call each rule set a *module* and describe the major ones in subsequent sections.

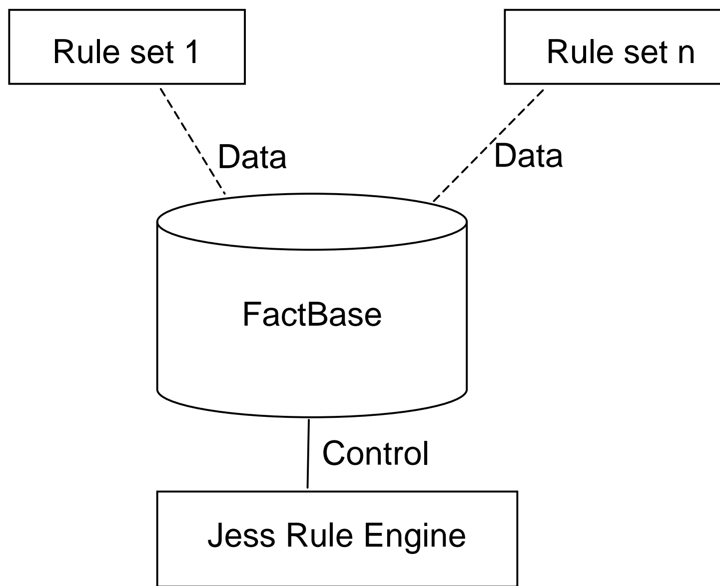

Figure 3: Blackboard Architecture of ArchE

We organize ArchE to achieve the goal of creating a design. This root goal has multiple subgoals that, in turn, have subgoals. Thus, ArchE can be viewed as navigating a goal tree to create the design. This terminology is used in the PRIDE system [Mittal 86]. Each goal is achieved by a separate module.

Goals “know” only about their immediate subgoals; no other subgoals are visible. All goals have access to the entire FactBase through the appropriate interface. Our intent is for ArchE to be easily modifiable. Adding new subgoals, rules, queries, and so forth should be simple.

In addition to the subgoals, each node has the following four types of rules:

1. designer interaction rules. Most interaction with the designer is intended to not block other ArchE activities, allowing the designer to provide ArchE with multiple types of information at once, from either a file or direct interaction. The designer can change any piece of information at any time.
2. sequencing rules. These rules control the sequencing and setting of focus on the subgoals. They act on data in the FactBase but do not modify it.
3. computation rules. These rules do the actual work of the node and modify the FactBase.
4. out-of-focus rules. These rules move focus to a different portion of ArchE. Focus normally is managed as a stack. When a node has no more rules to activate, it is removed from the focus stack and the new top of the stack has focus. These rules violate that stack control. Examples of such rules are exceptions that might be raised by the node. Out-of-focus rules are executed (if their conditions are true) regardless of the current focus.

In subsequent sections, we describe the action of the ArchE modules in more detail. We illuminate each module with examples that are based on the rules listed above.

Our Example Syntax

We present this example in a syntax that we hope is understandable by readers somewhat familiar with rule-based systems. As Jess determines the actual syntax for an implementation, this example syntax shows the intent to the reader, but will not actually be used in any application. The syntax, however, is intended to mimic the way that rule-based systems actually work. These systems are driven off a database (the FactBase), and the rules contain conditions that cause them to be executed if the condition is true, as in the following example:

```
If (Scenario.type="modifiable") then Print Scenario.text
```

The Print statement executes exactly as many times as scenarios of type “modifiable” occur in the FactBase. This rule, by itself, prints all the modifiability scenarios in the FactBase. The flow of control for a rule-based system is embedded in the conditions for the rules; control statements have no explicit flow. When the condition for a rule is true, the rule is executed according to the particular data in the FactBase that caused it to activate. For example, Scenario.text gets printed when the scenario type is “modifiable.”

The conditions of the rules implicitly determine the flow of control; therefore, in those cases when a specific flow of control is desired, special provision is made. Also, for those cases where the condition is based on some derivation from the FactBase rather than the explicit data in the FactBase, special provision is made. For example, a rule should activate if the FactBase contains more than 10 items with a particular value. Rule-based systems usually have some provision for querying the FactBase to determine ad hoc relationships. We assume this query provision in our syntax as well.

Not only does the syntax in our example differ from what will be true in practice, but the conditions do as well. The conditions are not drawn directly from the FactBase. For example, the FactBase does not contain a field called “RawScenarios.” Instead, this information is inferred from the existence of scenarios with incomplete portions. Again, we present the example in this fashion for readability and assume that the interested reader can make the translation between the conditions we present and the actual field in the FactBase.

Planner Module

Goals Achieved by the Planner Module

The Planner module is the root node of the goal tree. Its goal is to control the whole design process, with three types of activities:

1. It gathers requirements in terms of scenarios and interacts with the designer to produce and maintain a prioritized list of quality attribute scenarios.
2. It "scrutinizes" the list of scenarios, one group at a time, and augments the evolving design (or starts a new design) in a way that will satisfy the scenario while looking for interactions between scenarios.
3. It manages global interaction with the designer. At any point during the execution of ArchE, the designer can indicate that more scenarios are present and view or specify portions of the design.

Subgoals and Sequencing Rules for the Planner Module

The Planner module has the following subgoals:

- AcquireRequirements
- RefineScenario
- ChooseReasoningFramework
- BuildQualityAttributeModel
- BuildDesign

One "path" through the Planner module begins with the acquisition of a set of requirements, either *en masse* or one at a time from the designer. This acquisition occurs in the AcquireRequirements module and results in a list of raw scenarios. The Planner module then takes the scenarios and refines them into well-formed scenarios in the RefineScenario module. The RefineScenario module determines what types of scenarios are in the current collection. This information is subsequently used by the ChooseReasoningFramework module to select the relevant reasoning frameworks. The Planner module uses an instance of the BuildQualityAttributeModels module to either start or continue building the quality attribute model for each set of relevant scenarios. And then the BuildDesign module translates the model elements, relations, and properties into design elements, relations, and properties. Finally, the designer sees the design and determines its disposition.

We expect over time to experiment with a variety of different paths or sequences through these subgoals. Initially, we will implement the path described above and expect to have the following rules in priority order. The priority of rules determines which one is to be executed if multiple rules are eligible.

```
If (UnreadScenarios) then AcquireRequirements
If (RawScenarios) then RefineScenario
If (UnAttachedtoRF) then ChooseReasoningFramework
If (UnmodelledScenarios) then BuildQualityAttributeModels
If (UnDesignedScenarios) then BuildDesign
```

We might use the sequencing rule of acquiring scenarios whenever ArchE has completed all its current work (rather than assuming as we did that all scenarios are available when process-

ing begins and that when they are processed, ArchE is finished). The following syntax example indicates this rule:

```
If (ArchECompleted) then AcquireRequirements
```

This rule is represented in Jess syntax as the following:

```
;
; Every time when nothing else can be done ArchE checks if new
; requirements have been given by the user.
;
(defrule CheckForNewRequirements
    (declare (auto-focus TRUE))
    ()
=>
    (focus AquireRequirements)
```

Rules Manipulating the FactBase

The Planner module contains no rules that manipulate the FactBase.

Out-of-Focus Rules

Some of the sequencing rules might be auto-focus rules and outside the normal sequencing. In particular, we anticipate that if the designer indicates a desire to examine the responsibilities, the DisplayResponsibilities utility will be invoked via an out-of-focus rule.

Example

Since the Planner module does not manipulate the FactBase, we have nothing to demonstrate with our sample scenarios.

1. The garage door must detect an obstacle and halt within .1 seconds.
2. Produce products based on a different processor.
3. The garage door must respond to a user request to lower the door within .3 seconds.

Acquire Requirements Module

Goal Achieved by the AcquireRequirements Module

This module inputs requirements in various forms and places them into the FactBase. They might take the form of proto-scenarios (not necessarily well formed), feature trees, or state charts.

Subgoals and Sequencing Rules for the AcquireRequirements Module

This module has the following subgoals:

- AcquireRawScenarios
- AcquireFunctionalRequirements

The sequencing rules for these subgoals are

```
If (UnreadScenarios) then AcquireRawScenario
If (UnreadRequirements) then AcquireFunctionalRequirements
```

These rules execute if AcquireRequirements has the focus. Thus, during experimentation with general sequencing policies, these rules will not need to be modified.

Rules Manipulating the FactBase

We visualize both subgoal modules as being procedural rather than rule based, and we describe their actions with respect to the FactBase.

The AcquireFunctionalRequirements module retrieves the functional requirements in whatever form specified, abstracts the responsibilities from these requirements, and enters the responsibilities into the Responsibility data structure in the FactBase. These responsibilities may be reviewed by the designer for determination of relationships among the responsibilities. The AcquireFunctionalRequirements module can indicate to the designer that new responsibilities have arrived by presenting the designer with an indicator, and the designer can then review them when desired.

The AcquireRawScenarios module retrieves raw scenarios (in the form of the example scenarios) and marks them as being “Raw” in the Scenario data structure in the FactBase.

Out-of-Focus Rules

The AcquireRequirements module raises no exceptions and has no out-of-focus rules. Problems such as an incorrect file name for input or an incorrect format of requirements are handled within the module.

Example

The AcquireRequirements module reads in the four scenarios in the form given above.

Refine Scenario Module

Goal Achieved by the RefineScenario Module

This module converts a raw scenario (possibly ill formed) into a refined scenario (with all six portions) and classifies it as belonging to a particular quality attribute. Interaction of the responsibilities with the other responsibilities is also (potentially) determined by the designer.

Subgoals and Sequencing Rules for the RefineScenario Module

This module has the following subgoals:

- DetermineScenarioType
- IdentifyScenarioParts
- IdentifyResponsibilities
- DisplayResponsibilities

The sequencing rule for these goals is

```
If (RawScenarios) then DetermineScenarioType; IdentifySce-
narioParts; IdentifyResponsibilties; DisplayResponsibilities
```

When the IdentifyScenarioParts module identifies new responsibilities, the DisplayResponsibilities module is invoked so that the designer can specify how these new responsibilities interact with existing ones.

Rules Manipulating the FactBase

The RefineScenario module does not have any rules that manipulate the FactBase. However, in the example below, we show the rules associated with the DetermineScenarioType module and the IdentifyScenarioParts module that present the scenario to the designer, ask the designer to classify the scenario with respect to a quality attribute type, and then identify the

responsibilities. The DisplayResponsibilities module is a utility module that displays all responsibilities and their interactions to the designer, and provides an opportunity to modify names of the responsibilities or identify relationships among the responsibilities.

```
If (TRUE) then
   QueryUserforQualityAttribute;
   ParseScenario;
   IdentifyResponsibilities;
```

The result of the RefineScenario module is the completion of portions of the parsed scenario in the FactBase.

Out-of-Focus Rules

This module has no out-of-focus rules.

Example

After parsing the first scenario but before parsing the second, the FactBase has the following entries:

FactBase.Scenarios:
Scenario(1)

- Scenario(1).Raw = "The garage door must detect an obstacle and halt within .1 seconds"
- Scenario(1).State = "Parsed"
- Scenario(1).Quality = "Performance"
- Scenario(1).Type = ""
- Scenario(1).Stimulus = "Detect obstacle"
- Scenario(1).Stimulus.Type = "Sporadic"
- Scenario(1).Source = "Garage door pressure sensor"
- Scenario(1).Context = "Detect that the door is descending"
- Scenario(1).Response = "Halt door"
- Scenario(1).Response_Measure = "within a deadline of.1 seconds"

Scenario(2)

- Scenario(2).Raw = “Produce products based on different processor”
- Scenario(2).State = “Unparsed”
- Scenario(2).Quality = “”
- Scenario(2).Type = “”
- Scenario(2).Stimulus = “”
- Scenario(2).Source = “”
- Scenario(2).Context = “”
- Scenario(2).Response = “”

At this point, the Responsibility data structure also is updated as follows:

- Responsibility(1).responsibility = Scenario.Stimulus(1)
- Responsibility(2) .responsibility = Scenario.Context(1)
- Responsibility(3) .responsibility = Scenario.Response(1)

Every scenario automatically creates these three responsibilities. Since responsibilities can be decomposed and aggregated, and can have precedence relationships, the designer is notified of new responsibilities and has the opportunity to specify the relationship between these new and existing responsibilities.

FactBase.Responsibilities:
Responsibility(1)

- Responsibility(1).responsibility = “Detect obstacle”
- Responsibility(1).parent(1) = “”
- Responsibility(1).child(1) = “Read pressure sensor”
- Responsibility(1).child(2) = “Determine if threshold is exceeded”
- Responsibility(1).ExecutesAfter = “”
- Responsibility(1).ExecutesBefore = “Halt door”

ChooseReasoningFramework Module

Goal Achieved by the ChooseReasoningFramework Module

This module determines the build-model type to process each scenario.

Subgoals and Sequencing Rules for the ChooseReasoningFramework Module

This module has the following subgoals:

- DetermineScenarioType
- AssignReasoningFramework

The sequencing rules for these goals are

```
If (noScenariotype) then DetermineScenarioType
If (noReasoningFramework) then AssignReasoningFramework
```

We first determine the type of the scenario, which currently includes “hard-deadline” and “cost of modification.” The type depends on the response measure of the scenario. Then based on the type, we assign the reasoning framework. Currently, the only cost of the modification reasoning framework is “modifiability.” We have two hard-deadline reasoning frameworks—fixed-priority scheduling and cyclic executive—and a set of rules in the AssignReasoningFramework module that differentiates between these two.

As the number of reasoning frameworks grows, these two modules will become progressively more sophisticated in the rules they use to assign a reasoning framework to a scenario.

Rules Manipulating the FactBase

This module has no rules that manipulate the FactBase.

Out-of-Focus Rules

This module has no out-of-focus rules.

DetermineScenarioType Module

Goal Achieved by the DetermineScenarioType Module

This module is invoked with a scenario and then determines the scenario type. Currently, the only type defined for performance scenarios is “hard-deadline,” and the only one for modifiability scenarios is “modifiability.”

Subgoals and Sequencing Rules for the DetermineScenarioType Module

This module has no subgoals and no sequencing rules.

Rules Manipulating the FactBase

```
If (scenario.quality = "Performance") then scenario.type = "hard-
deadline"
If (scenario.quality = "modifiability") then scenario.type  =
"modifiability"
```

Out-of-Focus Rules

This module has no out-of-focus rules.

AssignReasoningFramework Module

Goal Achieved by the AssignReasoningFramework Module

When invoked, this module examines all the scenarios to determine their type. Those of type “modifiability” are assigned to the modifiability reasoning framework. Those of type “hard-deadline” are assigned either to the cyclic-executive reasoning framework or the fixed-priority-scheduling reasoning framework.

The query facility in Jess offers one method for dealing with groups of scenarios. It searches the FactBase and returns all items that satisfy the criteria.

Subgoals and Sequencing Rules for the AssignReasoningFramework Module

This module has no subgoals and no sequencing rules.

Rules Manipulating the FactBase

```
If(TRUE)then
   For each in (QueryFactBase(Scenario, Scenario.type-
   "modifiability"))
   Scenario.ReasoningFramework = "modifiability"

If (TRUE) then
   If (count of (QueryFactBase(Scenario, Scenario.type = "hard-
   deadline")) >= 10)
      then
   For each in (QueryFactBase(Scenario, Scenario.type = "hard-
   deadline"))
   Scenario.ReasoningFramework = "Fixed-priority-scheduling"

If (count of (QueryFactBase(Scenario, Scenario.Stimulus.Type =
   "Sporadic") > 0 ) then
   For each in (QueryFactBase(Scenario, Scenario.type = "hard-
   deadline"))
   Scenario.ReasoningFramework = "Fixed-priority-scheduling"

If (count of (QueryFactBase(Scenario, Scenario.Stimulus.Type =
   "Stochastic") > 0 ) then
   For each in (QueryFactBase(Scenario, Scenario.type = "hard-
   deadline"))
   Scenario.ReasoningFramework = "Fixed-priority-scheduling"

If (Scenario.ReasoningFramework = "" AND "Scenario.type = "hard-
deadline) then
   Scenario.ReasoningFramework = "cyclic-executive"
```

Out-of-Focus Rules

This module has no out-of-focus rules.

Example

Scenario(1).ReasoningFramework = “fixed-priority-scheduling”
Scenario(2).ReasoningFramework = “fixed-priority-scheduling”
Scenario(3).ReasoningFramework = “modifiability”
Scenario(4).ReasoningFramework = “modifiability”

BuildQualityAttributeModel Module

Goal Achieved by the BuildQualityAttributeModel Module

When invoked, this module creates a model that satisfies all the scenarios.

Subgoals and Sequencing Rules for the BuildQualityAttributeModel Module

This module has the following subgoals:

- BuildFixedPrioritySchedulingModel
- BuildCyclicExecutiveSchedulingModel
- BuildModifiabilityModel

The sequencing rules for these subgoals are

```
If ( notconsistentmodel AND notinfiniteloop) then
   BuildFixedPrioritySchedulingModel
   BuildCyclicExecutiveSchedulingModel
   BuildModifiabilityModel
```

Rules Manipulating the FactBase

This module has no rules that manipulate the FactBase.

Out-of-Focus Rules

This module has no out-of-focus rules.

BuildFixedPrioritySchedulingModel Module

Goal Achieved by the BuildFixedPrioritySchedulingModel Module

This module assigns properties to the responsibilities associated with hard-deadline scenarios so that a design conforming to those properties will achieve the scenario deadlines.

Subgoals and Sequencing Rules for the BuildFixedPriorityScheduling-Model Module

This module includes the following subgoals:

- DetermineBoundParametersforFixedPrioritySchedulingModel
- InitialValuesofFreeParametersforFixedPrioritySchedulingModel
- EvaluateFixedPrioritySchedulingModel
- CreateFixedPrioritySchedulingModel

The sequencing rules are

```
If (FixedPrioritySchedulingResponsibilitiesIdentified) then
```

```
    DetermineBoundParametersforFixedPrioritySchedulingModel
    InitialValuesofFreeParametersforFixedPrioritySchedulingModel
If (SchedulingModelNotSatisfied and FixedPrioritySchedulingParam-
etersHaveValues) then
    CreateFixedPrioritySchedulingModel
    EvaluateFixedPrioritySchedulingModel
```

Rules Manipulating the FactBase

```
//
// The first rule sets the context for building the models.
// This retrieves all of the
// responsibilities associated with the hard-deadline
// scenarios and makes this the current
// context for the building of the model. It is only invoked
// when this module is in focus
//
If (TRUE) then Bind(QueryFactBase(Responsibilities deriving from
Scenario.RF=”fixed-priority-scheduling”))

// if any responsibilities are not assigned a priority during the
// model building—i.e., those
// not included in a fixed-priority-scheduling scenario—they are
// given lowest scheduling
// priority

If (Responsibility.schedulingpriority = “”) then Responsibil-
ity.schedulingpriority =
                        “lowest”
```

Out-of-Focus Rules

An exception is raised if a suitable model cannot be found.

Example

Responsibility

- Responsibility(1).responsibility = “Detect obstacle”
- Responsibility(1) ExecutionTime = 10ms
- Responsibility(1).schedulingpriority = 100
- Responsibility(3).responsibility = “Detect user request”
- Responsibility(3).ExecutionTime = 25ms
- Responsibility(3).schedulingpriority = 90

CreateFixedPrioritySchedulingModel Module

In the BuildFixedPrioritySchedulingModel, the only subgoal not obvious is the creation of the next fixed-priority-scheduling model to be tested. We present this module and omit the others.

Goal Achieved by the CreateFixedPrioritySchedulingModel Module

This module creates parameters for a fixed-priority-scheduling model that subsequently can be evaluated to determine whether it satisfies the constraints of the hard-deadline scenarios. The rules presented here are probably not complete and will need to be augmented to cover all the different cases that might arise.

Subgoals and Sequencing Rules of the CreateFixedPriorityScheduling-Model Module

This module has no subgoals.

Rules Manipulating the FactBase

```
if  (NOT UnsetParametersforFixedPriorityScheduling) AND
      GetSumofExecutionTimeofDeadlineScenarios LTE
      GetMinDeadlineofDeadlineScenarios) then
      AllocateScenariostoSingleUnitofConcurrency
      AllocateUnitsofConcurrencytoSingleProcessor
else
   ApplyTacticIncreaseLogicalConcurrency
   AllocateUnitsofConcurrencytoSingleProcessor

if (NOT UnsetParametersforFixedPriorityScheduling AND
   GetSumofExecutionTimeofDeadlineScenarios GT
   GetMinDeadlineofDeadlineScenarios)) then
   ApplyTacticBoundExecutionTime
```

Out-of-Focus Rules

This module has no out-of-focus rules.

Example

For our example, the sum of the execution times of the responsibilities is greater than the deadlines, so the first portion of the first rule does not activate, but the “else” portion does. The tactic “increase logical concurrency” creates units of concurrency. The responsibilities are then allocated to these units in a systematic fashion. The last rule also does not activate for our example.

BuildModifiabilityModel Module

Goal Achieved by the Modifiability Model Module

This module creates a modifiability model for the modifiability scenarios. It assigns all responsibilities added from other reasoning frameworks to a single module.

Subgoals and Sequencing Rules of the BuildModifiabilityModel Module

This module includes the following subgoals:

- DetermineBoundParametersforModifiabilityModel
- InitialValuesofFreeParametersforModifiabilityModel
- EvaluateModifiabilityModel
- CreateModifiabilityModel

The sequencing rules are

```
If (ModifiabilityResponsibilitiesIdentified) then
   DetermineBoundParametersforModifiabilityModel
   InitialValuesofFreeParametersforModifiabilityModel

If (ModifiabilityScenariosNotSatisfied and
   ModifiabilityParametersHaveValues) then
   CreateModifiabilityModel
   EvaluateModifiabilityModel
```

Rules Manipulating the FactBase

```
//
// The first rule sets the context for building the models. This
// retrieves all of the
// responsibilities associated with the modifiability scenarios
// and makes this the current
// context for the building of the model. It is only invoked when
// this module is in focus
//
If (TRUE) then Bind(QueryFactBase(Responsibilities deriving from
Scenario.RF=”modifiability”))

// if any responsibilities are not assigned a priority during the
// model building—i.e., those
// not included in a fixed-priority-scheduling scenario—they are
// given lowest scheduling
// priority

If (Responsibility.moduleassignment = “”) then Responsibil-
ity.moduleassignment =
                        “other”
```

Out-of-Focus Rules

An exception is raised if a suitable model cannot be found.

Example

Responsibility

- Responsibility(4).responsibility = “processor independent”
- Responsibility(4) module = “processor independent”
- Responsibility(5).responsibility = “convert from processor independent to processor dependent”
- Responsibility(5).module = “virtual machine”

EvaluateModifiabilityModel Module

ArchE uses a cost model (admittedly crude) to estimate the cost of making the changes specified by the scenarios under consideration. The cost model has the following assumptions:

- The cost of making the specified changes to particular responsibilities can be estimated either by the designer or by a process that does not involve designer input.
- The side effects and the ripple effects of a change are captured by the type of change and the dependencies among the modules. Specifically, ArchE uses the following rules to estimate the cost of a particular change:
  - The designer provides the cost of changing the specific responsibilities associated with the modification.
  - Add the cost of changing any other responsibilities related to the specific responsibilities being modified (with an associated probability of affecting these co-located responsibilities).
  - Add the cost of changing any responsibilities in other modules that are dependent on the modules being modified. The rippling of a change to a dependent module depends on the type of change, the type of dependency, and the probability of changes of this type being propagated along the specific type of dependency.

CreateModifiabilityModel Module

We only sketch the algorithm that is used to create the modifiability models. This algorithm (and the rules that implement it) certainly will evolve as we gain experience with using ArchE.

Goal Achieved by the CreateFixedPrioritySchedulingModel Module

This module creates parameters for a modifiability model that subsequently can be evaluated to determine whether the model satisfies the constraints of the modifiability scenarios.

Subgoals and Sequencing Rules of the CreateModifiabilityModel Module

This module has no subgoals.

Rules Manipulating the FactBase

```
// if the sum of the cost of modifying the responsibilities
// that are directly derived from the
// modifiability scenarios is greater than the response
// measure than the scenario cannot
// be satisfied

If (Scenario.responsibility.cost > Scenario.responsemeasure) then
ReportFailure

// The following rules test the various modifiability tactics.
// They look for large
// contributors to the cost. We arbitrarily define a
// large contributor as 10% of
// the total cost but this is merely a rule of thumb

If (MultipleResponsibilitiesAssignedSameModule AND
   ResponsibilityInteraction GT 10% * TotalCost) then
      ApplyTacticSemanticCohesion

If (RippleCostFromModuleAtoB GT 10% * TotalCost) then
   ApplyTacticInsertIntermediary
   ApplyTacticHideInformation

If (CostSingleResponsibilityModification GT 10% * TotalCost) then
   ApplyTacticRaiseAbstractionLevel

If (ResponsibilityOverlap) then ApplyTacticAbstractCommonServices
```

Out-of-Focus Rules

This module has no out-of-focus rules.

BuildDesign Module

Goal Achieved by the BuildDesign Module

This module constructs a design from the parameters of the responsibilities. For example, one such parameter is the module to which the responsibility is assigned. A design is module-centric rather than responsibility-centric, so a module has responsibilities rather than a responsibility having a module. Furthermore, modules are packaged into processes by the BuildDesign module in conformance with the concurrency units.

Once a design is constructed, it can be exported to a variety of recipients. One of these recipients will be responsible for presenting the design to the designer. Furthermore, the designer can specify portions of the design and the BuildDesign module can then import the specifications and convert them into properties of responsibilities.

The BuildDesign module also may export the design to external entities for analysis or other purposes.

Utility Modules

Two important utility modules of ArchE are DisplayDesign and DisplayResponsibilities. Each one allows the designer to modify as well as observe the current state of the design, and each is intended to be highly interactive and provide sophisticated display and modification facilities.

- The DisplayDesign module displays the current state of the design to the designer. The design can be displayed at the designer's request or when ArchE is stable and all requirements or scenarios have been added. The designer has the opportunity to browse through the architecture using a variety of graphical input techniques.

  This module must understand the different views that ArchE constructs and how they are related. Modifications can occur through any view, and this module must translate those modifications to all relevant views.

- The DisplayResponsibilities module displays the current set of responsibilities and their relationships to the designer. The designer will have the opportunity to browse through and modify the information shown. Some of the modifications may include changing the name of particular responsibilities, decomposing responsibilities, or specifying the relationship among responsibilities (e.g., contained in).

Since a large number of responsibilities is likely, this module also will contain search and retrieval mechanisms for locating responsibilities.

References

URLs valid as of the publication date of this document

[Bachmann 02] Bachmann, Felix; Bass, Len; & Klein, Mark. *Illuminating the Fundamental Contributors to Software Architecture Quality* (CMU/SEI-2002-TR-025, ADA407778). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <http://www.sei.cmu.edu/publications/documents/02.reports/02tr025.html>.

[Bachmann 03] Bachmann, Felix; Bass, Len; & Klein, Mark. *Deriving Architectural Tactics: A Step Toward Methodical Architectural Design* (CMU/SEI-2003-TR-004, ADA413644). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tr004.html>.

[Clements 03] Clements, Paul; Bachmann, Felix; Bass, Len; Garlan, David; Ivers, James; Little, Reed; Nord, Robert; & Stafford, Judith. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley, 2003.

[Friedman-Hill 03] Friedman-Hill, Ernest. *Jess in Action: Java Rule-Based Systems*. Greenwich, CT: Manning Publications Company, July 2003.

[Giarrantano 98] Giarrantano, Joseph & Riley, Gary. *Expert Systems, Principles and Programming, Third Edition*. Boston, MA: PWS Publishing Company, 1998.

[Mittal 86] Mittal, Sanjay; Dym, Clive L.; & Morjaria Mahesh. "Pride: An Expert System for the Design of Paper Handling Systems." *IEEE Computer 19*, 7 (July 1986): 102 – 114.

[Pace 03] Pace, J. Andres Diaz & Campo, Marcelo R. "DesignBots: Towards a Planning-based Approach for the Exploration of Architectural Design Alternatives," 46 – 67. *Proceedings of the 2003 Argentine Symposium on Software Engineering (ASSE 2003).* Buenos Aires, Argentina: Argentine Society of Informatics and Operational Research (SADIO), September 2003.

[Wirfs-Brock 03] Wirfs-Brock, Rebecca & McKean, Alan. *Object Design: Roles, Responsibilities, and Collaborations*. Boston, MA: Addison-Wesley, 2003.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

Field	Content
1. AGENCY USE ONLY	(Leave Blank)
2. REPORT DATE	September 2003
3. REPORT TYPE AND DATES COVERED	Final
4. TITLE AND SUBTITLE	Preliminary Design of ArchE: A Software Architecture Design Assistant
5. FUNDING NUMBERS	F19628-00-C-0003
6. AUTHOR(S)	Felix Bachmann, Len Bass, Mark Klein
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213
8. PERFORMING ORGANIZATION REPORT NUMBER	CMU/SEI-2003-TR-021
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116
10. SPONSORING/MONITORING AGENCY REPORT NUMBER	ESC-TR-2003-021
11. SUPPLEMENTARY NOTES	
12A DISTRIBUTION/AVAILABILITY STATEMENT	Unclassified/Unlimited, DTIC, NTIS
12B DISTRIBUTION CODE	
13. ABSTRACT (MAXIMUM 200 WORDS)	This report presents a procedure for moving from a set of quality attribute scenarios to an architecture design that satisfies those scenarios. This procedure is embodied in a preliminary design for an architecture design assistant named ArchE (Architecture Expert), which will be implemented on a rule-based platform. This report includes the theory and rationale precipitating the design of ArchE and then describes this design in detail.
14. SUBJECT TERMS	architecture design, quality attribute scenarios, ArchE, rule-based platform, design assistant
15. NUMBER OF PAGES	66
16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT	Unclassified
18. SECURITY CLASSIFICATION OF THIS PAGE	Unclassified
19. SECURITY CLASSIFICATION OF ABSTRACT	Unclassified
20. LIMITATION OF ABSTRACT	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102