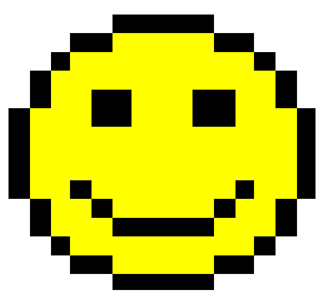
The MORALE Guidebook to the Evolutionary Design of Complex Software

Gregory Abowd
Ashok Goel
Michael McCracken
Melody Moore
Colin Potts
Spencer Rugaber
Linda Wills

Georgia Institute of Technology



ISBN 0-000-000000-0

ABCDEFGHIJ-DO-89

MORALE METHODOLOGY GUIDEBOOK
Preface

Problem: Nobody reads prefaces
Solution: Call the preface Chapter 1.
New problem created by solution: Chapter 1 is boring.
Resolution: Throw away Chapter 1 and call Chapter 2 Chapter 1.

—Donald Gause and Gerald Weinberg

Most software engineering books have it wrong. They give a thorough description of the steps in creating a software system, from initial conception through requirements gathering and analysis, on to design, implementation, and testing. Some even discuss delivery and post-deployment maintenance. But most software development does not work that way. Most software development is concerned with correcting, adapting, and enhancing existing software. This book tells the software development story from the point of view of software evolution—how a software system can grow and change in a controlled way over its lifecycle, how design deterioration can be avoided, how future decision making can take effective advantage of the analyses and design decisions made while developing previous versions, how the impact of changing requirements on the underlying system architecture can be assessed and used to avoid costly mistakes, and how the legacy value of legacy software can be understood and leveraged.

This book is for software engineers. A software engineer need not be a programmer. A software engineer is anyone who works on a software development project and whose interests go beyond the purely technical concerns of programming languages or test cases. A software engineer is concerned with the *process* of software development as well as the *product* being developed. A software engineer is concerned with the success of a product during all phases of its development and throughout its lifetime. Software engineers include software managers who have to make tough decisions like buy-versus-build, scrap-versus-renovate, slip-versus-ship.

The book assumes that you are engaged in software development as a day-to-day activity, that the software you are developing is large enough to require a team of developers, and that the software is intended to last long enough that the long-term benefits of documentation, version control, and configuration management are appreciated. Although we define our terminology before we use it, we also assume that your are fluent with the language of software development, either as the result of formal education in computer science or due to the hard-won experience of actually fighting software battles from the front lines.

The book is structured as a series of guidebooks, each covering a different aspect of the software evolution process. Each guidebook describes in concrete terms how to effectively engage in its part of the software development process. And each guidebook presents a complete, worked example of applying its technique to a real software evolution problem. The worked examples all consider the same underlying software system, the Mosaic world wide web browser. Mosaic has gone through many versions over several years. Each of the guidebooks looks at the evolution of Mosaic from version 2.4 to version 2.5 and, in particular, how a specific change made for version 2.5—the addition of user-configurable viewers—can be analyzed, judged, and implemented in a controlled way.

The techniques described the guidebooks constitute the MORALE software evolution process. *MORALE* is an acronym for **M**ission **Or**iented **A**rchitectural **L**egacy **E**volution. *Mission oriented* refers to MORALE's assumption that the changes being made to a system are fundamental to the system's purpose and users, rather than being a purely technical change such as adapting the program to a new version of the hardware on which it runs. *Legacy* refers to the fact that evolution of an existing version is being considered and that determination of the most effective way to use its resources is central to planning the new version. And *architectural* emphasizes that the alterations being contemplated may have profound effects on the basic structural and coordinational principles upon which it has been build.

The book begins with an overview of MORALE and the Mosaic case study. The techniques from each guidebook are introduced and how it deals with the evolution of Mosaic is summarized. There are five guidebooks. The first talks about mission-oriented requirements elicitation and analysis. It also introduces a theme that pervades MORALE—scenarios. Scenarios are sequences of activities that together describe one aspect of system execution behavior. Scenarios are also central to the second guidebook, which presents a way to analyze the effect of changes on a software architecture. Changes to a system architecture are costly, and unexpected architectural changes can spell doom to an otherwise successfully managed project. The third guidebook is concerned with legacy-system understanding, in particular, architectural extraction. That is, how can the underlying system components and their interactions be determined, even in the face of out-of-date or incomplete documentation, deteriorating structure, and overwhelming size and complexity.

Even if the mission-oriented requirements can be determined, the architecture extracted, and the impact of the former on the latter determined, there is still the question of how to actually make the change. The fourth guidebook is concerned with planning for these changes. In particular, it considers the question of how the system's overall goals are realized in its functional components and how changes to those goals affect the components and their interaction. The fifth guidebook is concerned with a specific kind of change, change to a system's user interface. Not only is effective user interface design a difficult and time consuming process, but its result often determine the ultimate customer acceptance of a product. This guidebook describes a technique for analyzing the user interface of an existing system version and transforming it to conform to its new requirements.

Following the guidebooks are several useful appendices. The first describes the MORALE tool suite, Esprit de Corps, as an illustration of how some of the techniques described in the guidebooks can be supported by automated tools. The tools range from an information system for managing scenarios, to a program analyzer, to a graphical visualization tool for architectures, to two knowledge-based tools for performing, respectively, program adaptation and user-interface migration. The second appendix is a glossary, collecting and synthesizing terminology from all of the guidebooks. References, however, are not collected, appearing instead at the end of each of the chapters.

MORALE itself evolved from the work of its individual contributors. Some of the work described here derives from earlier written descriptions co-authored by colleagues who are not part of the MORALE project. Nevertheless, we wish to express our appreciation to them both for their contributions and for their permission to reprint their work. Individual acknowledgment appear in each of the chapters. MORALE first took its current form in a proposal to the Defense Advanced Research Projects Agency (DARPA) of the U.S. Department of Defense. MORALE has been funded by DARPA since July 1996. The MORALE project is part of DARPAs Evolutionary Design of Complex Software program (EDCS) with John Salasin as its program manager. We would like to thank John, not only for his financial support but also for his constant encouragement to make our ideas tangible, practical, and of use to real developers. The MORALE team consists not only of its faculty authors but also its awesome graduate students without whom much of what you will read in this book would only be so many words. They have challenged us with their questions, stimutated us with their ideas, and supported us with their hard work. Finally, we would like to thank the College of Computing at Georgia Tech as a community in the true sense of the word encouraging collaboration, not only with its outward looking aspect, but at a fundamental, architectural level.

Research Methodology Report

MORALE
Mission Oriented Architectural Legacy Evolution

Gregory Abowd, Christian Ertmann-Christiansen, Ashok Goel, Dean F. Jerding, Michael McCracken, Melody Moore, J. William Murdock, Colin Potts, Spencer Rugaber[1], and Linda Wills
Georgia Institute of Technology

Abstract

Software evolution is the most costly and time consuming software development activity. Yet software engineering research is predominantly concerned with initial development. MORALE is a development method specifically designed for evolving software. It features an inquiry-based approach to eliciting change requirements, a reverse engineering technique for extracting architectural information from existing code, an approach to impact assessment that determines the extent to which the existing system's architectural components can be reused in the evolved version, a reflective approach to actually performing the evolution, and a specific technique for dealing with the difficulties that arise when evolving user interfaces. MORALE is described in the context of making a specific change to an existing system: adding user-configurable viewers to version 2.4 of the Mosaic web browser. Issues that arise are discussed and the Esprit de Corps tool suite is described.

Keywords: software evolution, software architecture, scenarios, reverse engineering, program visualization, user interface migration, adaptive design

I. Introduction

A. MORALE

The MORALE project addresses the problem of designing and evolving complex software systems. The MORALE acronym summarizes its goals:

- **M**ission **OR**iented: We want the legacy system enhancement process to be driven by the mission to be accomplished rather than by purely technical criteria.
- **A**rchitectural: The most time consuming and costly alterations to software are those that distort architecture, by which we mean its structure and patterns of component interaction. We want to provide a mechanism for predicting the impact of architectural changes so that the risks of making those changes can be ascertained early in the evolution process.
- **L**egacy **E**volution: We are concerned with the evolution of legacy systems. We want to provide a cost effective way of analyzing existing software, and once analyzed, extracting those parts of it which can be used in the new version.

MORALE addresses these goals by integrating several innovative technologies.

- A mission-directed requirements determination process that, through a systematic process of inquiry and refinement, turns mission-oriented goals into specifications of the desired behaviors of architectural components.
- An architectural assessment process that can ascertain the impact of new requirements on an existing system's architecture. Our interest here is in the analysis of architectural descriptions in order to determine how well they satisfy both functional and non-functional changes to

1. Point of Contact: Spencer Rugaber, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, USA, (404) 894-8450, (f) (404) 894-9442, spencer@cc.gatech.edu.

requirements.

- A reverse engineering process for extracting architectural information from an existing system. Beyond the traditional structural analyses provided by commercial and research tools, MORALE extracts behavior information derived from dynamic analysis of the message flow among architectural components.
- A software adaptation process that suggests how to go about making the changes designated by the analyses described in the previous three bullets.
- A tailored approach to the specific case of user interface evolution.

B. Research Context

This paper describes the MORALE project in the context of a specific software evolution scenario, the enhancement of the Mosaic web browser, version 2.4, with a new feature: user-configurable viewers. This feature was on the “wish-list” of features to be added to version 2.4 by the designers of Mosaic.

C. Approach

There is an unexpected symmetry between requirements analysis and reverse engineering that can be exploited to improve the process of mission-driven system evolution. Effective system evolution requires understanding both the way an existing system accomplishes its tasks and also the mission-oriented rationale for any changes that drive the evolution. The appropriate level at which to integrate these two sources of knowledge is the architectural. Understanding the higher level structuring, or architecture, of an existing system aids in predicting the impact of change that is mandated by new mission-oriented requirements. We use requirements analysis techniques to suggest what concepts are most useful in understanding how an existing system works and how it should be evolved. We use reverse engineering techniques to extract high level architecture, both in terms of static and dynamic analysis. The MORALE suite of techniques and tools harnesses this symmetry by growing a common model of the architecture for multiple versions of a system or system family. The common model is a basis for assessing the effects of proposed changes and the extent to which legacy code can be reused.

MORALE analysis can naturally be broken up into three activities, one each for requirements analysis, architectural extraction, and change impact analysis. The relationships between these activities and the artifacts relied upon and produced are depicted in the data flow diagram shown in Figure 1. The initial inputs to evolution are the existing system in the form of legacy source

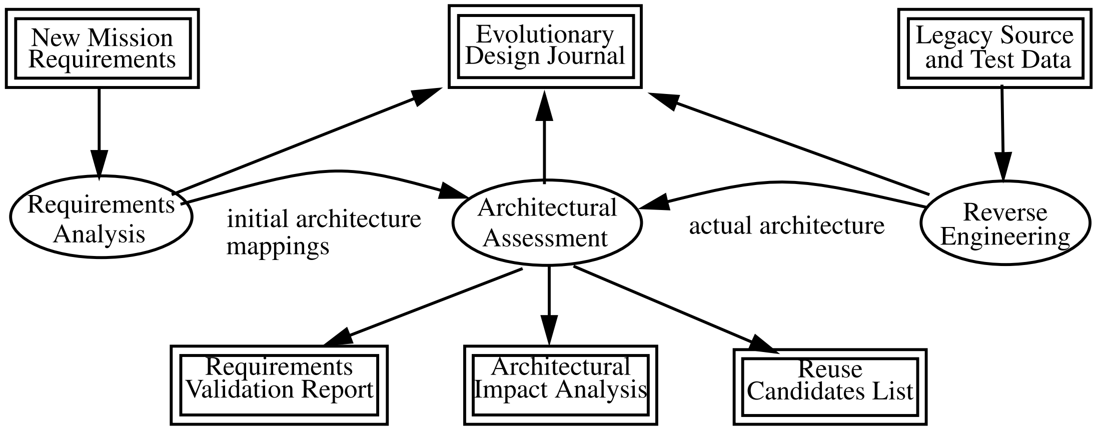


Figure 1 MORALE inputs, activities and outputs

code and test data and a statement of the new requirements that the system must meet in the form of mission-oriented goals. Requirements analysis uses the mission-oriented statement to suggest concepts of importance in the overall structuring of the system; that is, it provides a suggested taxonomy for an initial architectural description of the system. Requirements analysis also provides a collection of scenarios that represent a complete description of the new system requirements. These will serve as the basis for the architectural impact analysis. Architectural assessment predicts the impact a set of scenarios will have on an architecture. Reverse engineering provides techniques for assuring that the architectural representation, the basis for the impact analysis, is an accurate reflection of the actual system under scrutiny. The three pieces work together to predict the extent to which the architecture of the old system can be adapted to meet the new requirements.

Besides this assessment, MORALE produces several other artifacts in planning system evolution: a list of candidate code components from the old version that may be reused in the new version and the ability to enhance impact analysis by considering requirements on the old version of a system that must still be supported in the new system (a sort of regression suite for architectural evaluation). The entire MORALE suite of techniques provides a historical record in the form of an evolutionary design journal, incorporating items such as system goals and problems, traceability information between requirements and code, design alternatives and rationale. The result of the MORALE analysis process is the development of instantiated architectural components together with a record of design decisions and the mission-oriented goals that were met or traded off to develop the new system.

II. The MORALE Software Evolution Method

A. Mission Oriented Requirements Analysis

The first step in evolving an existing system is understanding the proposed change in the context of the system's current goals and behavior. MORALE includes a technique using scenarios [16] and structured inquiry [15] for eliciting such information, called ScenIC. ScenIC takes as input both human responses to the questions it suggests and a set of actors, system and environment components capable of accomplishing various subgoals. It produces as output a set of operational requirements and a network of rationale describing both the reason why certain actions take place and the situations (scenarios) when they might take place.

ScenIC consists of three concurrent activities: expression, discussion, and refinement. Expression constructs three artifacts: (1) a teleological network model of rationale including system goals, actions, actors, and obstacles, (2) a set of scenarios (action/actor pair sequences), and (3) a set of operational requirements specifying those actions that are the responsibility of system components, the goals these actions accomplish, their preconditions, and their effects.

The second ScenIC activity is discussion. ScenIC discussions are driven by a set of standard questions, such as *why is this action performed?*, *what are the subgoals of this goal?*, *what obstacles can prevent the accomplishment of this a goal?*, *how can the obstacle be prevented from happening?*, and *what secondary goals come into play if the obstacle occurs?* The questions are elaborated in terms of scenarios that can serve both to rationalize system behavior and elicit further requirements from customers or designers. There is a rich set of heuristics for generating scenarios from the teleological model.

The third ScenIC activity is refinement, which is the incremental process of making the artifacts more detailed, precise, correct, and complete. In particular, refinement generates subgoals from goals, alternative allocations of actions to actors, elaborated goals that take account of obstacles that had previously been ignored, and episodes, scenario fragments that illustrate the accomplishment of goals or the occurrence of obstacles.

To make these steps more tangible, consider the situation where Mosaic is being extended to add user-configurable viewers. An initial set of actors include the browser, the user, and the server. An initial scenario includes end user actions specifying a URL and requesting access, server actions of accessing the page and formatting a response, and browser actions of receiving the resulting page and displaying it. ScenIC helps the designer understand the reasons behind current behaviors (i.e. the ascription of goals to actors) and to consider alternative allocations and any obstacles that may need to be handled. The resulting information is generated and expressed resulting in operational requirements and a semantic network.

During the ScenIC analysis of Mosaic, the action of interpreting an incoming page arises. An obvious obstacle is the browser not being able to display pages of that format. Questions are posed leading to alternative ways of mitigating the difficulty, such as saving the file to disk, converting to a default format such as text, or asking the user to specify a viewer for displaying the data. It is the latter possibility that pertains to the Mosaic enhancement being considered. Elaboration of this situation leads to scenarios that can be used by the SAAM architectural evaluation technique described in subsection C. Figure 2 below shows an illustrative fragment of a scenario, a portion of the teleological model created in this design episode, and the new operational requirement that results.

B. Architecture Extraction

There are many unexploited resources that an existing version of a software system can provide to the developers of a new version.

- There is the source code itself. The difficulty here is deciding exactly which pieces are directly usable, which can be readily adapted, and which are either no longer needed or not worth the trouble. Less obvious is that the existing system can provide requirements for the new version. The new version is expected to reproduce most of the old version's behavior, adding new features or modifying parts of the old version. Hence, the old version can serve to augment and

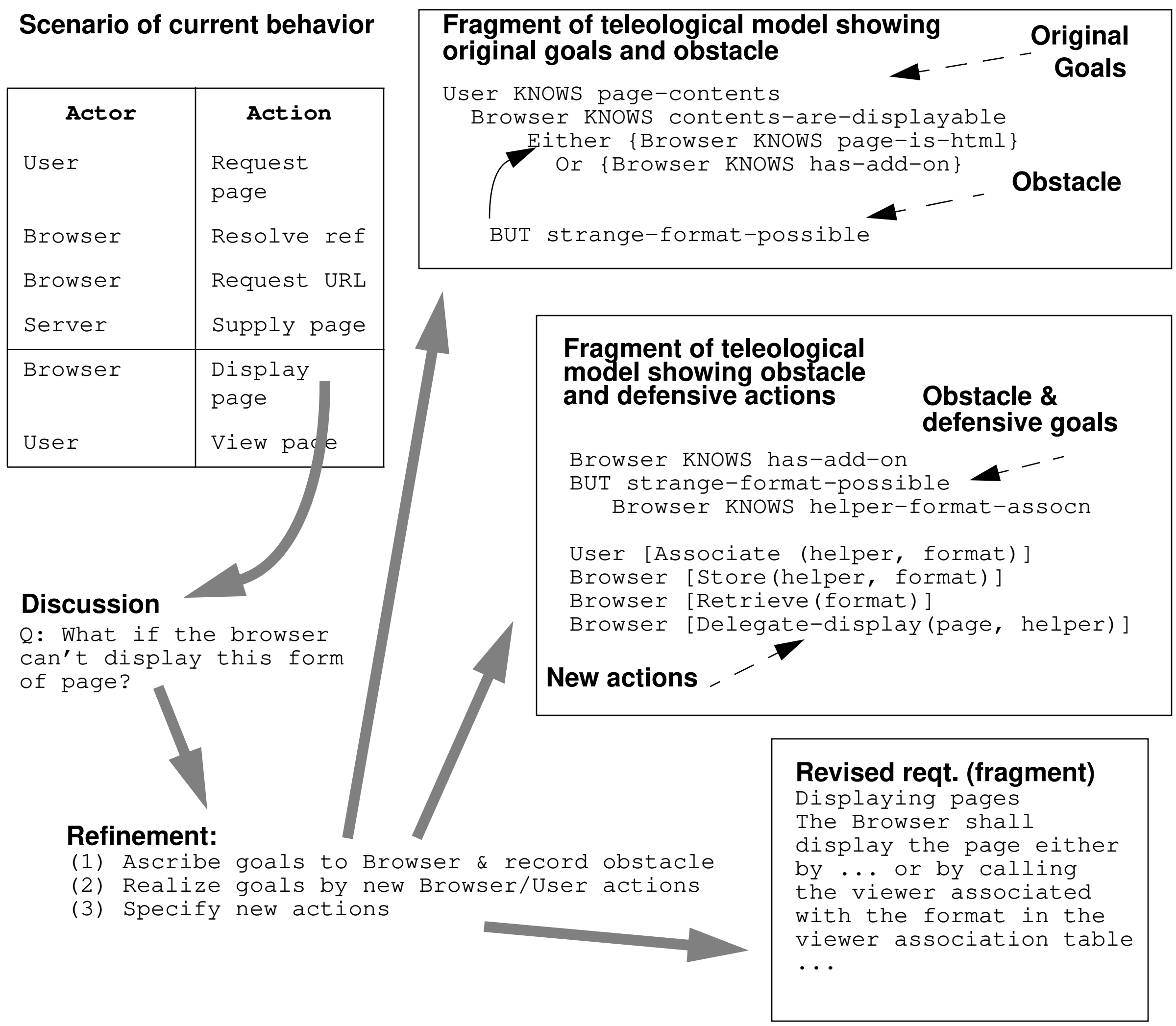


Figure 2: Illustration of ScenIC being applied to the addition of a functional requirement. Exploring a scenario results in a question that is answered by refining the teleological model and specifying a new functional requirement.

validate the requirement provided for the new version.

- Old versions of a system can also provide developers data about what worked and what didn't, particularly concerning architectural design decisions. If a particular aspect of the architecture has over time been proven brittle as evidenced by a history of structural alterations, then there is evidence that that architectural approach is not flexible enough to adapt to the kind of changes that this product family confronts.
- Old versions can also act as laboratories, providing a test environment for developers to learn more about how a system intended to solve a particular class of problems actually works. This includes information about what components are exercised and in what combination when confronted with particular classes of requests from users.
- Finally, in some cases other artifacts such as regression tests, source code control histories, and design documentation can provide insight into the relationship between original intent and actual use.

These resources are typically not very well exploited when constructing a new version of a system, and one of MORALE's primary contributions is to remedy this situation. Doing so requires understanding the existing version in ways that go beyond current reverse engineering technology. In particular, the MORALE architectural extraction process proceeds as follows.

- An analyst is confronted with the task of preparing a plan for evolving an existing version of a system to satisfy new mission requirements. The analyst has a draft of the new requirements, access to the source code of the existing system, and possibly unreliable documentation, test sets, etc.
- The new requirements are expressed in terms of usage scenarios. The scenarios describe how the architectural elements of the new version interact to accomplish its mission.
- Using an appropriate scenario, significant event types are identified and the source code of the existing version instrumented to report those events. Test data is generated and the code executed. Raw event trace data is obtained.
- Using a visualization tool, the analyst peruses the event trace data. During this analysis, low-level events are abstracted up to design-level behavior by identifying and understanding recurring sequences of events. Additionally, tool features are provided to help an analyst locate and focus on particular aspects of the system's behavior and to compare expected and observed behavior.
- The abstract results are fed back to the architectural analyst who refines and augments the architectural description with new architectural elements and with behavioral descriptions of element interactions. The process is iterated to increase the accuracy of the description.

This process results in several useful pieces of information. First, there is a description of the actual architecture of the existing version as it responds in a set of situations known to be relevant to the new version. This description includes both architectural elements and behavioral interactions. We can determine the differences between the architecture of the existing system and the proposed architecture of the new system. The differences can then be used to judge the amount of effort required to evolve the old architecture to its new form. Likewise, the differences can be used to suggest software components suitable for reuse.

The approach described above makes use of existing program understanding technology, including the following:

- Ability to produce an initial approximation of system architectural elements and their interactions using, for example, structure charts or cross-reference information;
- Ability to visualize event data [10];
- Ability to infer abstractions [5];

Our contributions extend this work in the following ways:

- Use of dynamic analysis, rather than purely static structural analysis;
- Detection of different types of architectural abstractions (e.g., protocols).

As an example, imagine that a maintainer has been charged with modifying the behavior of the Mosaic 2.4 web browser to allow for user-configurable viewers. Given that the design documents are unavailable, the visualization tool can be used to analyze the behavior of the existing system for that particular usage scenario.

The first step is to perform a static analysis of the system by compiling it using the Solaris C compiler, which when certain compilation flags are set provides access to the static analysis data the compiler generated during the compilation process. The source code is then instrumented and new source generated. Next, an execution trace is generated by executing the instrumented code in a fashion prescribed by the usage scenario; i.e. by following some URL links, especially those

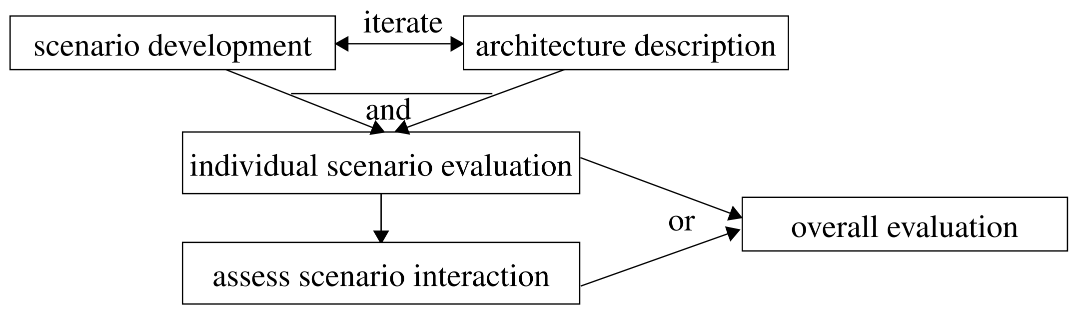


Figure 3 : Activities and dependencies in SAAM's scenario-based analysis

which invoke external viewers. These traces are then analyzed to identify the behavior of Mosaic that involves following a link and deciding how to display the resulting page. The result is an understanding of how the current version of the system reacts when confronted with a situation similar to the one the enhanced version will be asked to handle. The understanding takes the form of a description of the components involved and the interactions they use to communicate.

C. Change Impact Assessment at the Architectural Level

It is desirable to assess the impact that a new set of mission requirements will have on the architecture of the existing system. A particular method for doing a scenario-based architectural analysis is SAAM (Software Architecture Analysis Method). SAAM was originally developed to enable scenario-based comparison of competing architectural solutions [6, 1, 13, 12, 13], but we are using it within MORALE to help designers predict the impact that a set of changes, in the form of scenarios, will have on an existing system as it evolves to meet new requirements. Figure 3 shows the steps of SAAM and the dependency relationships between those stages. The steps of SAAM, and the products of each, are:

1. **Describe candidate architecture.** The candidate architecture or architectures should be described in a syntactic architectural notation that is well-understood by the parties involved in the analysis.
2. **Develop scenarios.** Develop task scenarios that illustrate the kinds of activities the system must support and the kinds of changes which it is anticipated will be made to the system over time. Scenarios are either *direct* (supportable by the current version) or *indirect* (anticipated for the new version). In developing these scenarios, it is important to capture all important uses of a system. Thus scenarios will represent tasks relevant to different roles such as: end user/customer, marketing, system administrator, maintainer, and developer.
3. **Perform scenario evaluations.** For each indirect task scenario, list the changes to the architecture that are necessary for it to support the scenario and estimate the cost of performing the change. A modification to the architecture means that either a new component or connection is introduced or an existing component or connection requires a change in its specification. By the end of this stage, there should be a summary table which lists all scenarios. For each indirect scenario the impact, or set of changes, that scenario has on the architecture should be described. A tabular summary is especially useful when comparing alternative architectural candidates because it provides an easy way to determine which architecture better supports a collection of scenarios.

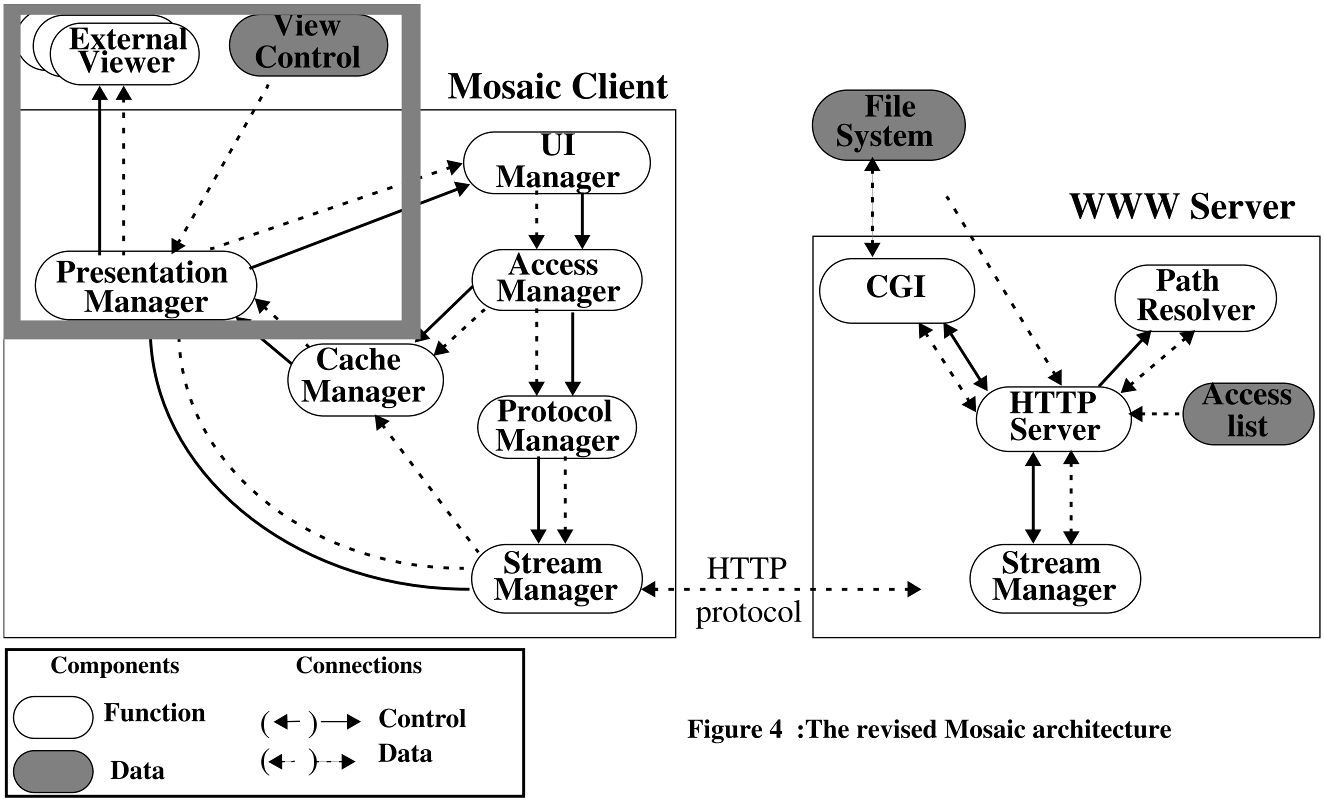


Figure 4 :The revised Mosaic architecture

4. **Reveal scenario interaction.** Different indirect scenarios may necessitate changes to the same components or connections. In such a case we say that the scenarios *interact* in that component on connector. Determining scenario interaction is a process of identifying scenarios that affect a common set of components. Scenario interaction measures the extent to which the architecture supports an appropriate separation of concerns. For each component determine the scenarios which affect it. SAAM favors the architecture with the fewest scenario conflicts.

5. **Overall evaluation.** Finally, weight each scenario and the scenario interactions in terms of their relative importance and use that weighting to determine an overall ranking. This is a subjective process, involving all of the stake-holders in the system. The weighting chosen will reflect the relative importance of the quality factors that the scenarios manifest.

The Mosaic client/server architecture

The Mosaic 2.4 browser is part of the World Wide Web distributed hypermedia system, an organization of clients and servers that share a common set of communication protocols and markup languages. Servers make Internet resources available to a community of clients that speak a common protocol. The software architecture for this client/server system, modified as explained below to support the new scenario requirement of user-configurable external viewers, is shown in Figure 4.

WWW clients provide a graphical User Interface Manager that captures user requests for information retrieval in the form of a Uniform Resource Locator (URL) and passes the information to the Access Manager. The Access Manager determines if the requested URL exists in cache and also interprets history-based navigation, e.g. ‘back’. If the file is cached, it is retrieved from the Cache Manager and passed to the Presentation Manager for display to either the User Interface or an external viewer. If the file is not cached, the Protocol Manager determines the type of request and invokes the appropriate protocol suite to service the request. This protocol is used by the client Stream Manager for communicating the request to the server. User configurability of external viewers is to be provided by allowing the Presentation Manager to consult a static View Control configuration file that maps document types to external viewers. The area of the Mosaic client that is boxed by a gray square indicates the location in the architecture that is the subjec of changes from this new requirement.

D. Adaptive Design

A common problem in evolutionary design of software systems is incremental addition of new functionalities. In this class of problems, an operational software system delivers a set F(old) of functions f(1),...,f(n-1), but subsequently a new set, F(new), containing an additional function, f(n), is desired. The specific problem of adding user-specified external viewers to Mosaic 2.4 is an instance of this class of problems. Often the additional function, f(n) in F(new), is similar and related to some function f(i) in F(old). For example, while Mosaic 2.4 enables external viewing of Postscript files using a utility program such as Ghostview, an end user may wish to view other types of files, such as PDF files using the Acrobat Reader, where the new functionality is similar to the one already delivered. The design issue in this class of problems becomes the adaptation of the current structure of the software system, S(old), that delivered F(old), into a modified structure, S(new), for delivering F(new).

The MESA (Model-Based Evolution of Software Agents) methodology addresses this class of problems in evolutionary design. It uses a Structure-Behavior-Function (SBF) model [9] [19] of the software system. Figure 5 illustrates the organization of the SBF model of the portion of Mosaic responsible for external viewing of Postscript files. The model describes the architecture of the system in terms of its task-method structure and knowledge sources. Tasks (e.g., Display-Interpreted-File) constitute the building blocks of the system's architecture. Methods (e.g., External-Display) decompose a task into subtasks, which, in turn, are recursively decomposed into smaller and simpler subtasks. The leaf tasks (e.g., Access-Display-Program) are directly accomplished by procedures that use domain knowledge and get encoded in the program. A task is characterized as a transformation from an input information-state to an output information-state. It is specified by the type(s) of information it consumes as input and produces as output, and specific relations that hold true among the task's inputs and outputs. In Mosaic, a task such as Display-Interpreted-File actually is a family of tasks, whose members differ from each other only in specifics of the information they take as input and give as output. but not in the relations between the input and output information. A method is characterized as a partially-ordered sequence of information-states and information-state transitions that compose tasks at one level into a task at the next higher level in the task structure. It specifies the data and control dependencies between the tasks at the lower level. Again, in Mosaic, a method such as External-Display really is a family of methods, whose members differ from each other only in specifics of the information in the state transitions, but not in the data/control dependencies among the lower-level tasks. The SBF language provides primitives for specifying the functional semantics of tasks (including the leaf tasks), the compositional semantics of methods, and also the causal (data/control) semantics of the interdependencies among tasks, methods, and domain knowledge.

The SBF model of a software system enables multiple strategies for modifying S(old) to obtain S(new). Here we outline one strategy that we will call *within-problem-analogy:* (1) The function f(i) in F(old) that most closely matches the additional function f(n) in F(new) is identi-

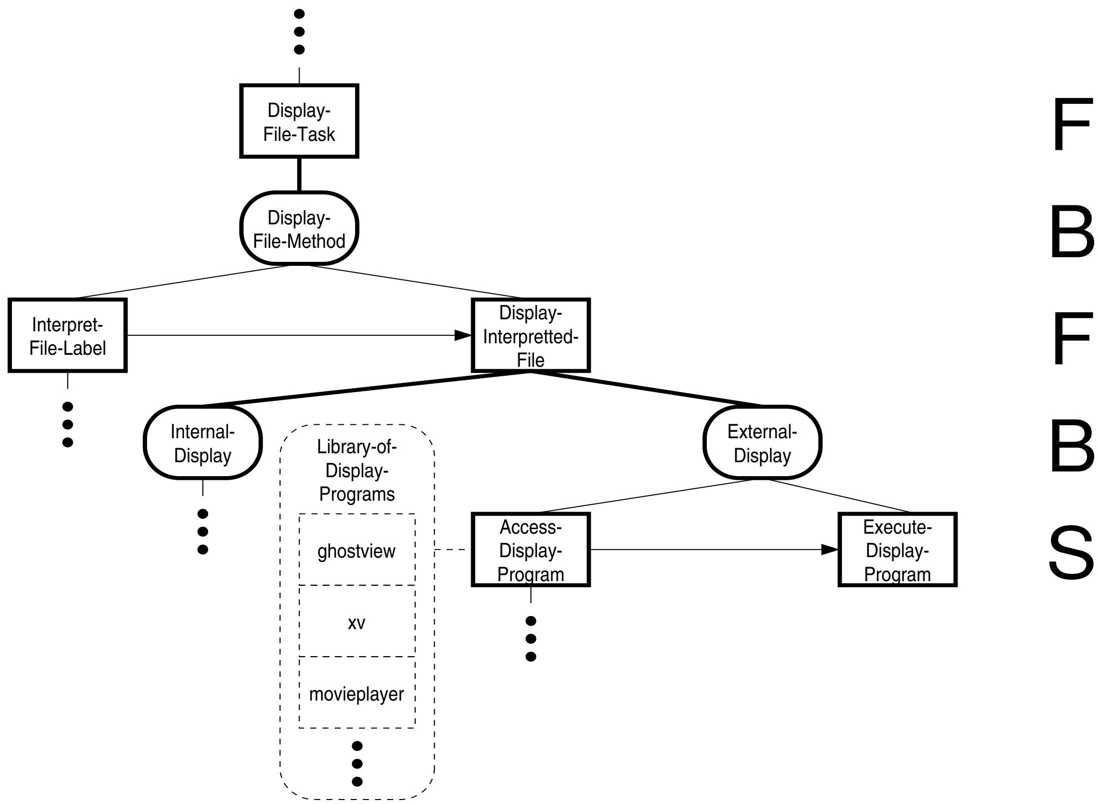


Figure 5: A partial SBF model of a Mosaic 2.4.

Rectangular boxes indicate tasks while rounded boxes indicate methods. Knowledge sources are contained within dotted lines. Thick lines between boxes represent the availability of methods to accomplish a task; thin lines represent a method's composition of subtasks; Horizontal arrows indicate ordering of subtasks within a method.

fied. In the example of viewing PDF files, this may result in the identification of viewing Postscript files as the closest match. The representation of the functions in the SBF language enables this matching. (2) The components and the behaviors responsible for achievement of f(i) are localized and identified. In our running example, the components of Access-Display-Program, Execute-Display-Program, and Library-of-Display-Programs are identified, along with the behavior of External-Display that composes the functions of these components into the function Display-Interpreted-File for Postscript files. The functional, compositional and causal semantics of the SBF model enable this localization and identification. (3) The identified components and behaviors are abstracted into a Generic Teleological Mechanism (GTM) [4] by removing all structural information specific to display of Postscript files. For example, in the GTM, the components Access-Display-Program and Execute-Display-Program are specified only by their functions, and the functional specifications refers only to types of information they take as input and give as output, not to the specifics of information relevant to Postscript files. A GTM thus specifies a functional, compositional and causal design pattern. The SBF language provides the vocabulary for representing GTMs. (4) The abstracted GTM is instantiated in the context of the additional function f(n). In our example, the GTM for external viewers is instantiated in the context of viewing PDF files. This instantiation results in the placement of a new display program (the Acrobat Reader, specified as part of f(n)) in the Library-of-Display-Programs and the introduction of a new method for viewing PDF files in the family of External-Display methods. The SBF model focuses the GTM instantiation. In addition, the original SBF model is now revised to reflect the modified structure for the software system.

E. Evolution of User Interfaces

Since it has been estimated that half or more of the code for an interactive system is devoted to implementing the user interface [20], modifying or enhancing the user interface can be a significant part of the software evolution effort. Rewriting the user interface is a tedious and time-consuming task that is usually accomplished by hand.

The Model Oriented Reengineering Process for Human-Computer Interface (MORPH) [14] provides a framework for deriving abstract models of user interfaces from legacy applications and support for modifying and refining the models in order to maintain the user interface from the model level rather than from the specific implementation level. Figure 6 shows the three steps of the MORPH process:

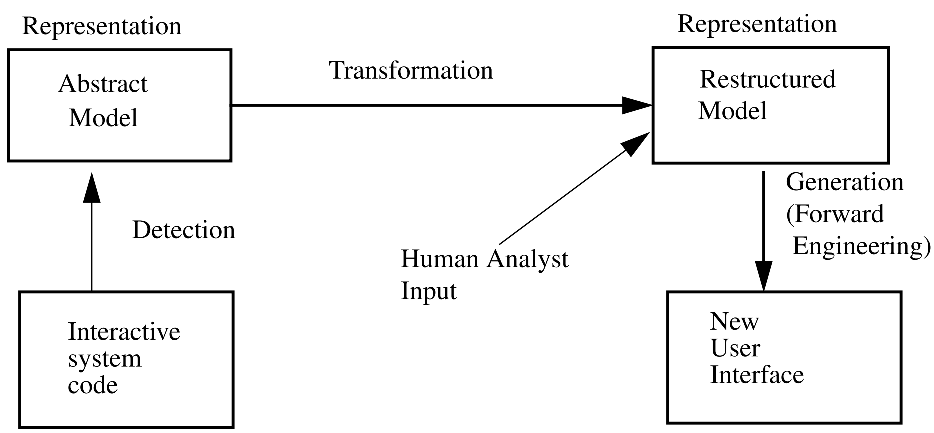


Figure 6 : The MORPH Reengineering Process for User Interfaces

Detecting the user interface model from the legacy code

MORPH performs this program understanding step by applying a set of rules which identify syntactic patterns that implement basic user interface tasks as defined by Foley et al. [8]. The rules are used to build a model of the interface by generating an abstract representation of the task mapped to the code from which it was derived. Attributes of the interface task, based on the declarative models described in [7], can also be detected in order to provide information that can be used later to choose an appropriate specific implementation (for example, a row of push-buttons vs. a cascade menu for a selection task).

Representing the user interface model

MORPH uses a knowledge representation language to store the user interface tasks discovered in the detection step. An abstract concept hierarchy is implemented in the knowledge base, and as user interface tasks are detected, they are defined in terms of the abstract concepts. Once the representation is complete, the designer can evolve the user interface by changing the model.

Transforming the model

The knowledge representation allows MORPH to use inferencing to map the abstract user interface model to the most appropriate components of a specific toolkit implementation. For example, a text number field in a character oriented application might map to a slider in a graphical user interface. The description of the text number field (a "quantify" basic user interface task) in the model allows the MORPH inferencing mechanism to choose the closest match in a particular user interface toolkit (such as Motif or Java AWT).

In our scenario, the Mosaic 2.4 browser user interface needs enhancement to add user-specified viewers. Currently, if an unknown file type is encountered, Mosaic tries to determine if it contains text. If so, it simply defaults to text and tries to display the file. Otherwise, the user is asked to save the file to disk. We need to add a new dialog that allows the user to select between four choices: saving the file, taking the default (displaying it as text), configuring an external viewer, or canceling. The **save** option brings up an existing save dialog. The **default** option simply closes the dialog box and displays the file as text. The **cancel** option closes the dialog box and does not display the file. When the **configure-external-viewer** option is chosen, a new window displaying an editable text version of the *mailcap* file is displayed, in order for the user to add the new type.

MORPH tools are used to derive a model of the current Mosaic interface using static and dynamic analysis. To add the new dialog, a selection task having four alternatives is added into the appropriate place in the model. The text task is added in as an associated action for the selection task alternative, along with the **save** and **default** option alternatives. At that point, inferencing can be used to identify the appropriate Motif widget to implement the selection and task in the code—perhaps a row of push-buttons. From this information, the code can be generated using a GUI builder application, and the user interface enhancement is complete. Since MORPH maintains the mapping between user interface task and the code that implements it, calls to the application functionality can be added as well.

III. Tool Support

The processes described in Section II can be automated in various ways. The MORALE project is developing a collection of tools, called the *Esprit de Corps Suite*. Currently, the tools provide support to the individual processes. Eventually, we intend to integrate them into a comprehensive software evolution support environment.

A. ScenIC View

ScenIC View is a planned collection of tools to manage the expression, discussion and refinement of teleological models, scenarios, and requirements, and the discussion of any of these artifacts. One existing component, GBRAT, is a tabular editor/viewer for goals and their relationships [3]. Support for discussion will resemble the EColabor Inquiry Cycle discussion tool [20].

B. ISVis

ISVis (Interaction Scenario Visualizer) [10] is a tool which supports software engineering tasks requiring dynamic program understanding. The tool uses static analysis data to instrument a subject system, such that upon usage program execution traces are generated. These traces are read by ISVis and views of the actors and interactions in the traces presented to the user. Features allow the filtering and abstraction of information, including global overviews of scenarios containing hundreds of thousands of interactions, facilities to find recurring sequences of interaction in scenarios, grouping of interactions into higher-level scenarios, grouping of low-level actors into higher-level components, and saving and restoring of analysis sessions. Figure 7 contains a screen dump for ISVis where the foreground window contains a segment of an event trace, and the window in the background provides the analyst with a variety of mechanisms for abstracting event data into intercomponent protocols at the architectural level.

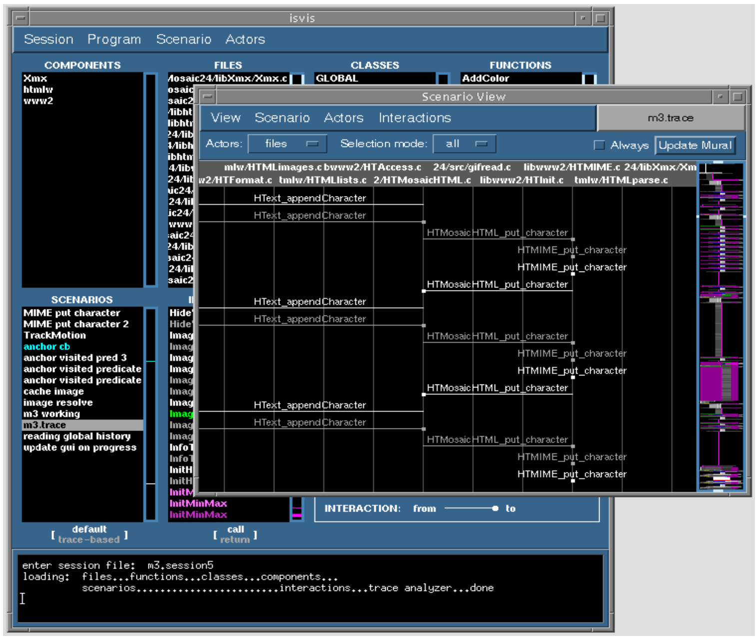


Figure 7 ISVis Interaction Scenario Visualization Tool

C. SIRRINE

SIRRINE (Self-Improving Reflective Redesigner Integrating Noteworthy Experience) is a tool under development for instantiating the MESA methodology. The knowledge-based tool is a shell that provides both the SBF language for representing the architecture of software systems such as Mosaic 2.4, and methods such as within-domain-analogy for addressing problems of evolutionary design. The tool is being developed in Common Lisp.

D. MORPH Tools

MORPH tools are under development to automate the extraction of user interface models from legacy code. MORPH uses a rule base that implements syntactic pattern recognition and builds a knowledge-based representation of the user interface. The interface model can then be transformed into specific windowing toolkit implementations. The MORPH rule base is implemented in the Refine [17] language, and the knowledge representation language for the abstract model is CLASSIC [18].

E. SAAM Tools

The SAAM process for impact analysis can involve a large number of scenarios affecting a large number of architectural components. A tool for compiling all of the SAAM analysis has been developed by researchers at University of Waterloo and the Software Engineering Institute.

We are currently modifying this SAAM tool to integrate it within the larger MORALE tool suite and method.

IV. Issues Raised and Current Status

MORALE is currently a collection of related techniques and tools that need to be more tightly integrated. The resolution of several issues will further this objective. The first is the concept of scenarios. ScenIC, SAAM, and ISVis each currently have their own idea of what a scenario is. These ideas differ in granularity and formality. For example, a SAAM scenario is informal and expressed in terms of architectural constructs. For ISvis, scenarios correspond to sequences of actual system-level events such as subprogram calls and returns. For ScenIC, a scenario is expressed in terms of goals, actors, actions, and obstacles. We see nothing incompatible in these views, but a more precise description must be formulated.

Another issue that our work with MORALE has raised is how to describe architecture. There is currently much work in the research community in the area of Architectural Description Languages (ADLs). For example, SBF is a derivative of the FR architectural description language [2]. We need to ascertain MORALE's needs in the ADL area and determine the extent to which SBF satisfies them.

A third issue relates to design rationale. There is much debate on what this terms means, but for MORALE, the concept certainly includes ScenIC's goals, actors, actions, and obstacles, SAAM's component interactions and cost weightings, the actual system behavior as abstracted by ISVis, MESA's task-method structure and knowledge sources, and the abstract user interaction model derived by MORPH. We need a better understanding of how these relate and what, if any, aspects of rationale we have overlooked.

As far as tools are concerned, *Esprit de Corps* is currently completely unintegrated. We need to develop a common data model to support interoperation and a unified user interface. Moreover, the MORALE project is funded by the U.S. Department of Defense as part of its program in the area of the Evolutionary Design of Complex Software. In addition to the integration activities described above, we also need to interact with the fourty other projects in this program.

References

[1] G. Abowd, R. Kazman, and J. Pitkow. "Analyzing Differences Between Internet Information System Software Architectures." *Proceedings of ICC '96*, Dallas, Texas, June 1996.

[2] D. Allemang. "Using Functional Models in Automatic Debugging." *Expert*, December, 1991, 13-18.

[3] Ana AntÓn, Eugene Liang and Roy Rodenstein. "A Web-Based Requirements Analysis Tool." *Proceedings of the Fifth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'96),* Stanford, California, June 19-21, 1996, 238-243, IEEE Computer Society Press.

[4] Sambasiva Bhatta and Ashok Goel. "Learning Generic Mechanisms from Experiences for Analogical Reasoning." *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, Boulder, Colorado, July 1993, 237-242, Lawrence Erlbaum, Hillsdale, New Jersey.

[5] Jonathan E. Cook and Alexander L. Wolf. "Toward Metrics for Process Validation." *Proceedings Third International Conference on the Software Process,* Reston, Virginia, October 10-11, 1994, 33-44.

[6] P. Clements, L. Bass, R. Kazman, G. Abowd. "Predicting Software Quality by Architecture-Level Evaluation." *Proceedings of the International Conference on Software Quality*, Austin, Texas, October 1995.

[7] Dennis deBaar, James D. Foley, and Kevin E. Mullet. "Coupling Application Design and User Interface Design." *Proceedings of CHI `92*, May 3-7, 1992.

[8] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice, Second Edition*, Addison-Wesley, 1990.

[9] Ashok Goel. "Model Revision: A Theory of Incremental Model Learning." *Proceedings of the Eighth International Conference on Machine Learning*, Chicago, Illinois, June 1991, 605-609, Morgan Kaufmann.

[10] D. Jerding and J. Stasko. "Using Visualization to Foster Object-Oriented Program Understanding." Technical Report GIT-GVU-94-41, Graphics, Visualization and Usability Center, Georgia Institute of Technology, July 1994.

[11] R. Kazman, G. Abowd, L. Bass, and P. Clements. "Scenario-based Analysis of Software Architecture." *IEEE Software*. **13**(6):47–56, November 1996.

[12] R. Kazman, L. Bass, G. Abowd, and S.M. Webb. "SAAM: A Method for Analyzing the Properties Software Architectures." *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, 81-90.

[13] D. S. McCrickard and G. D. Abowd. "An Architectural Analysis of Graphical Debuggers." *Proceedings of the International Conference on Software Maintenance — ICSM'96*. Monterey, CA, November 1996.

[14] Melody Moore. "Rule-Based Detection for Reverse Engineering User Interfaces." *Proceedings of the Third Working Conference on Reverse Engineering*, IEEE Computer Society Press, Monterey, California, November 1996.

[15] Colin Potts, Kenji Takahashi, and Annie I. AntÓn. "Inquiry-Based Requirements Analysis." *IEEE Software*, 11(2): 21-32, March, 1994.

[16] Colin Potts. "Using Schematic Scenarios to Understand User Requirements." *Proceedings DIS'95: Symposium on Designing Interactive Systems*, Ann Arbor, Michigan, August 23-25 1995, ACM.

[17] Reasoning Systems. "The Refine User's Guide." Reasoning Systems Inc., 3260 Hillview Avenue, Palo Alto, CA 94304.

[18] Laurie Alperin Resnick et al. "CLASSIC Description and Reference Manual for the Common LISP Implementation Version 2.1." AT&T Bell Labs, Murray Hill, New Jersey, May 15, 1993.

[19] Eleni Stroulia and Ashok Goel. "A Model-Based Approach to Reflective Learning." *Proceedings of the 1994 European Conference on Machine Learning*, Catania, Italy, April 1994, 287-306.

[20] J. Sutton, and Sprague, R. "A Survey of Business Applications." *Proceedings of the American Institute for Decision Sciences 10th Annual Conference, Part II*, Atlanta, Georgia, 1978.

[21] Kenji Takahashi, C. Potts, V. Kumar, K. Ota and J. Smith. "Hypermedia Support for Collaboration in Requirements Analysis." *Proceedings of the Second International Conference on Requirements Analysis (ICRE'96)*, Colorado Springs, Colorado, April 15-18, 1996. 31-40. IEEE Computer Society Press.

ScenIC

Guidebook for the Scenario-Based Inquiry Cycle Method

Colin Potts
Georgia Tech, College of Computing
March 9, 1997

-

-

I .Introduction

A .Purpose

This document describes the ScenIC methodology as used to support the evolution of legacy software in the MORALE project (Mission-Oriented Requirements Analysis and Legacy Evolution). The emphasis of this document is the forward engineering of requirements for new systems or new versions of existing systems. Other guidebooks in the MORALE series deal with architectural evaluation and reverse engineering.

ScenIC builds on a generic model of requirements and design activities known as the Inquiry Cycle. The Inquiry Cycle is a process model and an associated prescriptive framework for acquiring, analyzing and refining requirements descriptions and design specifications for systems. The Inquiry Cycle is method-neutral. Specializing the Inquiry Cycle process for a specific method or collection of techniques involves selecting a set of standard representations for describing the system and a set of discussion guidelines to direct the refinement process.

ScenIC, as the acronym suggests, is a specialization of the Inquiry Cycle (IC) that emphasizes the invention of scenarios and the refinement of requirements and early designs based on an analysis of these scenarios. ScenIC provides the following features:

b Descriptions of system purpose in terms of goals and their dependencies.

b Scripts: descriptions of scenarios (concrete examples of envisaged system behavior).

b Guidelines for identifying and refining goals and for realizing goals in terms of behaviors of system components.

b Representations of the rationale underlying these refinement decisions.

ScenIC is an abstract refinement method and does not require the adoption of a particular project organization or team practices. For example, ScenIC can be used in conjunction with contractual development projects in which the requirements are elicited by staff of a contracting organization interviewing customer representatives; equally, however, it can be used to focus discussions in team-based decision-making procedures, such as Joint Application Design sessions or Participatory Design "future workshops." The insertion of ScenIC guidelines into project organizations and team practices is not considered in this guidebook.

The remainder of this section introduces ScenIC and its context in the evolution of legacy systems. We start by explaining the philosophy of requirements analysis and design as the incremental development of descriptions and what ScenIC descriptions are about and how they are structured. We then provide a quick summary of the ScenIC method and an outline example. In Section II, we describe the method in more detail, concentrating on the knowledge or information that is required when applying the method, the information that it produces, a step-by-step guide that emphasizes the refinements of this information, and a set of evaluation criteria for reviewing ScenIC descriptions. Section III provides an example in more detail, the addition of some functionality to a web browser. At the end of the guidebook are a glossary of terms and an annotated bibliography of research and practice.

A .ScenIC as a Strategy for Incremental Description

1 .Software as Description

In the Inquiry Cycle and its specializations like ScenIC, we take seriously Michael Jacksonõs thesis that software engineering amounts to the production of descriptions.

> Description is at the heart of software development. It is not an exaggeration to say that the whole business of software development is making descriptions. Programs are descriptions of machines; requirements are descriptions of the application domain and the problems to be solved there; and specifications are descriptions of the interface between the machine and the application domain. Every product of development is a description of some part or aspect or property of the problem or of the machine or of the application domain.
>
> [Jackson 1995 p. 3]

ScenIC, then, is a method for producing descriptions of a systemõs requirements. As such, it involves recommendations for expressing information in these descriptions, analyzing and criticizing descriptions, and refining them so that they better capture requirements.

a)Subject of ScenIC Descriptions

The subject of a description is whatever the description describes. Requirements are usually thought of as being descriptions of what a system should do (and not how it should accomplish its requirements). We agree with Jackson, though, that a requirements description usually includes a lot of factual information about the application domain as well as information about the problem to be solved. In ScenIC, therefore, requirements descriptions may have application knowledge, the current situation or problem, envisaged or hypothetical solutions, and desired goals or system behaviors as their subjects. To avoid confusion, these subjects tend to be described in different kinds of description, but for now it is only necessary to be aware of the generality of subject matter in ScenIC. Moreover, because ScenIC is used to help guide the evolution of legacy systems, the subject matter of a ScenIC description may legitimately be the current architecture of the system (that is òhowó as well as òwható information).

It is not necessary in ScenIC to distinguish between software systems and systems in general. In aerospace, Defense and industrial applications, this distinction is sometimes held to be important because it reflects organizational boundaries and responsibilities. There is no need to abandon such a deeply rooted distinction when using ScenIC, and in this guidebook, we will be concerned exclusively with software requirements, but the concepts of goal, agency, responsibility and behavior are general enough that they could be used to describe other types of system, and as you will see even software-only requirements can be understood only in the broader context of the physical and organizational environment of the system.

Distinguishing between a system and its environment reflects a division of responsibility between the designer and others. For example, in designing an interactive system, the designer is responsible for the system but not for the behavior of the users. The designer cannot implement users, even if the design of the system influences their behaviors, but the designer does implement the system and is responsible for its qualities.

Do not confuse this distinction between system and environment with another distinction: that between an actual situation and a desired situation. The actual situation in the case of an evolving system includes both system and environmental parts, as does the desired situation reflected in the requirements.

a)Moods of ScenIC Descriptions

Jackson distinguishes between two subjects of descriptions in terms of the grammatical mood of the verbs that are used. A description of an actual situation is a collection of statements of fact in the indicative moode. The real test of such a description is whether the claims it embodies are actually true. If a description states that overtime is always billed at time-and-a-half, or that only department managers may authorize travel plans, or that the architecture of the system currently has the database updates being performed on a central server, then these are all statements of fact that are either true or false. Whether they are desirable or undesirable is not the issue; the issue is whether they are true. We refer to this type of description as a *factual description.*

Whereas descriptions of the current situation employ the indicative mood and are judged by their accuracy, descriptions of a desired situation employ the optative mood and are judged by whether the situation described really meets the readerõs values. Thus, a requirement to reduce travel authorization time by fifty percent or to decentralize and replicate mission data are neither true nor false: but they may be more or less desirable. We refer to this type of description as an *ideal description*.

A third type of description, unrecognized by Jackson, refers to hypothetical situations and tends to employ the subjunctive mood. Thus an imagined situation or behavioral scenario describes a possible state of affairs to which the customer and designer are not yet committed. Such a situation does not exist, so the description cannot be said to be true of false; and it is not yet committed to, so it is not desirable or not. To the extent that hypothetical descriptions are like factual descriptions, we judge them by the plausibility of their assumptions and the feasibility of their occurrence, and to the extent that they are like ideal descriptions, we judge them by the desirability of their consequences and the insights that they provide about the requirements. We refer to this third type of description as a *hypothetical description*.

The following table summarizes the differences between the three types of description. We need all three in ScenIC. Ignoring the ideal descriptions is obviously not appropriate in a method aimed at clarifying requirements; ignoring factual descriptions is a poor idea also, because the desirability of requirements can often only be judged against assumptions of fact; and finally, to ignore hypothetical descriptions would be to assume that statements of need can be derived and analyzed in a purely rational manner without imagining or illustrating what the **Formality of ScenIC Descriptions**

	Factual descriptions	*Ideal descriptions*	*Hypothetical descriptions*
Mnemonic	òAs isó description	òTo beó description	òMaybeó description
Examples	Domain models Architecture of system	Statements of need Goals or objectives Software/system requirements	Cases Scenarios
Grammatical mood	Indicative	Optative	Subjunctive
ScenIC representations	Domain model Actors in the system (components) and its environment (agents).	Teleological model Behavioral specifications	Scenario scripts

Descriptions may be expressed with varying degrees of formality. At one extreme is informal oral or written text, while at the other are languages with formal semantics that admit the possibility of formal inference. In between are schemes that provide for some structuring of descriptions so that quality checks may be carried out without the description itself being processed. ScenIC includes informal and semi-formal representations for descriptions of the factual, ideal and hypothetical. Formal representations akin to KAOS [REF] and a ScenIC interface to SAAM [REF] and SBF [REF] are under investigation.

a)Language of Descriptions

The ScenIC philosophy is that the process of developing descriptions (that is, expressing, criticizing and refining them) is more important than the nature of the representations used to express the descriptions. ScenIC provides a number of sub-languages for expressing factual, ideal and hypothetical descriptions, but it is not necessary to use these sub-languages to use ScenIC. Natural language, other textual languages, and graphical languages may also be used. For example, goal hierarchies and networks are quite common in artificial intelligence research, management science and human factors task analyses. The ScenIC representation for teleological models can easily be replaced by any other representation of goal hierarchies without losing the methodological support of the method. Or the representation for semi-structured critical rationale may be replaced by IBIS [REF] or Design Space Analysis [REF] if the user prefers.

1 .The Generic Inquiry Cycle Framework

In the Inquiry Cycle (IC) framework, the process of developing a description or set of related descriptions involves three intertwined and continually executed activities: expression, criticism and refinement. IC is a generic framework for describing and supporting design and planning methods, not just the production of system requirements. During the application of any method, there will be large volumes of informal and formal information that describe the status of the evolving design, plan or requirements specification. We call this information the "corpus" of descriptions. One way of thinking of the corpus is as an integrated model of the current, proposed and envisaged systems and their environments, together with process information such as who proposed what and when, why a decision was made, etc. Such an integrated model presupposes an automated repository and update, access and analysis tools, and it presupposes methodological policies for maintaining the consistency of the model. But another way of thinking of the corpus is as a collection of documents (whether on paper or maintained electronically does not matter much) that collectively represent all the information. The integrated model view of the corpus obviously has great promise in supporting a complex development process, but the required tools are not all available, and the costs in routinizing the representation of descriptions and the processes for expressing, criticizing and refining them would probably be too onerous. In the remainder of the guidebook, therefore, we will assume that the corpus consists of many independent (though conceptually related) documents of varying levels of formality.

In this section, we will discuss IC in the context of requirements acquisition and elaboration. To illustrate the repeated application of the canonical IC step of expressing some information, criticizing it and refining the description accordingly, consider how a single mission-oriented goal might be transformed into a functional requirement. First, the mission-oriented goal is written down (i.e. expressed in the corpus). Then, a discussion may ensue that explores several issues: What does the goal really mean? How can the goal be fulfilled by system functions and user actions? How does the goal interact with others? What could happen in the environment that would block the fulfillment of the goal, and how should the system be specified so that it stops this from happening or recovers when it does? Each of these discussions may refer to the contents of the corpus or add auxiliary information to it (such as scenarios or documentation of trade-off analyses). Finally, the resolution of each of these issues may lead to a refinement of the corpus. This refinement may take the form of a change to the element of the corpus that initiated the discussion (i.e. a refinement of the goal) or a change to another element (i.e. a refinement of another goal) or the addition of a new entity in the corpus (e.g. the introduction of a functional requirement that traces from the goal).

ScenIC is a scenario-based method for acquiring, analyzing and elaborating requirements. It is a specialization of the IC, because it consists of several representations for expressing information in the corpus, several standard discussion templates that are triggered by patterns in the corpus, and standard refinements that modify the content of the corpus. The objective of ScenIC is to transform a corpus consisting of information mainly about the architecture of an existing system and mission-oriented goals for a new version of the system into a corpus consisting of requirements for the new system and rationale for those requirements. The distinguishing features of ScenIC as a requirements analysis method are the heavy and systematic emphasis on mission-oriented goals and their refinement, the pervasive use of scenarios at varying levels of detail to help stakeholders understand the implications of possible requirements, and rich analysis heuristics for identifying patterns in the corpus that deserve stakeholder discussion and scenario-based exploration.

A .Overview of the ScenIC Method

The ScenIC framework is depicted in Figure <ScenIC>. Inputs to the process come from the customer representative and the "Discover Architecture" component of the MORALE architecture. Outputs go to the "Evaluate Architecture" and "Implement Architecture" components of the MORALE architecture.

>>> Figure <ScenIC> about here.

1 .The overall MORALE architecture

-- This part is common across guidebooks. Include Figure <MORALE>.

1 .The ScenIC architecture

a)Inputs to ScenIC

The inputs to ScenIC consist of mission-oriented goals (from the customer), existing requirements (from the customer), and architectural constraints (from "Discover Architecture").Mission-oriented goals and existing requirements may be introduced in several forms. The most easily understood and managed is in the form of informal documentation. Goals and requirements are unlikely to be clearly differentiated in customer-generated documentation. Some guidelines for separating them are given below, as are guidelines for inferring goals that may lie behind documented requirements.

Other goals and requirements may come interactively and incrementally through interviews or meetings. As explained above, we do not present guidelines on conducting interviews or team-based requirements gathering meetings. However, we do present some guidelines on gathering goals and requirements during interviews and meetings.

Mission-oriented goals and existing requirements are purely informal statements either embedded in documents or summarized from interviews and meetings. Goals and requirements produced by the ScenIC method have a semi-formal expression that is described in Section <III.B.???>.

Architectural constraints are relevant for all evolving systems. Some decisions that are made during the requirements determination process involve the allocation of operational requirements to system components. A system may only implement requirements that are feasible in a given architecture. If the architecture is constrained to an existing structure, as it usually is in an evolving system, some requirements may be feasible whereas others are not. Architectural constraints are therefore an important influence on the ScenIC process.

The format of architectural constraints is expanded on in Section <III.B.???>.

a)Outputs from ScenIC

The outputs of ScenIC consist of requirements specifications and requirements rationale.

Requirements specifications describe the required system in enough detail for implementation. They consist of functional requirements for the required system, quality requirements for the required system, and environmental assumptions.

Functional and requirements for the required system are statements that must be true of the required system once implemented. Functional requirements specify what the required system must do in specific situations. Quality requirements specify the degree to which the required system must meet specified quality criteria. Environmental assumptions are assertions about the environment that may be assumed to be true.

We shall refer to all three types of statement, and not just the functional and quality requirements for the required system, as "requirements".

Functional requirements, quality requirements and environmental assumptions may all be attributed to components of the required system or environment. In the case of an evolving system, the feasibility and detailed refinement of many functional and quality requirements depends on the architecture of the current version of the system. Even in the case of a new system, there may be obvious architectural constraints on the structure of the required system (perhaps suggested by the suitability of commercial off-the-shelf or other non-delivered components). In these cases, the functional and quality requirements may be further attributed to architectural components, and not just attributed to the required system as a whole. (We will leave to others the debate as to whether this constitutes premature design.)

Requirements rationale is a collection of representations that explain why the requirements take the form they do. Rationale includes pre-requirements traces, scenarios, and semi-structured discussion records.

Pre-requirements traces are traces back from the requirements to originating statements of various kinds. The most systematically preserved form of pre-requirements traceability in ScenIC is traceability back to mission-oriented goals. Other requirements may be traced back to statements made by stakeholders in meetings or documents.

Scenarios are descriptions of interesting usage patterns for a system. Scenarios may be constructed for the current version of the system, the required system once specified, or for an intermediate (partly specified and putative) system. Scenarios are used to help elaborate the requirements in such a way that they fulfill the mission goals and are feasible in the context of an architecture.

Scenarios may be elaborated for typical, anomalous or otherwise significant usage patterns.

Semi-structured discussion records are records of discussions about the requirements that preserve the major points of the discussion, separating those points from the text of the discussion summary itself. Such points include the stakeholders contributing to the discussion, the major issues or alternatives discussed, the reasons put forth in support of an alternative, and other descriptions referred to during the discussion (e.g. scenarios, mission-oriented goals).

1 .Overview of ScenIC

Because ScenIC can be thought of as a method for transforming one kind of corpus into another, we shall describe ScenIC in terms of its start and end corpus states and the refinements that collectively effect the transformation between them.

The ScenIC corpus consists of two kinds of information: information that cannot be formally manipulated ("unstructured" information), and information that can be formally manipulated ("structured" information). Unstructured information normally takes the form of natural language text (oral as well as written) and sketches. Structured information is expressed in one of the ScenIC sub-languages or expression formats. The start state of ScenIC consists of unstructured input from stakeholders about their requirements for a new version of a system and structured representations of architectural information about the current version.

a)Ideal Descriptions in ScenIC: The Teleological Model and Requirements

ScenIC does not consist of a series of steps that the analyst must perform in order. Rather, it consists of a collection of expression/discussion/refinement cycles that collectively transform the initial information into an acceptable requirements specification. We shall describe these cycles and their constituent representations in more detail in Section II. Here, briefly, is an overview of the major refinements.

Goal identification: A mission-oriented goal is identified in the unstructured information or is rationally reconstructed from the actions performed by components of the existing architecture.

Goal expansion: A goal is expanded into conjunctive or disjunctive sub-goals, and dependencies among goals are identified and recorded.

Obstacle identification: The obstacles that can thwart or block a goal or action are identified and are evaluated to see whether the risk of their occurring outweighs the cost and inconvenience of dealing with them.

Obstacle defense: In the case of obstacles that have been determined to be worth defending against, subsidiary defensive goals or actions are identified to prevent or reduce the likelihood of the obstacle occurring.

Obstacle mitigation: In the case of obstacles that have been determined to be worth mitigating, subsidiary mitigation goals or actions are identified that recover from or reduce the harm done by an obstacle that occurs.

Responsibility allocation: A goal or action is allocated to the required system, an architectural component of the required system, or an actor in the environment.

Goal realization: One or more actions are identified that fulfill a goal.

a)Hypothetical Descriptions in ScenIC: Scenarios

ScenIC gets its name from the extensive role played by scenario analysis in the discussions that lead to most of these types of refinement. Below are the uses of scenarios in each of the types of pre-refinement discussion.

Goal identification: Analysts and stakeholders can easily miss goals that pervasive goals (those that typically map to quality). When exploring scenarios it is common to find something wrong with the envisaged system that violates a hitherto unrecognized goal.

Goal expansion: Similarly, goals may have interdependencies that are only recognized when they are explored in the context of concrete scenarios.

Obstacle identification: The consequences of an obstacle occurring may be explored in a scenario to see whether it is worth developing a defensive or mitigation strategy to deal with it.

Obstacle defense and mitigation: As well as normal cases, scenarios are constructed to evaluate the effectiveness of obstacle defense and mitigation strategies, particularly in cases where significant obstacles co-occur.

Responsibility allocation: Alternative allocations of responsibility (specifically, alternative allocations of actions to architectural components or actors in the environment) may be evaluated by comparing scenarios that explore the different allocations.

Goal realization: Scenarios can only be used to explore goals that are realized by actions. Early analysis of scenarios therefore requires a tentative commitment to an allocation strategy that can be explicitly questioned by analyzing the scenarios that result.

a)Factual Descriptions in ScenIC

- No commitment to any kind of representation.

- Mapping of agency onto architectural components

1 .Rationale Capture

In ScenIC, rationale takes three forms:

1 .Pre-requirements traces from goals and originating statements to requirements. These goals and originating statements explain why requirements are present by explaining their origin or higher level expression in mission-oriented terms.

2 .Scenarios that illustrate desirable behaviors implied by the requirements or undesirable behaviors prohibited by the requirements. Scenarios explain why requirements are present by illustrating desirable or undesirable behaviors.

3 .Semi-structured records of IC discussions. These records are structured in that they have a well-defined

organization, but are unstructured in that the contents of the records are informal text as written or recorded during or after the discussion. Rationale records explain why requirements are or are not present by justifying the reasoning that contributed to the decision.

1 .Evaluation of ScenIC Specifications

A ScenIC-produced requirements specification is judged against the following criteria:

- Do the goals in the corpus from which the requirements are derived accurately record the stakeholders' mission-oriented intent for the system? Goal accuracy is important, because a system developed to fulfill the wrong goals is very unlikely to be effective. Goal accuracy is evaluated by careful feedback-eliciting questions to stakeholders and exploring scenarios with them.
- Do the obstacles in the corpus from which auxiliary goals and required actions are derived correspond with the likely environmental contingencies that the system will face? Obstacle accuracy is important, because obstacle defense and mitigation strategies may lead to many auxiliary requirements in the corpus. If these requirements are not necessary, the system will be avoidably expensive to implement. Obstacle accuracy is evaluated by eliciting feedback from domain experts and performing root-cause analyses on likely problems.
- Are the requirements complete? Completeness refers to the extent to which the stakeholders' intentions are reflected in the requirements. This is an informal criterion, because the requirements are structured whereas the stakeholders' intentions are not only unstructured but unexpressed. However, the requirements can be checked exhaustively against the originating goals and possible obstacles. With goal accuracy and obstacle accuracy assumed, therefore, requirements completeness can be assessed systematically by assessing whether a stated goal is fulfilled by the requirements or explicitly allocated to the environment, and by assessing whether an obstacle is adequately defended against or mitigated by auxiliary goals and system behaviors.
- Do the requirements exclude all unnecessary requirements ("gold plating")? Gold plating is known to be one of the primary risk factors affecting software development, so it is important to find it and weed it out as early as possible. Because an unnecessary requirement is one that is not needed for the system to meet stakeholder intentions, it cannot be assessed directly. In this respect, gold plating is the converse of completeness, and, as in the case of completeness, the requirements can be checked exhaustively against the originating goals and possible obstacles. With goal accuracy and obstacle accuracy assumed, some gold plating can be identified with precision. A necessary but insufficient condition for a requirement being gold plating is for there to be no trace back to any goal or obstacle. More subtle forms of gold plating are dealt with below.
- Do the requirements reflect an optimal realization of the goals in the corpus? There are many possible systems that could meet a set of mission-oriented goals. These systems exhibit different behaviors, different degrees of automation, and different allocations of responsibilities to environmental actors. Part of the evaluation of alternative realizations of a set of goals is whether each realization is architecturally feasible and which are superior with respect to some measurable criteria. These architecturally-aware evaluations are discussed in the SAAM guidebook.

  Another part of the evaluation of realizations, however, involves the degree to which each fits into the environment of the system. Scenarios and feedback-eliciting questions about skills or properties of users, interfacing systems, organizations and devices are used to ascertain the quality of a goal realization.

  A third form of evaluation of the realization of a goal is to ask whether it is as simple as it needs to be. Goals that are fulfilled with unnecessary complexity are a second form of gold plating. This form of gold plating is less easily detected than requirements that do not trace back to goals or obstacles at all and can be identified only with the help of domain and design expertise.
- Are the scenarios sufficiently diverse to induce confidence in their use to analyze the requirements? ScenIC is predicated on the belief that scenarios are valuable tools in the understanding of complex systems. They can be misleading, however, when they fail to explore the space of design alternatives and environmental contingencies with sufficient coverage. Insufficient coverage means that many requirements defects are not detected because the situations are never explored that would make the defects apparent. To ensure greater coverage, ScenIC includes some (admittedly weak) coverage heuristics for generating scenarios. For example, the fulfillment of every goal should be illustrated in at least one scenario; and the occurrence of every combination of obstacles and their defensive or mitigation strategies should also oc-

cur in at least one scenario. Significant combinations of obstacles may also be checked, but what makes a combination significant depends on the domain and can only be determined by an expert.

b Are the scenarios sufficiently detailed to induce confidence in their use to analyze the requirements? As with the coverage issue, scenario detail is vital for developing a good understanding of requirements-related issues. Several techniques can help to make scenarios detailed enough. One is to name all actors and data variables mentioned in the scenario. (Thus, in a library system, one would not explore the "Patron returns book late" scenario, but the "Colin returns 'War and Peace' two weeks late" scenario.) Another is to manage scenario scripts at varying levels of detail that are appropriate for different types of requirements investigations. (Thus, one version of the scenario above would describe what happens in terms of the library's business processes and its fine policies; a second would describe what would happen in terms of time-ordered operations performed by the participating actors; and a still more detailed version would involve events at the level of user interface dialogs and device interactions.)

b Will the system as specified fit into its broader context? There is always a possible residual source of requirements defects that is not removed even when the questions listed above have been answered. Sometimes a system does not "fit", it cannot be used effectively, and it just does not seem useful. In principle, we could predict and avoid this happening by accurate modeling of the stakeholders' goals and careful refinement of goals into requirements. In practice, however, this is not possible, because all the activities in requirements engineering must be undertaken in a state of imperfect and constantly changing knowledge. In particular, compatibility of the proposed system with concrete features of the current context of use may play a disproportionate role in determining whether the proposed system works effectively. These features can be identified by careful application of naturalistic data gathering and interpretive techniques (e.g. those from ethnographic practice). Applying naturalistic techniques in the context of goal refinement is the topic of a separate investigation (Potts and Hsi, 1996) and is not considered further here.

A .Example usage

We will now consider briefly a development scenario in which ScenIC is used in the requirements engineering of a new version of an existing system. This will help you make more sense out of the details in Section II. A more fully worked illustration is contained in Section III.

--- *t.b.w.*

I .Detailed Methodology Description

Section I presented an overview of ScenIC. In this section, we describe ScenIC in more procedural detail. Sub-section A deals with the format of inputs to the method, and Sub-section B with the format of outputs. The Sub-sections in C describe each of the ScenIC requirements refinements. Finally, in Sub-section D we revisit the discussion of evaluation, this time giving specific heuristics and techniques for evaluating requirements.

Figure <ScenIC> shows the position that ScenIC occupies in the MORALE framework for evolving systems.

A .Input formats

ScenIC makes use of three types of input information: mission-oriented requirements (externally, from the stakeholders), the previous requirements specification (externally, from the stakeholders) and architectural responsibilities (internally to MORALE, from "Discover Architecture").

1 .Format of mission-oriented requirements

Mission-oriented requirements are a type of unstructured information about the proposed version of the system. We make no assumptions about the form that they take and assume that they are expressed informally in natural language. It is even possible for non-critical systems that some mission-oriented requirements are obtained orally, not in writing.

In the refinement descriptions in Sub-section C, we assume that the mission-oriented requirements exist in writing and that they are expressed in natural language. The description of ScenIC procedures is essentially similar if requirements are obtained through interviews or meetings or if they are partially expressed in the form of informal diagrams.

Mission-oriented requirements are not essential for ScenIC. We obviously cannot analyze the requirements for a system if we have none. However, in some situations (e.g. the elaboration of requirements for consumer software products), it may be best to conduct product visualization using the ScenIC procedures and start with the goal identification activities. In this guidebook, we assume that mission-oriented requirements have already been written.

A typical example of an input mission-oriented requirement for a new version of a meeting scheduler might read as follows:

> "The system should allow resources such as rooms to be scheduled"

1 .Format of previous requirements specification

Existing requirements are requirements for the current system. They are of interest in ScenIC for several reasons. First, the requirements for the existing version of the system, if they have been kept up to date, may tell us much of value about the rationale for the functionality of the current system.

In addition to this, we may wish to use the existing requirements expression as a template for elaboration when expressing the requirements for the new version of the system. Not only is it unnecessary to rewrite all the requirements from scratch, but we are sure to miss something.

Finally, the existing requirements document may provide a form and organization that we can use for the new requirements.

It is not necessary to have previously documented requirements to use ScenIC. In the special case of requirements for a new system there is no previous version of the system. And in many cases the requirements for the previous system may have been poorly documented or be unusable for other reasons.

1 .Format of architectural responsibilities

Architectural responsibilities in the current version of the system may constrain the expression of new requirements. For this reason, it is useful to have a specification of the current architecture.

The representation of architectural information is described in the SAAM and SBF guidebooks. In ScenIC we are interested only in high-level features of architecture. For our purposes, an architectural specification can be considered as a set of component/action pairs. Components are pieces, subsystems or modules of the system. Actions are requirements-relevant operations, usually externally visible, that have well-defined triggering conditions, intervals of execution, preconditions and effects.

```
ArchSpec == (Comp X Action X Allocation X ActionDef)
```

where

- b Comp is a set of component names (including environmental actors).
- b Action is a set of action names.
- b Allocation is a set of component/action pairs in which each component in Comp and each action in Action are mentioned at least once, and each action in Action is allocated to exactly one component.
- b ActionDef is a set of definitions for actions, in which each action is defined exactly once. An action definition consists of an optional triggering condition, an optional precondition, a post-condition and an optional duration specification.

The simplified architectural sub-language of SBF that is to be used as the input to ScenIC will be defined later. However, the following illustrates how a fragment of an architectural specification for a meeting scheduler might be represented:

```
Action: Schedule-meeting

Allocation: Automated-scheduler

Trigger: OCCUR(Drop-dead-date) OR RECEIVE(Last-reply)

Assumption: Meeting-called AND Can-schedule-given-replies

Effect: NEW Time(Meeting)
```

Components and actions should be defined at compatible levels of detail. Although a full architectural specification may contain layers or hierarchies of architectural descriptions, in ScenIC these are flattened at whatever level of detail seems appropriate from the stakeholders' external viewpoint. We envisage an activity in SAAM that simplifies an SBF specification into a ScenIC architectural specification.

As with the requirements for the previous version of the system, there does not have to be architectural specification. In the special case of a new system, there obviously cannot be such a specification.

A .Output formats

The outputs of ScenIC consist of requirements specifications and requirements rationale.

1 .Format of requirements specification

A ScenIC output requirements specification is similar to an input architectural specification (see sub-section A.3 above) with the addition of quality requirements.

The quality requirements specification sub-language will be defined later. An illustration of how a requirement for a meeting scheduler might be specified is given below:

```
Action: Schedule-room

Allocation: Automated-scheduler

Trigger: DONE(Scheduled-meeting-time)

Assumption: Some-room-available

Effect: NEW Room(Meeting)

Performance: t.b.d.
```

1 .Format of requirements rationale

As explained in Section II, there are three kinds of requirements rationale in ScenIC: a trace from a requirement back to the goals that originated it, a scenario that illustrates the need for the requirement (or the lack of need for a plausible alternative), and a structured record of a discussion that gave rise to the requirement (or failed to give rise to a plausible alternative).

A trace rationale is illustrated by the following:

```
Action: Schedule-room
```

```
Rationale: FULFILLS "Improve room utilization" STAKEHOLDER Facilities-
manager
```

A scenario rationale is illustrated by the following:

```
Action: Schedule-room
```

```
Rationale: OTHERWISE SCENARIO "Nowhere to hold scheduled meeting" EX-
PLORED-BY November JAD team
```

Finally, a structured record of a discussion is illustrated by the following:

```
Action: Schedule-room
```

```
Rationale: RESOLVES ISSUE "Are additional resources needed to hold a
meeting, and if so can the
```

```
Automated-scheduler seize them?" POSTED-BY Analyst-Fred AGREED-BY No-
vember JAD team.
```

An ongoing activity within MORALE is concerned with integrating the rationale capture and scenario exploration aspects of ScenIC with the Win-Win requirements negotiation method (Boehm & In, 1996).

A .Refinements

ScenIC does not consist of a series of steps that the analyst must perform in order. Rather, it consists of a collection of expression/discussion/refinement cycles that collectively transform the initial information into an acceptable requirements specification.

1 .Goal identification

```
Start-corpus: unstructured information
```

```
End-corpus: Start-corpus + Goal + Actor
```

Typically, unstructured requirements information contains statements at many incompatible levels of abstraction. Even though a requirements statement sometimes is expressed at a detailed level, it is still important to take all the requirements information and rationally reconstruct a higher-level statement of mission-oriented goals.

A goal description consists of a goal name, a goal category and an optional textual description. For example,

```
GOAL-231:

NAME "Improve-resource-utilization"

CATEGORY Improvement

DESCRIPTION "Improve utilization of such resources as space and high-
value equipment."
```

There are three major categories of goals: improvement goals, maintenance goals, and achievement goals. The first two types of goals are pervasive goals, because they pervade the requirements for the system. The third type is easier to localize and therefore trace forward to the requirements, because the achievement of a goal is generally the responsibility of a single actor or component. We find it useful to indicate the category of a goal in its name by using a small repertoire of category-suggesting verbs.

Improvement goals are concerned with improvement in some business or mission factor. Category-suggesting verbs that can be used to name improvement goals include "improve", "speed up", "reduce", "increase", etc.

Maintenance goals are concerned with the continuance of some desirable mission-oriented condition or, conversely, the avoidance of some undesirable mission-oriented condition. Category-suggesting verbs that can be used to name maintenance goals include "keep", "preserve", "maintain", "prevent", "avoid", etc. Achievement goals are concerned with the achievement of some desirable mission-oriented condition, such as a physical state or the knowledge of some information. Category-suggesting verbs that can be used to name achievement goals include "make", "know", "become", "have", etc. We recommend avoiding names for achievement goals that suggest a specific action for fulfilling the goal. Thus "Know room's schedule" is a better name for a goal than "Look up room's schedule" because it denotes a condition to be achieved, not an action to be performed.

-- How goals can be identified.

Closely related to goal identification is actor identification. Actors include entities in the environment with which the system interacts and components of the system architecture that are assumed to exist in the proposed system (usually because they exist in the previous version). Actors are named and categorized. Naming and categorizing actors is generally unproblematic. Environmental entities exist in the discourse of stakeholders and are therefore easy to name; architectural components generally have names derived from the design documentation for the previous system. We recognize the following categories of actors: End-user, Indirect-user, Organization, System, Device, and Component, where all but Component exist in the system's environment. The category of an actor is relevant when looking for obstacles later.

1 .Goal expansion

-- improvement goals into maintenance goals

-- maintenance goals into achievement goals

-- conjunctive sub-goals

-- disjunctive sub-goals

-- supporting goals

-- pervasive effects of improvement and maintenance goals

-- scenarios for normative cases

-- episodic scenarios

1 .Obstacle identification

-- achievement goals

-- obstacles not just negation of goals but causes thereof

-- many obstacles arise because of allocation decisions

-- categories of obstacle

-- actor failure obstacles

-- medium failure obstacles

-- resource contention obstacles

-- replicate confusion obstacles

-- risk analysis

1 .Obstacle defense

-- deciding whether to defend against obstacles

-- feasibility of defensive strategies

-- defending against actor and medium failure

-- defending against resource contention

-- defending against replicate confusion

-- role of scenarios in exploring defensive strategies

1 .Obstacle mitigation

-- deciding whether to mitigate obstacle effects

-- feasibility of mitigation strategies

-- mitigating actor and medium failure

-- mitigating resource contention

-- mitigating replicate confusion

-- role of scenarios in exploring mitigation strategies

1 .Responsibility allocation

-- allocation of goals to actors

-- allocation of actions to actors

-- responsibility allocation goes hand in hand with realization

1 .Goal realization

-- realization of achievement goals

-- realization of defensive strategies

-- realization of mitigation strategies

-- role of scenarios in allocation and realization

1 .Scenario exploration

-- not a refinement, but relevant to most refinements

-- scenarios as named situations

-- scripts

-- scripts as event traces

-- scripts as episodic stories

-- scenario coverage heuristics

-- recommendations about scenario details

A .Evaluation

1 .Evaluation of goal accuracy

-- techniques for assessing accuracy of goals from stakeholders

-- techniques for resolving inter-stakeholder goal conflicts

1 .Evaluation of obstacle accuracy

-- techniques for assessing accuracy of obstacles

1 .Evaluation of requirements completeness

-- techniques for assessing fulfillment of goals

-- techniques for assessing adequacy of obstacle defenses

-- techniques for assessing adequacy of obstacle mitigation

1 .Evaluation of absence of gold plating

-- techniques for isolating unnecessary requirements

-- *techniques for isolating unnecessary defense*

-- *techniques for isolating unnecessary mitigation*

-- *techniques for isolating unnecessarily complex realizations of goals*

1 .Evaluation of goal realizations

-- *techniques for proposing realizations*

-- *the role of architecture analysis in realization*

-- *techniques for evaluating alternative realizations*

1 .Evaluation of scenario coverage

-- *measurement of goal coverage*

-- *measurement of defensive coverage*

-- *measurement of mitigation coverage*

-- *proposed measures of co-occurrence coverage*

1 .Evaluation of scenario detail

-- *instances in scenarios*

-- *three levels of detail*

-- *sharing of scenario episodes*

1 .Evaluation of contextual fit

-- *not here but refer to Potts & Hsi*

I .Tutorial Example

A .Introduction to example

-- Description of simple web browsing example

-- Envisionment of social context of example

A .Rational analysis of example

-- I: What are the high-level goals?

-- R: Goal identification

-- I: What subgoals and dependencies?

-- R: Expansion of goals and goal dependencies

-- I: What idealizations?

-- R: Identification of intrinsic obstacles

-- I: What obstacle defenses?

-- R: Identification of defensive goals

-- I: What alternative realizations?

-- R: Responsibility allocations

-- R: Operationalization of software-allocated goals

-- I: What exceptions?

-- R: Identification of realization obstacles

-- I: What realization obstacle defenses?

-- I: What realization obstacle mitigation strategies?

-- R: Identification of realization-based defensive and mitigation goals

-- R: Identification of realization-based defensive and mitigation operations

A .Concrete envisionment of example

-- I: What are the normal-case uses?

-- R: Normative scenarios

-- I: What is supposed to happen?

-- R: Scripts for normative scenarios

-- I: What are the exceptional cases?

-- R: Single-obstacle and combined-obstacle scenarios

-- I: What

I .Glossary

-- Part of this should be common across the guidebooks

Allocated requirement: A requirement that has been allocated to a total system (usually, automated system) component.

Achievement goal: A goal that is described in terms of a desiredstate of affairs that does not yet hold.

Architect: A developer who is responsible for analyzing the requirements for the automated system, and proposing and analyzing a solution architecture or modifications to an existing architecture. (Note: This definition cuts across traditional distinctions between systems and software engineers and between analysts and software designers.)

Architectural constraint: A fact about the architecture of the legacy system that probably limits the requirements and putative architecture of the target system.

Architectural scenario: A scenario in which the roles are users and automated system components.

Architectural script: A script for an architectural scenario.

Architecture: An organization of automated system components that meets the requirements for the automated system and the affordances provided by its environment.

Automated system: A system in which one or more components are devices, software configuration items or other automated systems.

Canonical refinement: A standard refinement pattern.

Component: A part of a system. A component may be regarded as a system in its own right or as an atomic agent.

Context: Concrete aspects of the system's environment that affect its likely success.

Customer: A stakeholder who represents the organization that is intended to benefit most from the total system. (Note: the contractual client may not be a customer in this sense).

Defensive goal or requirement: A goal or requirement that exists to reduce the likelihood of an obstacle occurring. The limiting case is a preventative goal or requirement.

Description: The activity of representing and documenting some aspect of a system.

Design Journal: The repository containing the current requirements and architecture of the target system, the history of refinements that led to it, and the rationale.

Developer: A stakeholder who is a member of an organization responsible for the development and evolution of the automated system.

Discussion: The activity of discussing, analyzing or reviewing a description.

Elaboration: A refinement that converts an idealized or simplified goal or requirement into one that accounts for obstacles in the target system or its environment.

Environment: The totality of things not in a system that interact with or influence it.

Expansion: A refinement that converts a goal into interdependent subgoals.

Fact: An assumed or verified statement about the world that affects the system requirements and which the system must be designed to accommodate but which is not a requirement on the system itself.

Goal: A desired purpose of a system or component.

Goal web: A description (probably graphical) of a collection of goals and their dependencies.

Human-Activity System (HAS): A system in which the components are people, teams or organizations. These system components may use automated tools to help them meet their goals, but the system does not contain such tools as principal components.

IBIS: Issue-based IS. A framework for conducting and representing design deliberations and for recording design rationale.

Inquiry Cycle: A collection of three kinds of activity involved in designing and planning a system, viz.: description, discussion and refinement.

Legacy system: An existing software system that is determined by a customer to be in need of improvement.

Maintenance goal: A goal described in terms of a desired state of affairs that already holds.

Mission-oriented goals & requirements: Goals and requirements that stem from the mission or context in which the target system is to be used, rather than those arising out of purely technical inadequacies of the legacy system.

Mission-oriented scenario: A scenario in which the structure of the scenario is related to the goals of the system or components in question.

Mission-oriented script: A script for a mission-oriented scenario.

Mitigation goal or requirement: A goal or requirement that exists to reduce the harm caused by an obstacle when it occurs. The limiting case is a recovery goal or requirement.

Obstacle: A state of affairs that can impede the fulfilment of a goal.

Operational requirement: A requirement that has been allocated to a software component and that is described in enough detail to be implemented in source code.

Owner: A stakeholder who defines the success or otherwise of the total system and has the authority to cancel its development.

Procedural representation: A representation of requirements or system architecture that emphasizes temporal behavior rather than system purpose.

Putative architecture: An architecture put forward by an architect as a viable design to meet the mission-oriented requirements or as an explanatory structure for the source code of a legacy system.

Quality goal: A goal described in terms of a trajectory of improvement in some criterion.

QOC: Questions/options/criteria. A framework for conducting and representing design deliberations and for recording design rationale.

Rationale: The reason or reasons for a decision, or the reasoning surrounding an issue that has not yet been decided.

Realization: The refinement process that converts goals into allocated operational requirements.

Refinement: The revision of one or more descriptions as a result of discussion. Refinements include the creation of new descriptions that are either subordinate to existing descriptions or otherwise dependent on them.

Requirement: A property of an automated system needed by a customer stakeholder.

Responsibility: An allocation of a goal or an operational requirement to a component.

SAAM: Software architecture assessment methodology(?). A method that uses scenarios to evaluate putative architectures.

SBF: Structure/Behavior/Function architecture modeling framework.

Scenario: An imagined set of concrete system behaviors in response to a state of affairs.

Script: A description of a scenario in which the component's actions are represented in time-ordered sequence.

Software requirement: A requirement that has been allocated to a software system or component.

Software system: A system that is to be implemented in software components.

Stakeholder: A person, group of people or with a stake in the success of the total system.

Synchronized refinement: An expectation-driven method for analyzing source code and documentation to recover a software architecture.

System: A collection of active components whose coordinated behavior meets a set of goals.

Target system: A system that is to be implemented in software and hardware components.

Teleological representation: A representation of requirements or scenarios that emphasizes purpose, goals or objectives, rather than process.

Total System: A large, complex system including one or more human-activity systems and one or more automated systems.

User: A stakeholder who is a member of a human-activity system in the environment of the automated system who directly interacts with the automated system. Direct interaction may take the form of operating a computer worksation, but could equally involve interaction with ubiquitous or embedded automated system components or the consumption of reports or other information digests generated by the automated system. A user may perform a collection of activities that have no direct counterpart in the human-activity system that exists prior to the implementation of the automated system.

I .Annotated References to the Literature

How this method relates to other methods.

Discriminators

for each of:

b -- Process models for requirements analysis

b -- Coherent methodologies (SADT, SSADM,...)

b -- Goal refinement methods (including SSM)

b -- Scenario-based design

b -- Design rationale models

Links to further information

-- Previous papers

MORALE METHODOLOGY GUIDEBOOK
Software Architecture Analysis Method (SAAM)

Gregory D. Abowd
College of Computing
Georgia Institute of Technology
abowd@cc.gatech.edu

Len Bass, Paul Clements and Rick Kazman
Software Engineering Institute
Carnegie Mellon University
firstname.lastname@sei.cmu.edu

1. Introduction

1.1 Purpose

Because of the importance of architectural decisions, it is fitting that they receive close scrutiny. In particular, few will argue that it is more cost-effective to evaluate software quality as early as possible in the life cycle. If problems are found early in the software life cycle, they are trivial to correct—a change to a requirement, or specification, or design is all that is necessary. Software quality cannot be appended late in a project; it must be inherent from the beginning, built in by design. It is in the project's best interest for prospective design candidates to be evaluated (and rejected, if necessary) during the design phase, before long-term institutionalization occurs. Therefore, an effective technique to assess a candidate architecture—*before* it becomes the project's accepted blueprint—is of great economic value. Furthermore, when *acquiring* or *evolving* a large software system that will have a long lifetime within the acquiring organization, it is important that an organization develop an understanding of the underlying architecture of the candidates. This allows an assessment of the suitability of the candidates with respect to qualities of importance.

On the other hand, it should be noted that an architecture cannot guarantee the functionality or quality required of a system. Poor downstream design, implementation, testing, or management decisions can always undermine an acceptable architectural framework. Decisions at all stages of the life cycle—from high level design to coding and implementation—will affect quality. Therefore, quality cannot be completely assessed on the basis of an architectural description. An architecture-based quality assessment provides only one dimension of a system's quality characteristics and is a necessary, but not sufficient, component of evaluating the overall quality of a system. It assesses the ability of the architecture to facilitate the desired qualities. Refinements of the architecture into implementation that preserve the qualities are necessary for the final product to actually achieve those qualities.

We present a method, the Software Architecture Analysis Method (SAAM), for predicting system level quality attributes based upon software architectural evaluation. SAAM makes use of scenarios derived from the different roles involved in a computer system (administrator, user, maintainer, etc.) in order to evaluate competing architectures. We illustrate SAAM in the context of a simple but classic software architecture example, a key-word-in-context system. We then demonstrate this method in a larger case study. We present an architectural analysis of three well-known Internet information systems—WWW, WAIS and Harvest

1.2 The overall MORALE architecture

(common across guidebooks)

1.3 Overview of method

Inputs to and outputs from the evaluation:

If the architecture being evaluated is not yet instantiated in a system—as ideally, it should not yet be—then what is the basis for the analysis? Obviously a description or specification of the architecture is required. Since, as we have seen, architectures may be described in many levels of detail and from many views, what kind of description or specification is needed for evaluation? The answer is whatever is needed to answer the questions imposed by the evaluation technique. If the evaluation will delve into issues of performance or parallelism, than a description of the architecture's tasking and communication structure will probably be required. If modifiability is an issue, then the architecture's decomposition into modules or work assignments will be needed. And so forth. SAAM starts by using whatever rendition of the architecture is available, and proceeds to make whatever evaluation it can based on that. If other views or finer-grained depictions of the architecture are needed, SAAM will make their need clear as it progresses.

What does an evaluation produce? As we shall see in the next chapter, results vary across methods. SAAM produces a relative ranking. That is, if two or more candidate architectures are being compared to see which one satisfies its quality requirements more fully, SAAM will produce a ranking of the candidates. If a single architecture is being evaluated, SAAM will point out places where that architecture fails to meet its quality requirements, and will in some cases show obvious alternative designs that would fare better. The results are coarse and broad; this reflects the variance inherent in performing architectural analysis at a very coarse grain.

In no case are absolute numbers on some mythical scale of "architecture excellence" produced. SAAM is predicated on the principle that there is no such scale; an architecture's suitability depends entirely on the specifics of the context in which it is being developed.

The coarse analysis provided by SAAM is justified by its low cost and the ability to achieve it very early in the software development life cycle.

Analyzing Qualities with Scenarios:

People often want to analyze software architectures with respect to quality attributes expressed using words like *maintainability, security, performance, reliability*, and so forth. These words provide convenient ways for describing and communicating a host of common, recurring problems in software. There are even IEEE standards that define many of them. However, most software quality attributes are too complex and amorphous to be evaluated on a simple scale, in spite of our persistence in describing them that way. Consider the following examples:

Suppose a system can change the background color of the user interface merely by changing a data file, but that same system requires a manual change to dozens of programs in order to accommodate a new data storage layout. Would you say that this system is or is not *modifiable*?

Suppose the user interface to a system is carefully thought out so that a novice user can exercise the system with a minimum of training, but the experienced user finds it so tedious as to be inhibiting. Would you say that this system is *usable* or not?

The point, of course, is that quality attributes do not exist in isolation, but rather only have meaning within a context. A system is modifiable (or not) with respect to certain classes of changes, secure (or not) with respect to specific threats, usable (or not) with respect to specific user classes, efficient (or not) with respect to its utilization of specific resources, and so forth. Statements such as "This system is highly maintainable" turn out to be entirely without operational meaning.

This notion of context-based evaluation of quality attributes has led us to adopt *scenarios* as the descriptive means of specifying and evaluating quality attributes. SAAM is a scenario-based method for analyzing architectures; it provides a means to characterize how well a particular architectural design responds to the demands placed on it by a particular set of scenarios, where a scenario is a specified sequence of steps involving the use or modification of the system. It is thus easy to imagine a set of scenarios that would test what we normally call modifiability (by proposing a set of specific changes to be made to the system), security (by proposing a specific set of threat actions), performance (by proposing a specific set of usage profiles that tax resources), etc.

A particular scenario actually serves as a representative for an entire class of scenarios. This class consists of all scenarios for which the system under consideration responds in exactly the same way. For example, the scenario "change the background color on all windows to blue" is under most system designs essentially equivalent to the scenario "change the window border decorations on all windows". SAAM measures, among other things, how many evaluation scenarios cause changes to the same architectural component; this is called *clustering*. As a consequence, judgment needs to be exercised as to whether the clustered scenarios represent variations on a similar theme or whether they are substantially different. In the first case, the clustering is a good thing; in the second, it is bad. In other words, if a group of scenarios are similar, and they all affect the same component or components in an architecture (i.e., they cluster), that is a good thing, because it means that the system's functionality has been modularized in a way that properly reflects the modification tasks. If, on the other hand, a group of similar scenarios affect many different components throughout an architecture, that is bad. We return to this theme after SAAM has been presented in more detail.

As an aid to creating and organizing scenarios, we appeal to the concept of *roles* related to the system. Examples of roles include: the person responsible for upgrading the software—the end user; the person responsible for managing the data repositories used by the system—the system administrator; the person responsible for modifying the run-time functions of the system—the developer; the person responsible for approving new requirements for the system, etc. The concept of roles matches the difference between run-time qualities and developmental qualities, i.e., those qualities that are a function of the system's execution (such as performance), and those that reflect operations performed off-line in a development environment (such as modifiability). Scenarios of the former variety would be performed by roles such as end user; the latter by developers or maintainers.

In summary, SAAM uses scenarios, or concrete descriptions of expected use of a system, as benchmarks to compare and contrast different candidate architectures. The purpose of SAAM is to provide a way to validate claims about system quality at the architectural level of description and not through vacuous appeal to abstract terms ("Our system is scalable"). SAAM relies on a description of all candidate architectures that identifies the important components and connections and the overall coordinated behavior of the system. These descriptions are then measured against an agreed set of scenarios to determine the extent to which the candidate architecture supports each scenario or must be modified in order to support it. Candidate architectures are then compared against each other by noting how they "perform" for similar scenarios.

Overview of SAAM:

In this section, we define the steps of a SAAM evaluation, and discuss the interactions among them. Not every SAAM evaluation need go through all of these steps. In fact, projects can benefit from performing almost any subset, as some of the case studies will show.

Develop scenarios. Develop scenarios that illustrate the kinds of activities the system must support and the kinds of changes which it is anticipated will be made to the system over time. In developing these scenarios, it is important to capture all important uses of a system. Thus scenarios will represent tasks relevant to different roles such as end user, customer, marketing, system administrator, maintainer, and developer.

Describe candidate architecture(s). The candidate architecture or architectures should be described in a syntactic architectural notation that is well-understood by the parties involved in the analysis. These architectural descriptions need to indicate the system's computation and data components, as well as all relevant connections. SAAM evaluations have tended to use very simplistic architectural primitives. A typical representation will distinguish between data (passing information between components) and control (one component enabling another component to perform its function) connections. This simple lexicon provides a reasonable static representation of the architecture. Accompanying this static representation of the architecture is a description of how the system behaves over time, or a more dynamic representation of the architecture. This can take the form of a natural language specification of the overall behavior or some other more formal and structured specification.

Classify scenarios. There is an important distinction between scenario types that we introduce at this point. Recall that a scenario is a brief description of some anticipated or desired use of a system. It may be the case that the system directly supports that scenario, meaning that anticipated use requires no modification to the system in order to be performed. This would usually be determined by demonstrating how the existing architecture would behave in performing the scenario (rather like a simulation of the system at the architectural level). If a scenario is not directly supported, that means that there must be some change to the system that we could represent architecturally. This change could be a change to how one or more components perform an assigned activity, the addition of a component to perform some activity, the addition of a connection between existing components, or a combination of the above. We refer to the first class of scenarios as *direct* scenarios and the second class as *indirect* scenarios. A particular scenario is only direct or indirect with respect to a particular architecture. One candidate architecture may execute a scenario directly while another may require modification before it can be executed. When using SAAM in an acquisition effort, whether scenarios are direct or indirect is often of great interest and often unknown to the customer before the analysis begins. Obviously, an architecture for which the scenarios of interest are directly supported is more capable than one that requires modification to support those same scenarios, and will score higher under SAAM.

Perform scenario evaluations. For each indirect scenario, list the changes to the architecture that are necessary for it to support the scenario and estimate the cost of performing the change. A modification to the architecture means that either a new component or connection is introduced or an existing component or connection requires a change in its specification. By the end of this stage, there should be a summary table that lists all scenarios (direct and indirect). For each indirect scenarios the impact, or set of changes, that scenario has on the architecture should be described. A weighting of the difficultly must also accompany this stage. Usually, this weighting is coarse grained, based on the understanding of the architecture. A tabular summary is especially useful when comparing alternative architectural candidates because it provides an easy way to determine which architecture better supports a collection of scenarios.

Reveal scenario interaction. When two or more indirect task scenarios require changes to a single component of a system, they are said to *interact* in that component. Scenario interaction is important to highlight because it exposes the allocation of functionality to the product's design. The interaction of semantically unrelated scenarios explicitly shows which system modules are computing semantically unrelated functions. Areas of high scenario interaction reveal potentially poor separation of concerns in a system component. Thus, areas of scenario interaction indicate where to focus the designer's subsequent attention. The amount of scenario interaction is related to metrics such as structural complexity [7], coupling, and cohesion [8]. Therefore, it is likely to be strongly correlated with number of defects in the final product.

Overall evaluation. Finally, assign a weight to each scenario and the scenario interactions in terms of their relative importance and use that weighting to determine an overall ranking. The purpose of assigning weights is to resolve the situation where the first candidate architecture scores well on half the scenarios, and the second candidate architecture scores better on the other half. By understanding which scenarios are more important, ties can be broken. Assigning weights is a subjective process, involving all of the stake-holders in the system. Rather than offering a single architectural metric, SAAM produces a collection of small metrics, a set of per-scenario analyses. Given this set of mini-metrics, SAAM can be used (and in fact was developed with the intent to) compare competing architectures on a per-scenario basis. It is left to the users of SAAM to determine which scenarios are most important to them, in order to resolve cases where the candidates out-score each other on different scenarios. It has also been the case that the process of performing a SAAM analysis has been used to gain a higher level understanding of the competing architectures; this higher level understanding, rather than just a scenario based table, is useful to perform comparative analysis. The case study of Section 4 emphasizes this aspect.

Crafting the scenarios and representing the architecture(s) are inter-dependent steps. Deciding the appropriate level of granularity for an architecture will depend on the kinds of scenarios you wish to evaluate (though not all scenarios are appropriate, such as a scenario dealing with the number of lines of source code). Determining a reasonable set of scenarios also depends on the kinds of activities you expect the system to be able to perform, and that is also reflected in the architecture.

Figure 8 illustrates the steps in a SAAM evaluation.

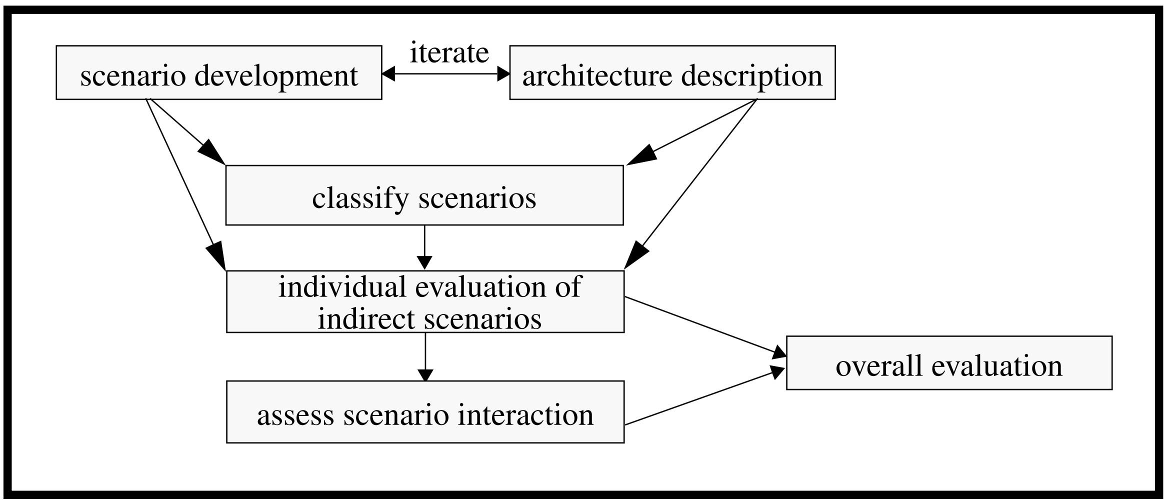


Figure 8: Activities and dependencies in SAAM analysis

1.4 A small example: KWIC

In this section, we apply SAAM to a small, decidedly academic application to provide an easily-understood example of each of the steps of SAAM. The application is the construction of a keyword-in-context (KWIC) index generator. It is the example that Parnas used in his 1972 landmark paper that introduced the concept of information-hiding [[REF]]. The application is small, easily understandable, and familiar. Thus, it makes a good example to illustrate the method. This example has also been used by others to illustrate points about software engineering and software architecture. In this example, the KWIC system has been completely specified beforehand. This would not always be the case when performing an architectural analysis.

The message of Parnas' paper in 1972 was the virtue of information-hiding as the design principle around which to organize a system's functionality. Parnas compared two designs for KWIC (one "traditional" and one developed using the information-hiding principle) and showed the advantages of the latter over the former. SAAM takes many of the ideas that are implicit in that paper and makes them explicit by virtue of its methodical approach to evaluation.

The behavioral requirement for the KWIC program is to take as input a list of sentences, where each sentence is an ordered set of words. The program outputs, in alphabetical order, a listing of all circular shifts of all sentences. So, for example, if the following sentences were given as input:

predicting software quality
architecture level evaluation

the KWIC program would produce, as output:

architecture level evaluation
evaluation architecture level
level evaluation architecture
predicting software quality

quality predicting software
software quality predicting

Defining Roles and Scenarios:

The KWIC example has two roles of interest: the *end user* who wishes to produce the KWIC index and the *developer* who produces or modifies the code that produces the KWIC index.

Two of the scenarios we use that have been discussed by various authors involve the end user:

1. Modify the KWIC program to operate in an incremental rather than a batch fashion. This version of the program would work by accepting one sentence at a time, and producing an alphabetical list of all permutations of all sentences input to date.

2. Modify the KWIC program to eliminate entries beginning with “noise” words (articles, prepositions, pronouns, conjunctions, etc.).

The other two scenarios we will use involve the developer but are not apparent to the end user:

3. Change the internal representation of the sentences (e.g., compressed or uncompressed).

4. Change the internal representation of intermediate data structures (e.g., either store the shifted sentences directly or store an index to the shifted words).

Describing Candidate Architectures:

The next step of the method is to represent the candidate architectures that are to be evaluated, using a common notation. This is done in order to ease the evaluation process so that the evaluator knows exactly what a box or arrow means in the architectural diagram.

Shared memory solution: I

n the first candidate architecture, there is a global storage area, called *Sentences*, where all of the input sentences are stored. The input sentences are: read by the *Input* routine; stored in *Sentences*; shifted by the *Circular Shift* routine (which builds a *Shift Index* of the starting locations of each of the shifted permutations of the original input sentence, rather than storing the sentences themselves in shifted form); alphabetized by the *Alphabetize* routine (which stores an alphabetized version of the *Shift Index*); and then output. *Master Control* passes control to the various routines as required.

Figure 9 is annotated with the evaluations from the scenarios (the numbers in the various computational components). In this step we can simply ignore the annotations.

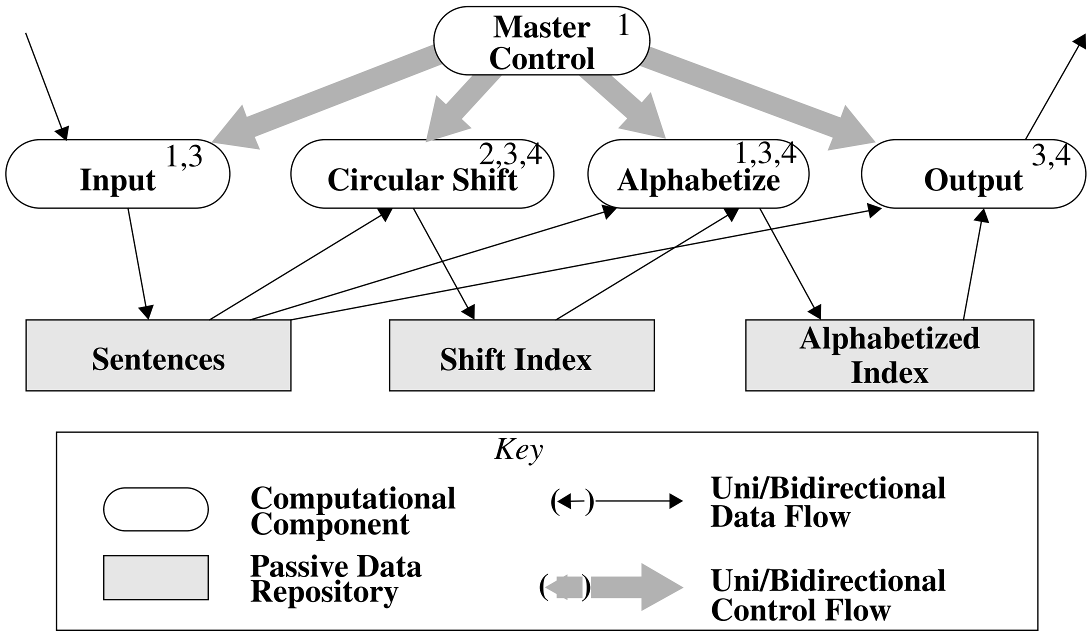


Figure 9: KWIC—shared memory solution

Abstract data type solution:

The second architecture, represented in Figure 10, uses an abstract data type (ADT) where each function hides and protects its internal data representation and provides specialized access functions as the only way to store, retrieve, and query the data. The *Sentence* ADT has access functions *set* and *getNext* to add and retrieve a sentence. The *Shifted Sentences* ADT provides the *setup* access function to set up its circular shifts of the sentences,[1] and the *getNext* function to retrieve the shifted sentences.

The *Shifted Sentences* ADT makes use of the *getNext* function from the *Sentence* ADT to retrieve stored input sentences. The *Alphabetized Sentences* ADT provides a *setup* function and an *i-th* function.The *setup* function calls *Shifted Sentences*'

1. As both Parnas [24] and Garlan and Shaw [17] point out, the ADT solution admits of other possibilities. For example, the *setup* function in the *Shifted Sentences* ADT may in fact do nothing when it is called, and all shifts are done "on demand." That is, whenever the *Shifted Sentence getNext* function is called, the ADT gets another sentence from the *Sentences* ADT and shifts it. However, for the purposes of architectural analysis, some assumptions must be made about how the functions will work together. SAAM assumes that each function is accompanied by a specification, so that when doing an architectural analysis, these decisions are known.

getNext function repeatedly to retrieve all the lines to be sorted and sorts them. The *i-th* function, called specifying some integer parameter *i*, returns the i^{th} sentence from the sorted list of *Shifted Sentences*.

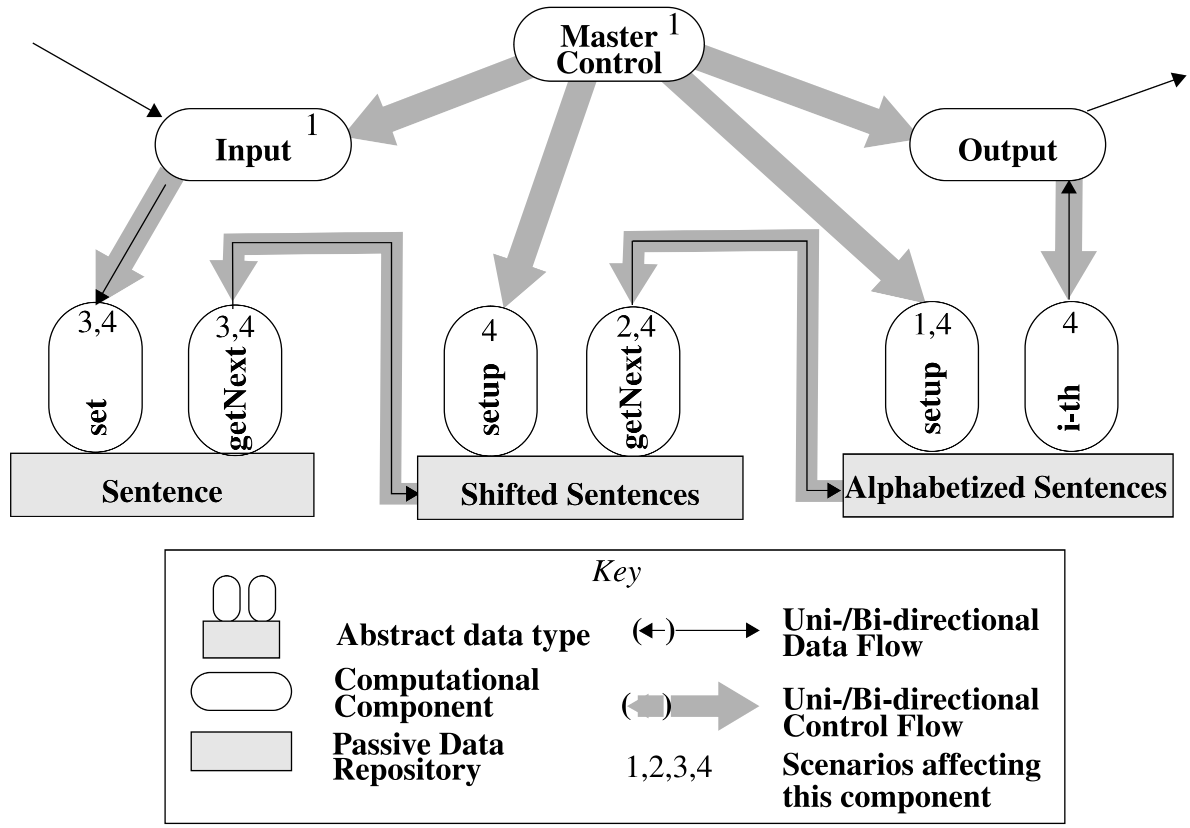


Figure 10: KWIC—ADT solution

Classifying the Scenarios :

Now that we have specified the candidate architectures in a common notation, the next step in the method is to classify the scenarios as direct or indirect against each architecture. In this case, each of the scenarios which we have chosen for evaluation of the KWIC architectures is indirect. None can be directly executed by either candidate architecture so that all of the evaluation is dependent upon some modification to the architecture.

Evaluating the Candidate Architectures:

The next step in the method is to evaluate the architectures against the scenarios. The evaluation begins by considering each of the scenarios in turn.

Scenario 1:

The first scenario is to move from a batch mode to an incremental mode. That is, operate on one sentence at a time rather than inputting all of the sentences and operating on the total input.

For the shared memory solution, this requires a modification of the *Input* routine, so that it yields control after each sentence. *Master Control* must be modified because the sequencing among the subordinate routines is going to be repetitive rather than calling each routine only once. The *Alphabetizer* will need to be modified because an incremental algorithm, such as insertion sort, must be used rather than one that sorts all of the sentences at once. Presumably, the *Circular Shift* routine operates on one sentence at a time and the output function outputs whatever is available whenever it is called.

Note the assumptions we are making about the specification of the routines in the shared memory solution. In general, the accuracy of the judgments are dependent on knowledge of the internal workings of the various computational components. That is why we expect the evaluation to be performed by individuals either with general knowledge of the computational components or with access to the specification of the components.

For the ADT solution, shown in Figure 10, the *Input* function needs to be changed so that only one line at a time is input each time it is called. The *Sentence* abstraction is assumed to be implemented to relinquish control when it has stored its input. Since it receives input one sentence at a time, this need not be changed. The *Shifted Sentences* abstraction is also assumed to be implemented to request and shift all available sentences when it is invoked. Thus, this routine also does not need to be modified. The *Alphabetized Sentences* abstraction must be modified as in the shared memory example.

For this scenario, since both candidate architectures have been affected equally, we rate the two architectures as a tie.

Scenario 2:

The second scenario is the elimination of noise words. This change is most easily performed by modifying the shifter function in both architectures (*Circular Shift* in the shared memory architecture and *Shifted Sentences* in the ADT architecture) to eliminate those shifted sentences beginning with noise words. Since the shifter function is localized in both architectures, and since noise word elimination does not affect the internal representation of the sentences, the difficulty of making this modification is equivalent for the two architectures.

Scenario 3:

The third scenario is the modification of the internal representation of an input sentence, e.g., moving from an uncompressed representation to a compressed representation.

In the shared memory architecture all functions share a common representation, and so all of the functions are affected by this scenario, except for the *Master Control*. In the ADT architecture, the input sentence internal representation is buffered by the *Sentence* abstraction. Thus, the ADT architecture is better than the shared memory architecture for this scenario.

Scenario 4:

In this scenario—changing the representation of the intermediate data structures—we observe that the shifting and alphabetizing functions maintain indices to the input sentences and only one copy of the input sentence exists.

For the shared memory example, this requires a modification of the *Circular Shift*, the *Alphabetizer* and the *Output* routines.

For the ADT solution, this requires a modification of the *Shifted Sentences* and the *Alphabetized Sentences* ADTs. Thus the ADT solution affects fewer components and is superior for this example.

Scenario interaction analysis:

Both figures have been annotated to reflect the impact of each scenario. For example, the “1” inside the *Master Control* component in Figure 9 reflects the change to that component required to support Scenario 1. Examining the candidate architectures to determine which one has more contention of scenarios (components with more than one scenario annotation) we see that the ADT is superior in this regard. While both architectures show scenario contention in four components, in the shared memory solution the contention is among three of the scenarios in two components (*Circular Shift* and *Alphabetize*), whereas no component in the ADT solution has contention from more than two tasks.

0.1 Summary of evaluation

Table 22 summarizes the results of the SAAM evaluation. A “0” indicates that neither architecture is superior with respect to this scenario. In a real evaluation, the evaluation would conclude by weighting the scenarios to represent the organization’s priorities. For example, if the addition of functions was the primary concern (as represented by scenario 2) then the two architectures would be ranked closely, since they do not differ with respect to this task.

If, on the other hand, a change in the internal representation of sentences (as represented by scenario 3) was of primary concern to the organization, then the ADT solution would be superior.

Table 22: Summary of evaluation

Architecture	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Contention
Shared Memory	0	0	-	-	-
Abstract Data Types	0	0	+	+	+

2. A Larger Example (Internet information systems)

Internet Information Systems (IISs) are large network-based distributed software systems which are increasingly important in both academic and commercial computing. Some examples of these systems are the World-Wide Web, Gopher, WAIS, and Archie. Significant amounts of resources are being devoted to creating information repositories hosted by servers and clients to browse the repositories. Given the importance and virtual ubiquity of these systems, it is natural to want to compare and contrast them. Some researchers have provided overviews and comparisons of the various IISs. We will briefly review some of that previous work.

Schwartz et al. [27] provided an early taxonomy of resource discovery tools. They provided a textual description of nine different IISs and then defined four different characteristics (granularity, distribution, interconnection topology and data integration scheme) that are used to classify how a given IIS handles both data and metadata. This survey provides a good set of scenarios (browsing, searching and organizing) from the perspective of both the consumer and provider of information. The descriptions of each system is fairly informal and the level of detail provided varies between. There is also no clear connection between these descriptions and the taxonomic classification provided at the end of the paper. Our approach is not taxonomic; we do not intend to provide a description of the whole space of possible IIS designs. We do, however, want to provide a means of comparing systems. Schwartz et al. do the comparison by means of how systems differ on the general characteristics for data and metadata. Our comparison will be based on much less general characteristics, but will be performed on more consistent and formal system descriptions.

Obraczka, Danzig and Li [25] provide another taxonomic approach to describing 11 different IISs across 8 different characteristics. Their taxonomy extends that of Schwartz et al. with the addition of four yes/no characteristics (query, browse, organize, directory of services) that are used to determine if an IIS provides some capability or service. These last four characteristics are similar in flavor to the scenarios that we propose in this paper, except that we want to go further to explain how a given architecture must be modified in order to provide some capability or service. They explicitly describe the architecture of a number of these systems using the familiar box-and-line diagram approach. A brief glance at the different architectural diagrams will show that there is no common interpretation for the diagrams and it is therefore very difficult to do any analysis based on the architecture. This is more a reflection on the immaturity of software architectural description languages than a poor reflection on designers.

There are also plenty of technical reports that provide details on any single IIS, some of which we make use of in this paper. But the overriding concern with all of those papers is that they either provide too much implementation detail to be communicated effectively to a more general design audience or they do not provide enough detail (vague box-and-line diagrams) in order that a reader could establish the suitability of the design for its intended purposes. Another purpose of this report is to provide an example of suitably high level yet sufficiently detailed architectural descriptions that can be the basis of analysis across a rapidly evolving domain such as IIS.

2.1 Scenarios for IIS

This section defines a set of scenarios according to the roles that they affect. These scenarios attempt to unify other scenario descriptions (Mealling, Daniels & NCSA repository requirements & Library Group at IITA] as well as contribute new tasks and clarify existing roles. Within the IIS domain, the roles of interest can be specifically named: **information consumer**, a person who uses an IIS to search for or browse information from one or a collection of data repositories; **information provider**, a person who wants to make some information available on the Internet either as a stand-alone repository or as an addition to an existing repository; and **system administrator**, a person who modifies or monitors the infrastructure of the IIS. Here we provide a name and a brief description of the scenario that will be the basis of later evaluation. Several more scenarios were done in the actual case study, but we will only do the analysis here for a subset.

Information consumer scenarios:

C.1. Resource location

The user knows where a document is located and wants to be able to retrieve it, e.g. the user knows the document is located on the W3C's ftp server in /pub/www/arena/README.

C.2. Resource discovery

The user wants a particular kind of information, but does not know the information's location, e.g. a user might want to find a verified copy of Xmosaic_2.6a, or to find online publications by a certain author within the past year.

C.3. Browsing

Browsing, or serendipitous discovery ([10]), involves undirected behavior. For example, a user might want to look to see what information is available across various sites.

Information provider scenarios:

P.1. Adding information of a supported type

The provider wants to include information of a supported type into an existing repository.

P.2. Adding lots of information of a supported type

An important generalization of the previous scenario is to ask how difficult it is to support many instances of the above task.

P.3. Creating new information of a type not currently supported

The provider wants to create new information of a type (FrameMaker, PhotoShop, etc.) not currently supported by this IIS.

System administrator scenarios[1]:

A.1. Security from consumers

The administrator wants to prevent information consumers from corrupting the repository, i.e. causing accidental or intentional changes to the repository that compromise the integrity of the information from an external consumer's perspective.

A.2. Security from providers

The administrator wants to prevent information providers from corrupting the repository in the same manner as described above.

2.2 Candidate architectures

We will now present an architectural description of three IISs, the WWW, WAIS, and Harvest. The architectural description consists of a modified data flow diagram of the system, indicating gross-level computational and data components[2], and the control and data connections between them, together with a textual description of the overall behavior of the system. Though this is not a fully formal definition, care was taken to provide an accurate representation of each system at a similar level of detail. In creating these descriptions, the authors referred to recent literature, inspected code, and consulted IIS domain experts.

The descriptions distinguish between data and control connections, even though these are frequently implemented by the same mechanism, e.g. a procedure call. A data connection between two components indicates that tokens of data are

1.These scenarios do not refer to encryption, traffic analysis, or any other information content based notion of security. Rather, the security scenarios listed deal directly with the architectures vulnerability to various roles of intruder.

2.We also use boxes in the diagrams to delineate important system boundaries, such as the client-server distinction.

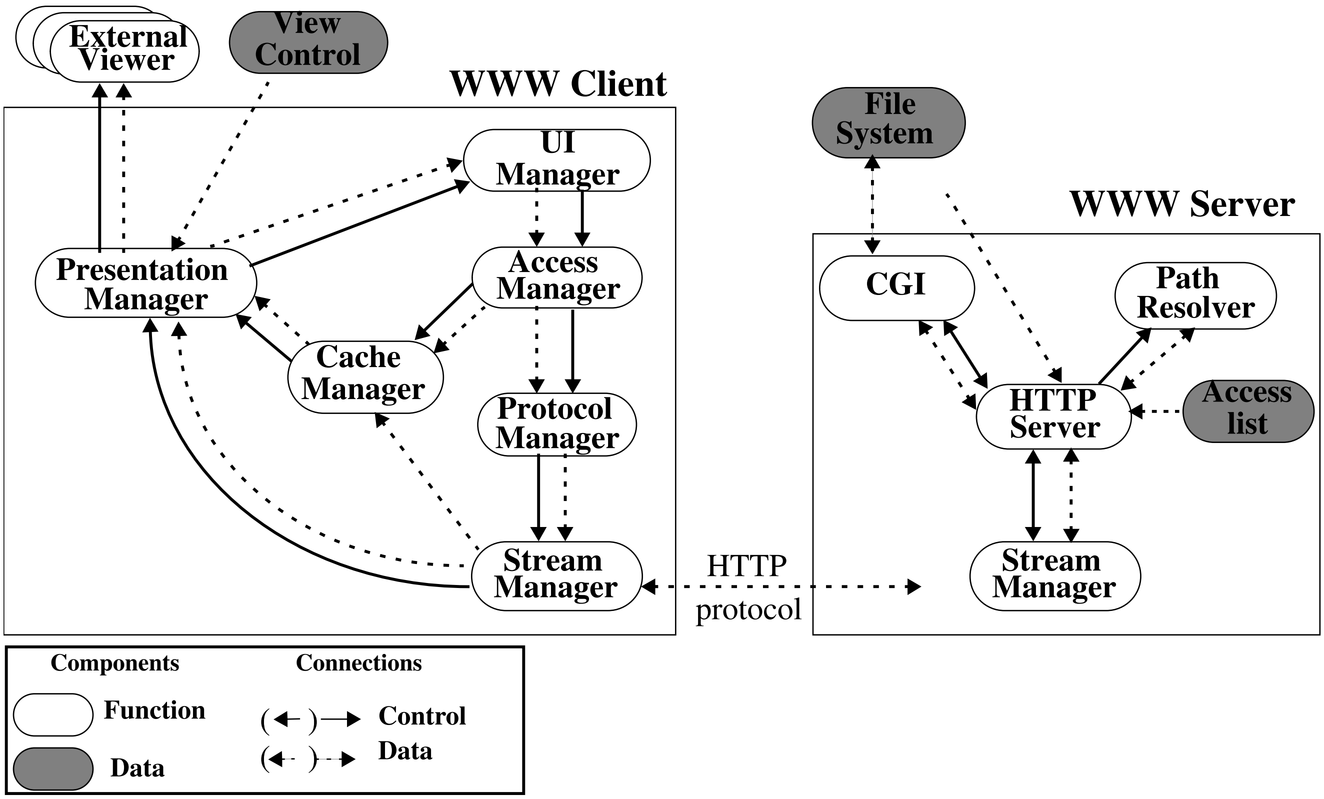


Figure 11: WWW architecture

passed between these components. A control connection between two components indicates that one component causes the other to perform some of its main computation. For example, when one executes a procedure call in a standard procedural language, one typically passes data tokens as the arguments and return value of the procedure call and a “life” token that enables the called procedure to become active, with the calling procedure being suspended until the called procedure returns the “life” token. Some connection types involve only a single type of connection. For example, TCP/IP connections are pure data: neither procedure can force the other procedure to become active or inactive by sending or not sending data. On the other hand, a Unix “virtual fork”, is mainly a control connection.

WWW/Mosaic:

The Web is a distributed hypermedia system organized as a loosely connected set of clients and servers that share a common set of communication protocols and markup languages. Servers make Internet resources available to a community of clients that speak a common protocol. In addition, WWW clients typically understand a number of other protocols ([4],[5],[6],[16]). Figure 11 depicts the WWW software architecture. In this example, we only consider the connection between a WWW client and WWW server, even though it is possible for a WWW client to natively speak to other IISs (Gopher, FTP, etc.).[1]

WWW clients provide a User Interface Manager.[2] This manager can be graphical or character based. The User Interface Manager captures user’s request for information retrieval in the form of a Uniform Resource Locator (URL) and passes the information to the Access Manager. The Access Manager determines if the requested URL exists in cache and also interprets history-based navigation, e.g. ‘back’. If the file is cached, it is retrieved from the Cache Manager and passed to the Presentation Manager for display to either the User Interface or an external viewer. If the file is not cached, the Protocol Manager determines the type of request and invokes the appropriate protocol suite to service the request. This protocol is used by the client Stream Manager for communicating the request to the server. Once the client Stream Manager receives a response from the server in the form of a document, this information is passed to the Presentation Manager for appropriate

1.The use of the Common Gateway Interface (CGI) provides another way for a WWW server to simulate other access protocols while still communicating with the WWW client using HTTP. This ability does not reduce the overall functional capability of WWW system presented.

2.In this description and later descriptions of WAIS and Harvest, we use capitalization to refer to components in the architectural description.

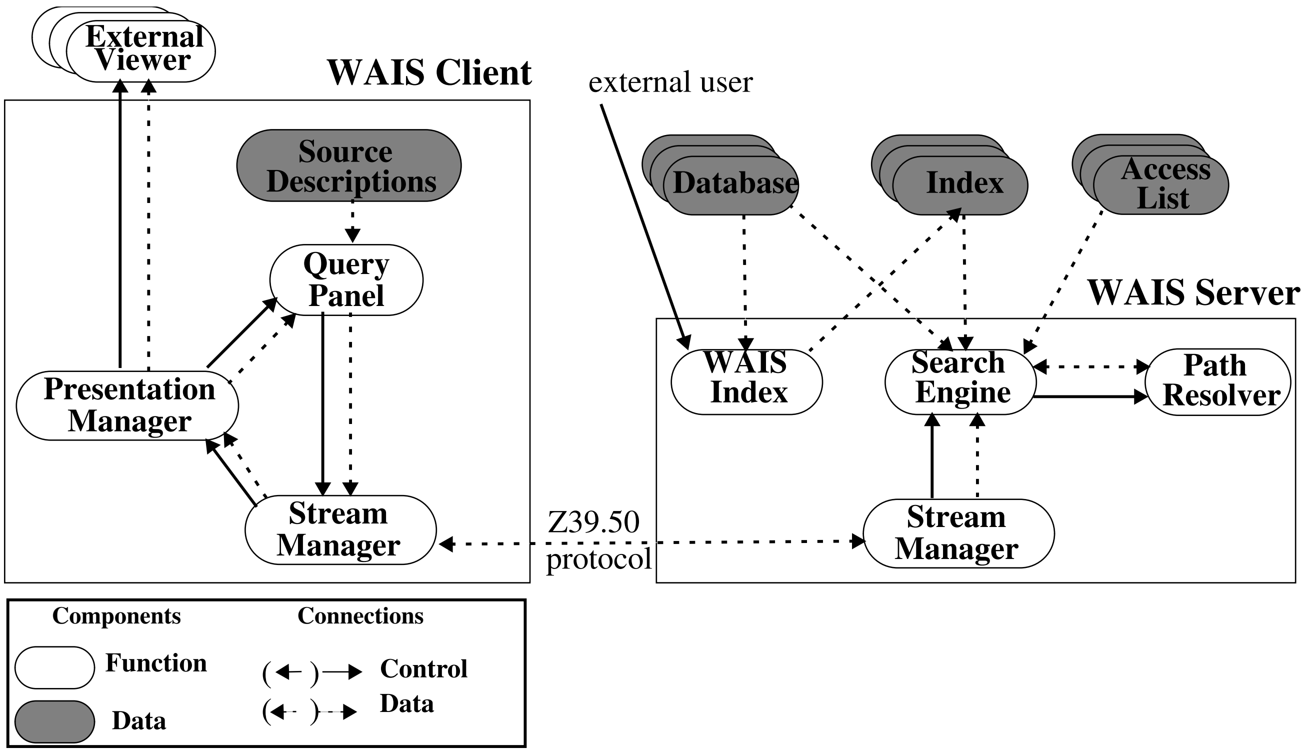


Figure 12: WAIS architecture

display. The Presentation Manager consults a static View Control configuration file (mimerc, mailcap, etc.) that aids in the mapping of document types to external viewers.

Currently, the WWW servers available implement a subset of defined HTTP requests. This subset allows for: the retrieval of documents; the retrieval of document meta-information; and server-side program execution via the Common Gateway Interface (CGI).

When a request is received by the server Stream Manager, the type of request is determined and the path of the URL is resolved via the Path Resolver. The HTTP Server consults an Access List to determine if the requesting client is authorized to access the data pointed to by the URL. The HTTP Server might initiate a password authentication session with the client to permit access to secured data. Assuming authentication succeeds, the HTTP Server accesses the File System (which is outside the WWW Server boundary) and writes the requested information to stream. If a program is to be executed, a process is made available (either new or polled) through the CGI and the program is executed, with the output written by the server Stream Manager back to the WWW Client.

WAIS:

Wide Area Information Server (WAIS)[1] is a network publishing and retrieval system designed to help users find distributed networked information using a mixture of natural language and boolean queries ([11], [29],[30]). Communication between client and server adheres to the NISO standard protocol Z39.50. Searching for information via the WAIS client-server model is a two step process. The directory of services—a centralized index of all registered WAIS servers—is queried to locate potential databases that contain information on a user-specified topic. The user then selects a subset of these potential databases and formulates a query to locate specific relevant documents. WAIS ranks all returned hits for a query, indicating which documents most closely meet the user's query. WAIS allows for an iterative search strategy in which a user can indicate that subsequent searches on a database should look for files that most nearly match some previously identified file. Once the user finds a document of interest, a request can be made to retrieve that document from its database. The architecture of WAIS is depicted in Figure 12.

The WAIS client provides the user interface as a Query Panel that is used to formulate a query and indicate the data sources (WAIS databases, including the Directory of Services) that the user wishes to search. The Query Panel also displays the ranked results of queries. Each Database provides descriptive information including its location, name and sum-

1.WAIS and Wide Area Information Servers are both registered trademarks of WAIS Inc. There are public domain versions of WAIS, called freeWAIS and ZDist. For the purposes of this report, the differences between these systems is not significant. Most of the technical documentation for freeWAIS has been consulted in our analysis, but we will still refer to the system as WAIS.

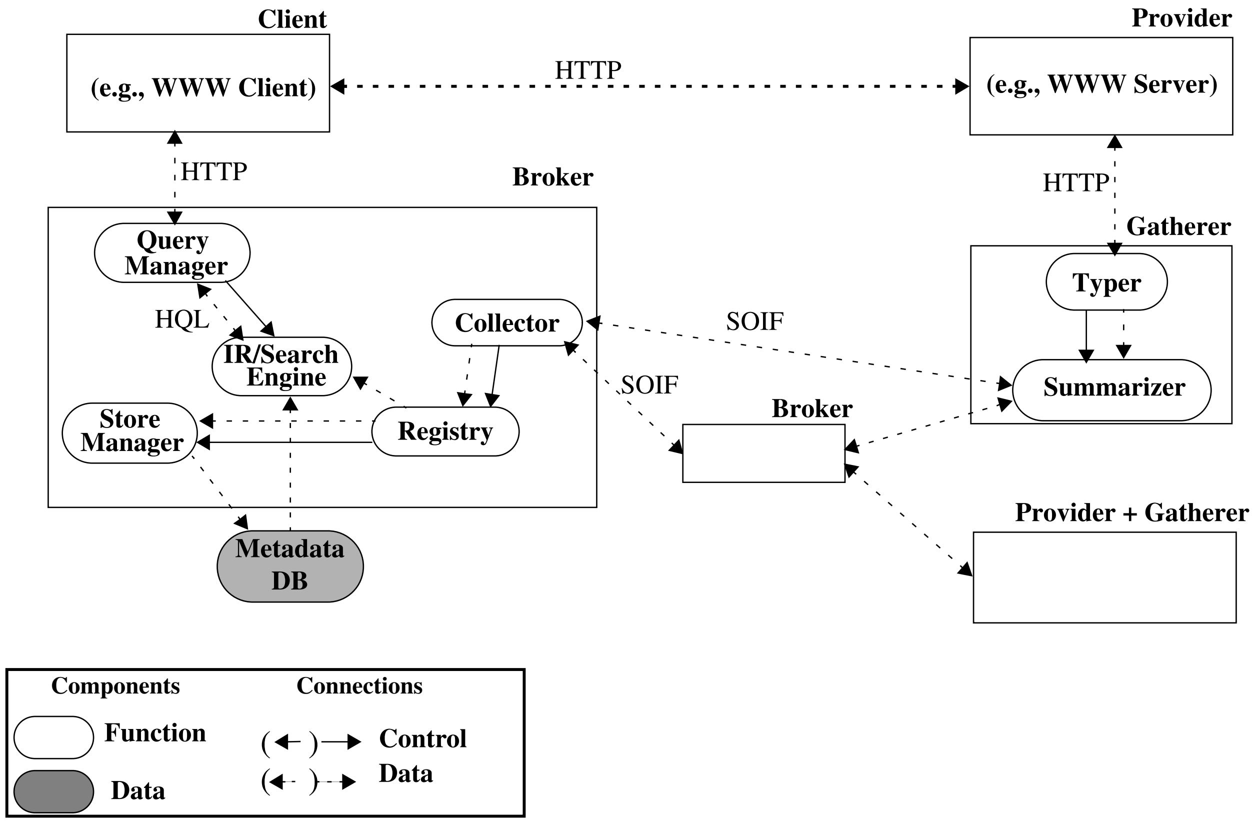


Figure 13: Harvest architecture

mary of contents which is accessible through the consumer-browsable Source Descriptions file. This information is used to establish the connection from the client to the server. After the user formulates a query, it is passed to the client's Stream Manager, which handles communication with the WAIS Server. Information returned to the client's Stream Manager is then passed to the Presentation Manager, which either sends the query result to the Query Panel or directs the document information to an External Viewer, thus enabling the viewing of various document formats.

The WAIS Server services the requests from the WAIS Client. Two types of requests are serviced: queries and document retrievals. Each server can host more than one database. Each database has an associated inverted keyword Index and Access List. This Index facilitates the matching of queries to documents. The Access List identifies those clients with permission to access the databases. For a query, the Search Engine determines which documents are relevant for the search using the Index and Access List. It then retrieves the documents from the Database itself and ranks the documents according to some relevance criteria based on the user's query.[1] The Search Engine then writes to stream an ordered list of documents with header information (including relevance to the original query) to the client for display to the user. For document retrieval, the client sends a request to the server, the Search Engine checks access privileges and retrieves the file if access by the client is allowed, compresses the data and writes it to stream.

Harvest:

Harvest is a resource discovery tool which leverages off of much of the functionality of the Web [8]. Unlike the Web, it attempts to provide mechanisms to facilitate global resource discovery amongst heterogenous Internet based services like FTP, Gopher, UseNet News, & the WWW. While Harvest does not introduce any new client technologies (WWW Browsers are the default Harvest client) and many components speak existing protocols (HTTP) and generate existing markup (HTML), Harvest does provide an architecture for global search and retrieval across existing network-based, information providers.

The architecture of Harvest (shown in Figure 13) which extends WWW is based up the notion of information "Gatherers" and "Brokers." A Gatherer resides either in an existing repository server (e.g., a WWW server or WAIS server) or interacts

1.Though the WAIS documentation suggests that a WAIS query can span across several Databases at different sites, we do not see in the documentation of this system any indication that the relevance ratings span across sites, that is, the individual WAIS Servers all produce relevance criteria for information in their databases and no attempt is made in the WAIS Client to do further ranking.

remotely via a network protocol (currently HTTP) and collects summaries of metadata for various Brokers to store. This collection is done by the Typer recognizing types of information in the repository and then the Summarizer extracting useful content metadata. Gatherers and Brokers use a specialized protocol (SOIF) for efficient communication and storage of metadata. Information from one Broker to another is replicated in a hierarchical manner, this organization being maintained by a Replication Manager (not shown). Periodically, the Collector component in the Broker will request updates of metadata from each Gatherer or Broker it services. Any updates are passed to the Registry, which records unique object identifiers and time-to-live values for the data. Identifiers are provided to the IR/Search Engine and the Store Manager is requested to archive a copy of summary metadata in the Metadata DB on the file system for later retrieval by the IR/Search Engine.

The Broker responds to queries from clients (currently only WWW clients) by returning the location (URLs) of the items of interest. The query is accepted by the Query Manager, which then translates the request into the Harvest Query Language (HQL) and passes it onto the IR/Search Engine. The IR/Search Engine then provides a response to the query by accessing the Metadata DB according to its own searching strategy.

0.2 Scenario evaluations

In this section, we will evaluate the three candidate scenarios against each of the consumer and provider scenarios defined in Section 2.1. For reasons of space, we will not discuss the administrator scenarios here.

Information consumer scenarios:

C.1. Resource location

WWW: The user can explicitly communicate the known URL via the User Interface to the Access Manager. This scenario, therefore, is directly supported.

WAIS: The notion of a universal name does not exist with WAIS, therefore the user would have to know the Database where the information resides as well as the handle to the file that contained the information. This scenario, therefore is only indirectly supported by the architecture. In order to support this scenario, the Query Panel would have to be modified to allow sending explicit document retrieval directives without first having to perform a search.

Harvest: Since Harvest is currently built on top of WWW clients, this scenario is directly supported.

C.2. Resource discovery

WWW: The current Web implementation does not directly enable users to find information on a topic. It is possible for WWW to provide other protocol services, including WAIS, either via CGI or the common library of protocols.

WAIS: WAIS supports a two-phase search strategy which allows a narrowing of the search to a set of potential databases and then an actual search of those databases. The effectiveness of this two-pass strategy depends on the accuracy of meta-information stored in the directory of services. We conclude that WAIS directly supports this scenario.

Harvest: The arrangement of topic-specific Gatherers and Brokers allows Harvest to directly support this scenario.

C.3. Browsing

WWW: Browsing is enabled to the extent that the User Interface component makes available locations of information (or URLs) at various places on the Web. WWW supports this scenario directly, though there is no support for organized browsing.

WAIS: A limited form of browsing is possible in WAIS. The Source Descriptions file is accessible from the Query Panel. But this information is restricted to a particular site's repository. It is also possible to view a catalog on each server, but this is still more restrictive than the scenario suggests and cannot be explicitly requested by the Query Panel. Any form of browsing to be supported by WAIS would have to modify the Query Panel and the Search Engine.

Harvest: This scenario is direct for the same reasons as WWW. We note that it is not currently possible to browse metadata in the Brokers, but the scenario was not referring to that capability.

0.2.1 Information provider scenarios

P.1. Adding information of a supported type

WWW: The Web servers attempt to be a MIME Content Type File System, so any existing MIME typed digital information can be made accessible. Permission to write new information to the File System is the only other provider requirement. Once that information is in the File System, it can instantly be accessed. This scenario is directly supported by WWW.

WAIS: The repositories are the Databases that the Server accesses. These are typed, allowing WAIS to perform its indexing, query response and text retrieval more efficiently. The entries in the databases are stored as files in the underlying file system, so there is no real difficulty introducing a new piece of information (again, assuming the provider has correct file system privileges). Even though the Index must be updated to make the new data available to the consumer, this scenario is directly supported.

Harvest: The information can be added to the repository, just as with WWW. The Gatherer must then detect the presence of the new information (either by automatic polling or explicit invocation) to extract the SOIF summary. When the Broker next requests updates from the Gatherer, this new information will be passed along and made available to a consumer. This scenario is direct, though we now have two points of delay in making information available to the user.new information has to be gathered and then forwarded on upon request by an interested Broker.[1]

P.2. Adding lots of information of a supported type to a single repository

WWW: The architecture indirectly supports this scenario. Efficient support of this scenario hinges on how well the underlying operating system supports adding numerous new files. Modifications would be required to the File System.

WAIS: Adding lots of information to the Database is similar to the WWW case, depending on the operating system for efficiency. The Index allows for incremental updating, meaning that the whole Index does not have to be recomputed after every addition. What is not clear from the technical documentation on the Index is how efficient the incremental indexing is over time. We therefore consider this scenario indirectly supported with potential bottlenecks at the Database and Index.

Harvest: Similar to WWW with added bottlenecks at the Gatherer and Broker. If Gatherer is local to repository, then efficient automated gathering can occur. If Gatherer is remote, then there is a network bottleneck. Broker-Gatherer communication depends on performance of high-level SOIF protocol, which is communicated in compressed form to speed transmission. There is an additional step involved here which is equivalent to the WAIS step for updating the Metadata DB for use by the IR/Search Engine. The efficiency of that step depends on the IR/Search Engine in place (which could be WAIS Search Engine).

P.3. Creating new data repository of a type not currently supported

WWW: Since WWW servers attempt to use MIME Content Types, the addition of an unregistered type would result in no header information being passed to the Client, unless the HTTP Server were modified. If the client was unable to type the file, it would default to ‘application/octet-stream’. If the HTTP Server adds a new MIME Content Type that the Client does not understand, the behavior of the client is undefined, unless the View Control is modified and an External Viewer application is provided.

WAIS: Modifying WAIS to support this task would require changing the Index because it currently needs to have the format of new types hard coded as a parsing routine.

Harvest: Harvest relies on having a SOIF format for data in order to operate effectively. For new types of information, the provider can use the Essence system [19] to change how the Typer recognizes data types and how the Summarizer extracts metadata. Since Essence is used at the core of defining the behavior of the Typer and Summarizer, we consider this scenario directly supported.

1.Both WAIS and Harvest introduce delays in making new information available to the consumer. Though immediate availability of information is an important feature of an IIS, it is not a matter considered by the P.1 scenario, and so we ignore it in our overall ranking of the candidates for this scenario.

2.3 Overall IIS evaluation

The results of the scenario evaluations are given in Table 23. The final three columns indicate the extent of architectural

Table 23: Summary of evaluation of WWW and WAIS

Scenario	**Changes/additions necessary**		
	WWW	**WAIS**	**Harvest**
Information consumer			
C1. Resource location	+	- Query panel	+
C2. Resource discovery	- Protocol Manager CGI	+	+
C3. Browsing	+	- Query Panel Search Engine	+
Information provider			
P1. Adding information of supported type	+	+	+
P2. Adding lots of information of supported type	+ File System	0 Database WAIS Index	- Provider repository Gatherer Broker-Gatherer SOIF Broker
P3. Creating new data of a type currently unsupported	- HTTP Server View Control External Viewer	0 Database WAIS Index	+

modification that is required for indirect scenarios, and rank the three architectures. Architectures which support the scenario directly are given a ranking of +. Architectures which require changes (i.e. those architectures for which the scenarios are indirect) are ranked according to the number of affected components.

This table can also be used to reveal the scenario interactions discussed as Step 4 in Section , but we will not discuss that here. We also do not consider weighting the scenarios in this report, as that depends on the intended use of the IIS, which can vary based on consumer, provider and administrator preferences. Given a relative ranking of scenarios, however, it should be clear how one could use the summary analysis in Table 23 to justify choosing one IIS over another.

3. Detailed Methodology Description

Already done.

3.1 Input formats

???

3.2 Output formats

???

3.3 Steps

Already done by example

3.4 Evaluation

???

4. Glossary

(part of this should be common across guidebooks)

5. Annotated References to the Literature

[1] Abowd, Gregory, Jonathan Engelsma, Luigi Guadagno and Okonon Okon. (Significant contribution from all authors, principal authorship by Engelsma and Abowd) Architectural Analysis of Object Request Brokers. To appear in Object Magazine special issue on distributed systems, March 1996, pp. 44-51.

[2] Abowd, G., Kazman, R. and Pitkow, J. (Significant contributions from Abowd and Kazman with Abowd as principal author) Analyzing Differences Between Internet Information System Software Architectures, Proceedings of ICC '96, (Dallas, TX), June 1996.

[3] Len Bass, Paul Clements, Linda Northrop, Amy Zaremski, Gregory D. Abowd and Rick Kazman (principal contributions from Zaremski and Abowd. Principal authorship by Abowd) Industrial best practice and recommendations for software architecture evaluation. Submitted to *IEEE Computer*. February, 1997.

[4] T. Berners-Lee, R. Cailliau, J.-F. Groff, "The World-Wide Web", *Computer Networks and ISDN Systems,* **25**, North-Holland, 1992, pp. 454-459.

[5] T. Berners-Lee, R. Cailliau, J. Groff, B. Pollermann, "World-Wide Web: The Information Universe", *Electronic Networking: Research, Applications and Policy*, 2(1), Meckler Publishing, 1992, pp. 52-58.

[6] T. Berners-Lee, R. Cailliau, A. Luotonen, H. Nielsen, and A. Secret, "The World-Wide Web", *Communications of the ACM*, 37(8), 1994, pp. 76-82.

[7] C. M. Bowman, P. Danzig, U. Manber, M. Schwartz, "Scalable Internet Resource Discovery: Research Problems and Approaches", *Communications of the ACM,* 37(8), 1994, pp. 98-107.

[8] C. M. Bowman, P. B. Danzig, D.R. Hardy, U. Manber, M. F. Schwartz and D. P. Wessels, "Harvest: A scalable, customizable discovery and access system", Department of Computer Science, University of Colorado, Technical Report CU-CS-732-94. March, 1995 revision of original August 1994 draft.

[9] Brown, A; Carney, D.; & Clements, P. "A Case Study in Assessing the Maintainability of a Large, Software-Intensive System." *Proceedings of the International Symposium on Software Engineering of Computer Based Systems*. Tucson. IEEE Computer Society, March 1995.

[10] L. Catledge and J. Pitkow, "Characterizing Browsing Behaviors on the World-Wide Web", *Computer Networks and ISDN Systems,* **27**, North-Holland, 1995 (in press).

[11] The Center for Networked Information Discovery and Retrieval, "freeWAIS Technical Documentation", On-line documentation. URL ftp://ftp.cnidr.org/pub/NIDR.tools/freewais/.

[12] Clements, P., Bass, L., Kazman, R. and Abowd, G. (Significant contributions by Clements and Bass with Clements as principal author) Predicting software quality by architecture-level evaluation. In the proceedings of *Fifth International Conference on Software Quality*. Austin, TX, October 1995.

[13] P. R. Daniel and M. Mealling, "URC scenarios and requirements", IETF Internet Draft report, June 27, 1995. URL http://www.acl.lanl.gov/URI/Scenarios/.

[14] M. De Simone, R. Kazman, "Using SAAM: An Experience Report", *Proceedings of CASCON '95*, (Toronto, ON), November, 1995, 8 pages, to appear.

[15] H. Frystyk, "WWW Library Internals—Overview", On-line documentation, URL http://info.cern.ch/hypertext/WWW/Library/User/Guide/Overview.html.

[16] H. Frystyk, H. Lie, "Towards a Uniform Library of Common Code: A Presentation of the CERN World-Wide Web Library", *Proceedings of the Second International World-Wide Web Conference,* 1994.

[17] D. Garlan, G. Kaiser, D. Notkin, "Using Tool Abstraction to Compose Systems." *IEEE Computer 25*, 6 (June1992): 30-38.

[18] D. Garlan and M. Shaw, An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific Publishing, 1993.

[19] D. R. Hardy and M. F. Schwartz, "Customized information extraction as a basis for resource discovery", Department of Computer Science, University of Colorado, Technical Report CU-CS-707-94, March 1994.

[20] R. Kazman, L. Bass, G. Abowd, S.M. Webb, "SAAM: A Method for Analyzing the Properties Software Architectures", *Proceedings of the 16th International Conference on Software Engineering*, (Sorrento, Italy), May 1994, pp. 81-90.

[21] Kazman, R., Abowd, G., Bass, L. and Clements, P. (Significant contributions and authoring by all) Scenario-based analysis of software architecture, *IEEE Software*. **13**(6):47–56, November 1996.

[22] R. McCool, “The Common Gateway Interface”, On-line documentation, URL http://hoohoo.ncsa.uiuc.edu/cgi/overview.html.

[23] B. Neuman, “The Virtual System Model: A Scalable Approach to Organizing Large Systems”, PhD Thesis, University of Washington, 1992.

[24] D. Parnas, “On the Criteria To Be Used in Decomposing Systems into Modules.” *Communications of the ACM 15*, 12 (1972): 1053-1058.

[25] K. Obraczka, P. Danzig, and S. Li, “Internet Resource Discovery Services”, *IEEE Computer,* **26**(9), 1993, pp. 8-22.

[26] D. Perry and A. Wolf. “Foundations for the study of software architecture”, *SIGSOFT Software Engineering Notes*, **17**(4), October 1992, pp. 40-52.

[27] M. Schwartz, A. Emtage, B. Kahle, B. Neuman, “A Comparison of Internet Resource Discovery Approaches”, *Computing Systems,* **5**(4), 1992, pp. 461-493.

[28] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, G. Zelesnik, “Abstractions for Software Architecture and Tools to Support Them.” *IEEE Transactions on Software Engineering 21*, 4 (April 1995).

[29] WAIS, Inc., “WAIS Server, WAIS Workstation, WAIS Forwarder for Unix: Technical Description”, Technical Description, Release 1.1, 1993.

[30] WAIS, Inc., “WAIS for UNIX”, On-line documentation, URL http://www.wais.com/company/Tech_TOC.html.(How this method relates to other methods, discriminators, links to further information)

MORALE METHODOLOGY GUIDEBOOK
Methodology Guidebook for Synchronized Refinement

Spencer Rugaber
College of Computing
Georgia Institute of Technology
spencer@cc.gatech.edu
(404) 894-8450

Abstract

Synchronized Refinement is a methodology for reverse engineering software for the purpose of documentation, reengineering, or reuse. Reverse engineering produces a high-level representation of a software system from a low-level one, typically the program itself. Synchronized Refinement constructs such a representation for a system from its source code. It proceeds by detecting design decisions in source code and relating the detected decisions to the intended functionality of the system. Examples are given demonstrating the application of the methodology to the reverse engineering of production software systems.

1. Introduction

1.1 Purpose

To effectively evolve a system, you must understand it. And that understanding must be expressed in a form with which high level (architectural) decisions about the evolution of the system can be made. *Synchronized Refinement* is a method for obtaining such an understanding and expressing it in a form that aids evolution. The process of gaining understanding and expressing it is called *reverse engineering*. Chikofsky and Cross [Chikofsky] define reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." Reverse engineering can be used for a variety of purposes: to reconstruct or otherwise improve documentation; to facilitate software maintenance or conversion activities; or to redesign and evolve an existing system.

Reverse engineering is a difficult and costly activity. The gap between the low-level source code expressing *how* computations are performed and a high-level description of *why* a particular construct is employed comprises both differences in levels of abstraction and domains of discourse. These two differences are confounded by the fact that understanding process is not uniform, moving as it does between the abstraction levels and the domains.

The Concrete and the Abstract: Computer programs are incredibly detailed. In essence they control the values of million of bits of memory inside of a computer. One of the jobs of the reverse engineer is to decide, from all this detail, which are the important concepts. This process is called abstraction; the reverse engineer must create an abstract representation of the program from the mass of concrete details. Compounding the problem is the fact that the abstraction process is not linear. That is, a given section of a program may be a part of several abstractions. The occurrence of such abstractions can be called *interleaved* or *delocalized* in the section [Pinto Letovsky Documentation Delocalized]. Typically there is no explicit indication in the source code of the interleaving. And detecting the abstractions in the first place involves complex pattern recognition. in which, unfortunately, some of the abstractions reside in the application domain where they are less accessible to the reverse engineer.

The Application Domain and the Program Domain: Programs are models or representations of problem situations from some application domain. There may or may not be hints in the program about the domain. Hints can take the form of mnemonic variable names and in-line comments. The hints are inherently informal and tend to be out-of-date with respect to the program. Because of this, totally automatic reverse engineering tools are restricted to working with the formal program text. As Brooks points out, it is the job of the reverse engineer, to reconstruct the mappings from the application domain to the program domain [Ruven Brooks Comprehension]. This of course requires knowledge, not only of programming, but also of the application domain. It is no surprise, therefore, to find that most automatic tools are restricted to analyzing the program text and do not address the application domain.

Top-Down and Bottom-Up: When a reverse engineer looks at a program, he or she is detecting patterns that indicate the intent of some section of code. Low level patterns are part of higher level constructs intended to accomplish larger purposes. In this way, the process of analyzing a program proceeds bottom-up. At the same time, the programmer has some idea of the overall purpose of the program and how it might be accomplished. As the program is perused, the overall concept is refined into lower level details. This synthesis process proceeds top-down. The difficulty is that both of these activities need to proceed at the same time and in a synchronized fashion.

This document describes *Synchronized Refinement* (SR) a reverse engineering methodology that simultaneously analyzes source code and synthesizes an application description. The analysis is based on detecting examples of characteristic design decisions in the code. The synthesis is driven by a formal description of the application domain. The resulting represention expresses the relationship between the two by instantiating the domain description with the detected code. The representation can then be used to support further software evolution via reengineering.

1.2 MORALE

SR is part of the MORALE software evolution methodology. MORALE is a method and tool suite that supports software evolution. It exploits the symmetry between requirements analysis and software understanding. When analyzing the requested changes for a new version of a system, the old version must be considered and every effort made to reuse existing infrastructure. Likewise understanding the old version can be greatly aided by an application-oriented description of the goals that the system is trying to accomplish, as obtained from its requirements specification. MORALE uses the system architecture as a common ground to synthesize the two sources of knowledge and behavioral traces (called *scenarios* at the requirements stage and *event traces* at the code understanding stage) as the common medium of comparison. The MORALE architecture can naturally be broken up into three activities, one each for requirements analysis, architectural evaluation, and reverse engineering. The relationships between these activities and the artifacts relied upon and produced are depicted in the data flow diagram shown in Figure 14.

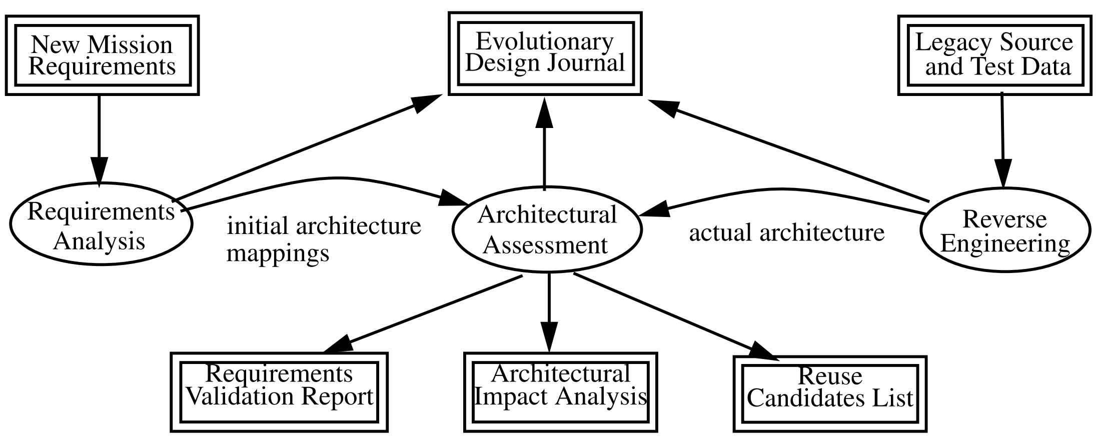


Figure 14 MORALE inputs, activities and outputs

The initial inputs to the MORALE process are the existing system in the form of legacy source code and test data, a statement of the new requirements that the system must meet in the form of mission-oriented goals, and (possibly) an architectural representation of the existing system. Requirements analysis uses the mission-oriented statement to suggest concepts of importance in the overall structuring of the system; that is, it provides a suggested taxonomy for an initial architectural description of the system in the case where no such architecture exists. Requirements analysis also provides a collection of scenarios that represent a complete description of the new system requirements. These serve as the basis for the architectural impact analysis. The architectural assessment predicts the impact a set of scenarios have on an architecture. Reverse engineering provides techniques for assuring that the architectural representation, the basis for the impact analysis, is an accurate reflection of the actual system under scrutiny. The three pieces work together to predict the extent to which the architecture of the old system can be adapted to meet the new requirements.

Besides the architectural impact analysis, MORALE produces several other artifacts useful in planning system evolution: a list of candidate code components from the old version that may be reused in the new version and a report on the extent to which the behavior of the existing system version is reflected in the requirements for the new version. Additionally, the MORALE suite provides a historical evolutionary design journal that incorporates items such as system goals and problems, traceability information between requirements and code, design alternatives and rationale. The result of the

MORALE process is the development of instantiated architectural components together with a record of design decisions and the mission-oriented goals that were met or traded off to develop the new system.

1.3 Overview of Synchronized Refinement

The overall SR process is presented in Figure 2.

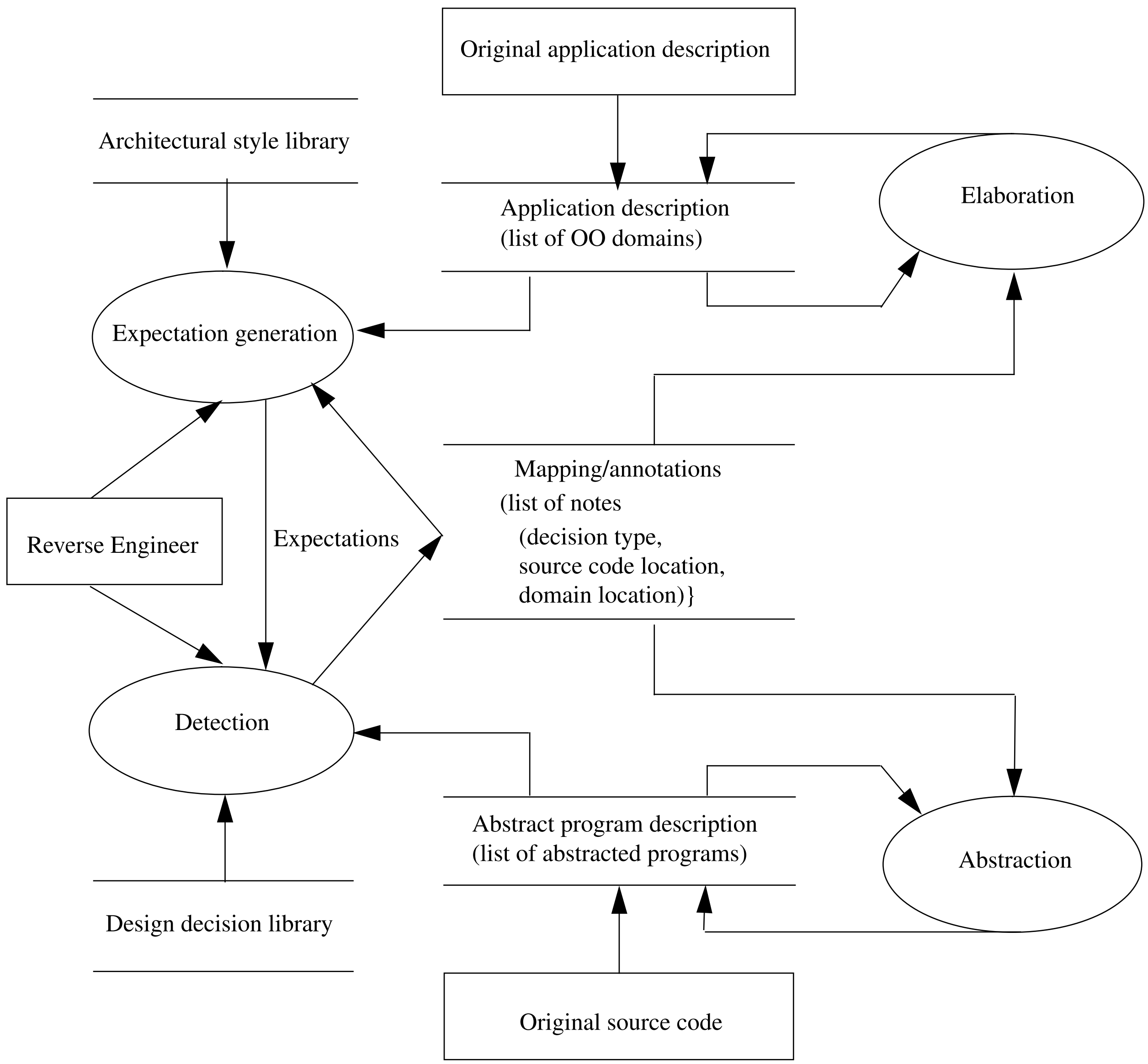


Figure 15 Data Flow Diagram for Synchronized Refinement

Synchronized Refinement takes as input the source code of a software system (labelled **Original source code** in Figure 2) and a description of the application domain that the system supports (**Original application description**). In addition, various sources of programming knowledge may be available to the analyst to aid the understanding process. These are labelled **Architectural style library** and **Design decision library** in Figure 2. The output consists of three parts: an elaborated and instantiated domain description (**Application description**), an abstracted program description (**Abstract program description**), and a description of how the code realizes the application (**Mapping/annotations**). The SR process consists of two parallel activities, roughly corresponding to the top and bottom halves of the diagram: analysis of the program (on the bottom) and synthesis of the application description (on the top). The analysis process consists of detecting design decisions and replacing the related code by an abbreviated version. In this way, the program description continues to become smaller and more abstract. The synthesis process begins with the domain description that is elaborated as more is learned from the source code. The two parts of the diagram communicate in terms of *expectations*. That is, domain concepts must be ultimately realized in terms of code constructs, and by navigating the domain description, expectations are established for what should be found in the source code. Likewise, the process of code analysis will confirm or deny that certain expectations are satisfied.

As the code analysis process proceeds, decisions are detected that relate to an expectation. The annotation for the detected decision is recorded together with the relevant expectation, specifying the type of the decision and the corresponding sections of the program (in the repository labelled *Mapping/annotations*). During the process, certain expectations will be confirmed and others may be refuted. A confirmed expectation may lead to others. Gradually, a hierarchical description of the structure of the program emerges. As the program source code shrinks, the domain description expands.

Synchronized Refinement is a labor-intensive process. As such, its value is dependent not only on its accuracy but also on the degree of precision and thoroughness required of an analysis. Ultimately, of course, the value of SR also depends on the extent to which its output representation proves useful within the larger context of system evolution.

1.4 Example Usage

Section 2 of this guidebook describes the use of Synchronized Refinement in support of the evolution of Mosaic internet web browser. In this scenario, it is assumed that an evolution requirement has arisen to add *mailto* URLs to those types understood by the browser. A MailTo URL, instead of immediately switching the browser to another HTML page, presents a window into which the user can type a mail message to be delivered to a party specified in the URL. The question to be addressed is “What parts of the Mosaic source code should be altered to accommodate this change?”

2. Tutorial Example

3. Detailed Methodology Description

3.1 Input formats

The Synchronized Refinement process takes its input from four sources: the source code, an application domain description, and programming knowledge in the form of an architectural style library and a design decisions library.

Program source code: The program to be analyzed is written in some programming language. It is assumed by SR that the analyst has a working knowledge of the language and that the program successfully compiles. Actually correct parsing and static semantics are assumed. In most cases, SR is not concerned with the vagaries of compiling to specific platforms.

Domain description: TAn SR application domain description takes the form of an object-oriented (OO) framework [johnson]. This comprises a set of cooperating abstract classes as well as the protocols by which they communicate. A class corresponds to a domain concept and its associated *features*. A feature is either a public method that objects of the class can provide or a component of the object’s state (an instance variable). The specific OO notation is not of concern to SR because the details of method implementation are not present. Instead, a specification of functionality is required expressed in a model-based specification notation such as Z [] or VDM [].

The classes in an application domains collaborate to accomplish the application’s goals. The collaboration can be thought of in terms of associations among the classes in the framework or as protocols for interobject messaging by which the classes cooperate to accomplish their ends. Associations are notated using both standard constraints (i.e. cardinality constraints) and invariants specified in the same notation used to indicate method functionality. In the case of protocols, path notation is used []. This is a variant of regular expressions denoting allowable sequences of method invocations among the constituent classes. The protocol together with the component classes can be thought of as providing an architecture for solutions to problems residing in the application domain.

Architecture style library: The SR style library is a mapping between a style name and a domain description. The library is populated with commonly occurring styles such as *client-server* and *pipe-and-filter.* The library serves as a repository for generating expectations and quickly elaborating a domain description once an expectation is ratified.

Design decision library: Programmer elaborate a design into a program by using programming language constructs to implement various kinds of *design decisions*. For example, the programmer may decide that all parsing of URLs should be done in a central place in the code rather than in components specific to a particular URL types. Reverse engineers using SR try to detect instances of such decisions by applying both their knowledge of the application domain and their experience in programming. The design decision library is a repository of such knowledge. Specifically, the repository is organized as a hierarchical classification of design decision prototypes. Prototypes are described more completely in Section 3.3 below.

3.2 Output formats

Output from SR consists of three artifacts: an **application description**, an **abstract program description**, and a mapping between the two (**mapping/annotations**). In addition, it may be the case that the **design decision library** or the **architectural style library** are updated if examples of missing generic programming knowledge are detected in the code.

Abstract program description: As SR proceeds, examples of particular categories of design decisions are detected in the source code. Once such an example is recognized and confirmed, it can be replaced in the code by an abstract description of its role. In the parsing example mentioned above, the *parse mailto* function can be abstracted to the generic *parse* method, possibly parameterized by any mail-specific information. This results in two versions of the program: the original and the (shorter) abstracted version. Two things should be noted about this representation. First, the detected example may not occur contiguously in the program text. This implies that what is actually extracted from the original version may not correspond to a single syntactic construct. SR refers to these constructs as *plans*, and, for the purposes of this example, can be thought of merely as a collection (set) of related syntactic constructs. The second thing to note is that SR does not require that both the original and abstracted version of the code be retained for all detected decision as long as such versions can be reconstructed if required.

Application description: As understanding of the code grows, it needs to be represented so that it can be used for its intended purpose such as reengineering, reuse, or documentation. The form that SR uses for this is to instantiate the OO classes used for the **domain description**. For example, if the net browser application domain contains a class for URL parsing, and if the code analysis process actually detects the code that accomplishes this, then the abstract parsing class can be instantiated with the identified code.

Mapping/annotations: The artifacts described in the previous two sections must be synchronized. That is, it must both be possible for an analyst to locate the specific code that implements a domain concept and to determine for any code construct what role it fills in the application domain. This two-way mapping is provided in the form of a set of annotations. An annotation consists of three pieces of information. The first is the type of design decision that the reverse engineer detected. This corresponds to an actual entry in the design decision library. If no such site exists, then the library needs to be updated appropriately. The second piece of information indicates the specific place in the source code at which the design was implemented. This corresponds to the set of syntactic constructs described above. The third piece of information is the place(s) in the OO framework where the code segment has been instantiated. Together the three pieces of information allow the reconstruction of the understanding process and provide the raw material for a permanent repository (the Evolutionary Design Journal).

3.3 Steps

The two key concepts underlying Synchronized Refinement are abstraction-based design decision detection in source code and domain-driven application model synthesis. The two ideas are manifested simultaneously (*synchronously*) and an annotated representation is generated linking the results. The source code analysis process proceeds by detecting program constructs and abstracting them from the code, thereby compressing it. The application description synthesis process explores the input domain model in order to generate target concepts for detection in the code and to build up the application description. In this way, the source code continually grows shorter while the application description grows more complete.

Design Decisions: Software design is the process of expressing structure, functionality, and behavior using the constructs of a particular programming language. If we want to effectively evolve software then it makes sense to reuse as much of the original version as possible. This means reusing not only the code, but also the architecture and other design choices. To do so, it is important to take advantage of as much of the reasoning that went into designing the existing version in deciding how to proceed with the new version. This reasoning is collectively called *design rationale*, and Synchronized Refinement structures the rationale that it detects based on the abstraction mechanisms used in the program's source code. The abstraction mechanisms are implementations of *design decisions* made by the original programmer or designer. For example, choosing a dynamic data structure versus one with fixed bounds reflects the decision that an original requirement to allow for an "unlimited" number of items rather than arbitrarily constraining the implementation to a fixed length is an example of an explicit decision implemented by using the programming language's dynamic memory allocation features rather than statically allocated a fixed-length structure.

Design decisions can be divided into several classes based on the type of abstraction they provide. For example, many languages have a feature for bundling together or packaging a group of named constructs, to better control naming conflicts and enforce information hiding. This is an example of the *encapsulation* abstraction. Among the classes that are useful to detect during the process of reverse engineering are the following.

- **Composition/decomposition:** Programs are built up from parts, and problems are broken down into smaller, more easily solvable sub-problems. This type of decision is manifested in the programming language code by such constructs as modules and data structures.
- **Encapsulation/interleaving:** Subcomponents interact with each other. If the interactions are limited and occur through explicit interfaces, the component is said to be encapsulated. If, usually for reasons of efficiency, two or more

plans are realized in the same section of code or by the same data structure, then the components corresponding to those plans are said to be *interleaved*.

- **Generalization/specialization:** Often one component is similar to another. It may then be possible to construct a more general and higher level, parameterized component capable of realizing both as special cases. In object-oriented programming, the process is sometimes reversed, with the more general component constructed first, and the special cases added later.
- **Representation**[1]: In translating from the problem domain to the solution domain, decisions are made that result in a program component serving as a model for some application domain entity. If efficiency is a concern, high level programming constructs can be further represented by other constructs closer to the machine, such as using an array to represent a stack. Languages such as Ada are emerging that support explicit representation, but the reverse engineering of programs from older languages requires the detection of these decisions.
- **Data/Procedure:** Programs are sequences of computations organized by control structures. Variables are ways of saving intermediate results for later use, either to avoid recomputation or to simplify the expression of the computation. The introduction of a variable is an important design decision that is, unfortunately, too easy to make without appropriate thought and annotation.
- **Modality:** In some situations, a designer has a choice of how to express the relationship between input and output parameters. This is particularly true in logic programming languages, such as Prolog. For example, imagine the relationship between two arrays, both of which contain the same elements, one of which is ordered. If the ordered version is offered as input and unordered versions are output on subsequent calls, then the program is a permutation generator. In the situation where the unordered array is offered as input and the ordered version is output, then the program is a sorter. A single relationship, expressing a high level specification, can lead to alternative functions depending on the modes (input/output designations) of the parameters. This decision is usually made at a very early stage of design, if it is made explicitly at all.

There is no strict order for detecting design decisions. Complex programs are typically designed as layers of abstraction, and, as one layer is detected during the reverse engineering process, it opens the door for the detection of other decisions in higher layers. Nevertheless, it is possible to give the following guidelines for the detection process.

- Particularly where older programming languages are used, begin by looking for the occurrence of modern control structures implemented by more primitive constructs. An example of this is the use of nested **IF** statements and **GOTO**s to implement a **SWITCH** statement.
- Likewise, look for the use of primitive data structures to represent unavailable ones, such as the use of specific integer or character constants to encode one of a set of values.
- It may sometimes be necessary to restructure the control flow of the program. This is essentially what a restructuring tool provides. The resulting code should be functionally equivalent to the original.
- Look for special cases; that is, look for similar sections of code that differ only in a small number of ways. Replace these by the parameterized use of a more abstract construct.
- If the language does not support modularization, look for code that should be grouped together and separated from the rest of the program by an abstract interface.
- In contrast, sometimes a programmer intentionally interleaves the accomplishment of two objectives within the same section of code. This is often done for reasons of efficiency. To understand the code, it becomes necessary to segregate the two functions and annotate them separately.
- If a collection of data items are used together to implement an unavailable construct, annotate that fact.
- Make sure that variables are only used for one purpose. Replace dual-use variables by two separate variables.

Design Decision Repository:

Domain analysis: TBS

3.4 Evaluation

4. Glossary

1. Design decision: An explicit refinement of a program design. Each decision reflects the elaboration of a design or pro-

1. This use of the term *representation* should not be confused with its use as a notation for capturing a high-level understanding of a program.

gramming plan into a more concrete realization.

2. Domain: A problem area. Typically, many application programs exist to solve the problems in a single domain. The following prerequisites indicate the presence of a domain: the existence of comprehensive relationships among objects in the domain, a community interested in solutions to the problems in the domain, a recognition that software solutions are appropriate to the problems in the domain, and a store of knowledge or collected wisdom to address the problems in the domain. Once recognized, a domain can be characterized by its vocabulary, common assumptions, architectural approach, and literature. [Arango Pri*'eto-Diaz 1991]
3. Domain model: "The domain model should serve as a unified, definitive source of reference when ambiguities arise in the analysis of problems or later during the implementation of reusable components, a repository of the shared knowledge for teaching and communications, and a specification to the implementer of reusable components." "A model of a domain should include information on at least three aspects of a problem domain: concepts to enable the specification of systems in the domain; plans describing how to map specifications into code; and rationales for the specification concepts, their relations, and their relation to the implementation plans." [Arango]
4. Delocalization: The phenomenon where "programming plans [are] realized by lines scattered in different parts of the program." [Letovsky86]
5. Object oriented Framework: [Johnson]
6. Interleaving: "The merging of two or more distinct plans within some contiguous textual area of a program." [icsm95]
7. Program plan: A description or representation of a computational structure that the designers have proposed as a way of achieving some purpose or goal in a program. [Letovsky86,PA]
8. Rationale: A record of the reasoning used to produce a software artifact from a design specification.
9. Software evolution: A continuous change from a lesser, simpler, or worse state to a higher or better state [for a software system]. [Arthur]
10. Software understanding: Program comprehension - "the process of acquiring knowledge about a computer program." [encyc]
11. Reverse engineering: "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." [Chikofsky and Cross]
12. Synchronized Refinement: a method for reverse engineering software system comprising both bottom-up program analysis and top-down application domain synthesis.

5. Annotated References to the Literature

How this method relates to other methods.

Discriminators

Links to further information

[24] Arango, Guillermo and Prieto-Diaz, Ruben. Domain Analysis Concepts and Research Directions, in Domain Analysis and Software Systems Modeling, ed. Ruben Prieto-Diaz and Guillermo Arango, IEEE Computer Society Press, 1991.

[25] Arango, Guillermo. Domain Analysis Methods. In Software Reusability. (Eds.) W. Schaeffer, R. PrietoDiaz, and M. Matsumoto. Ellis Horwood, New York, 1993, pp. 17-49.

[26] Automated Data Systems Manual, Installation Material Condition Status Reporting System (IMCSRS), Functional Users Manual, Commander FORSCOM, AFLG-RO, Ft. McPherson, Georgia, April 1, 1984.

[27] Campbell R. H. The Specification of Process Synchronization by Path-Expressions. In Lecture Notes in Computer Science, pages 89-102, 1974.

[28] Campbell R. H. and Islam, N. A Technique for Documenting the Framework of an Object-Oriented System. Technical report UIUCDCS, University of Illinois at Urbana-Champain.

[29] DeBaud, Jean-Marc. From Domain Analysis to Object-Oriented Frameworks, A Reuse Oriented Software Engineering Methodology. Thesis Proposal, CIMR TR# 94-04, Georgia Institute of Technology, January 1994.

[30] DeBaud, Jean-Marc, Moopen, Bijith, and Rugaber, Spencer. Domain Analysis and Reverse Engineering, Proceedings of the Conference on Software Maintenance, pp. 326-335, Victoria, British Columbia, September 1994.

[31] Devambu, Prem. GENOA/GENII A customizable, language and front-end independent code analyzer, Fourteenth International Conference on Software Engineering, Melbourne, Australia, 1992.

[32] Devambu, Prem and Frakes, Bill, Extracting Formal Domain Models from Existing Code for Generative Reuse. Unpublished Technical Report.

[33] Frakes, W.B., C.J. Fox, B.A. Nejmeh. Software Engineering in the UNIX/C Environment, Prentice-Hall, 1991.

[34] Hildreth, Holly. Reverse Engineering Requirements for Process-Control Software, Proceedings of the Conference on Software Maintenance, pp. 316-325, Victoria, British Columbia, September 1994.

[35] Johnson, Ralph E. and Foote, Brian. Designing Reusable Classes. Journal of Object-Oriented Programming, June/July 1988, Volume 1, Number 2, pp 2235.

[36] Katz, S., Richter, C.A., The, K. PARIS: A system for reusing partially interpreted schemas. In 9th International Conference on Software Engineering (Monterey, Calif. Mar. 1987). IEEE Computer Society Press, Los Alamitos, Calif., pp. 377-385.

[37] Lubars, M. Domain analysis and domain engineering in IDeA. In Domain Analysis and Software System Modeling. IEEE Computer Society Press, Los Alamitos, Calif. 1991. pp 163-178.

[38] Neighbors, James. Software Construction from Components, PhD thesis, TR-160, ICS Department, University of California at Irvine, 1980.

[39] Neighbors, James. DRACO: A Method for Engineering Reusable Software Systems. 1989 ACM, Inc. Addison-Wesley Publishing Co., Reading MA.

[40] Peterson, S., Kang, K., Cohen, S., and Hess, J. Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA 15213, Nov. 1990

[41] Reasoning Systems, Inc., Palo Alto, CA. REFINE Users Guide, 1990. For REFINE (TM) version 3.0

[42] Simos, M. The growing of an Organon: A hybrid knowledge-based technology and methodology for software reuse. In Domain Analysis and Software System Modeling. IEEE Computer Society Press, Los Alamitos, Calif. 1991. pp 204-221.

MORALE METHODOLOGY GUIDEBOOK
Functional Modeling and Model-Based Architectural Redesign of Software Agents

Ashok K. Goel
J. William Murdock
College of Computing
Georgia Institute of Technology
{goel,murdock}@cc.gatech.edu

6. Introduction

Building a computational theory of evolutionary software design involves addressing three questions pertaining to adaptation: *why*, *what* and *how*. The first question asks why might the design of a software artifact be adapted? Potential answers include failure of the artifact to meet its task requirements and evolution of its task requirements. The second question asks what in the artifact might be adapted? Potential answers include modifications to the artifact's information-processing architecture and to its domain knowledge. The third question asks how might the needed modifications be identified? Potential answers include use of models of the artifact's architecture and knowledge, specifications of its design rationale, and descriptions of its processing in specific scenarios. In this article, we are interested in these three issues for a class of software artifacts characterized by sequential and centralized processing. That is, the class of software artifacts of interest here does not cover parallel processing or distributed systems. The goal is to develop a computing environment for supporting (human) software engineers in interactive redesign of this class of software artifacts.

Building a computing environment for interactive evolutionary design of software artifacts raises an additional issue pertaining to autonomy. At one extreme, the software engineer may make all redesign decisions and realize the modified design by reprogramming the artifact. At this end, the computing environment may provide access to relevant design knowledge, e.g., the design rationale, software architecture, and program code, but leave all decision making to the software engineer. At the other end, the artifact may be self-adaptive, autonomously redesigning its own architecture and knowledge, and reprogramming its own code. At this extreme, the software engineer may monitor the processes of self-redesign and self-programming, and intervene if and when needed. In this article, we are interested in the middle of this autonomy spectrum, where the artifact makes the redesign decisions but leaves the reprogramming to the software engineer. Thus the artifact is an *intelligent software agent* that is capable of making decisions about its own redesign.

We describe a methodology called MESA (for Model-based Evolution in Software Agents) that seeks to address the above class of problems in evolutionary software design. The MESA methodology has two parts: (1) modeling of the information-processing architecture and domain knowledge of a software agent, and (2) use of the model for architectural and knowledge redesign. In this article, we focus on the modeling of the architecture and knowledge of the agent, and the use of the model for architectural redesign. We describe a modeling method called SBF (for Structure-Behavior-Function) that views a software agent as a telelogical artifact, i.e., as an artifact that has particular functions and specific behaviors that arise from the structure of the artifact and result in the accomplishment of its functions. The SBF modeling method leads to a Task-Method-Knowledge (or TMK) model of the software agent, which specifies both the functional architecture of agent in terms of its task-method structure, and its domain knowledge. We also outline the use of the TMK model for an example pertaining to MOSAIC.

7. Structure-Behavior-Function Modeling

The SBF method of modeling arises from our earlier work on conceptual phase of functional design of physical devices [Goel 1991a, 1991b, Goel and Chandrasekaran 1989, 1992]. The SBF model of a device specifies how the device works, i.e., how its structure delivers its functions. In particular, the SBF model explicitly represents not only the structure and functions of the device, but also its internal behaviors that explain how the functions of the structural components of the system get composed into the functions of configurations of components, and how the functions of the component configurations get recursively composed into the functions of the device as a whole. Thus the SBF model is organized in a hierarchy of the form *function -> behavior -> function -> behavior* ..., up to any desired level of component aggregation and abstraction. The functions at any level in this hierarchy act as indices into behaviors that achieve them, and the behaviors act as indices into functions of component configurations at the next lower level.

The SBF language provides a general vocabulary for representing the structure, behavior and functions of physical devices. The representation of a structure specifies its topology in terms of components, substances that can flow among the components, and relations among the components and substances. The representation of a function of a structure specifies the behavioral state the structure takes as input and the behavioral state it gives as output, where a state is specified in terms of properties of components and substances. The representation of a behavior specifies a sequence of behavioral states and state transitions that results in the function of the structure, where the transitions are specified in terms of functional, causal, and structural relations of the substructures.

In case of intelligent software agents, the SBF modeling method results in a TMK model of the agent's functional architecture and domain knowledge. Figure 1 illustrates the organization of the TMK model of the portion of Mosaic responsible for external viewing of Postscript files. The model describes the architecture of the system in terms of its task-method structure and domain knowledge. Tasks (e.g., Display-Interpreted-File) constitute the building blocks of the system's architecture. Methods (e.g., External-Display) decompose a task into subtasks, which, in turn, are recursively decomposed into smaller and simpler subtasks. The leaf tasks (e.g., Access-Display-Program) are directly accomplished by "procedures" that use domain knowledge and get encoded in the program. A task is characterized as a transformation from an input information-state to an output information-state. It is specified by the type(s) of information it consumes as input and produces as output, and specific relations that hold true among the task's inputs and outputs. In Mosaic, a task such as Display-Interpreted-File actually is a family of tasks, whose members differ from each other only in specifics of the information they take as input and give as output. but not in the relations between the input and output information. A method is characterized as a (partially ordered) sequence of information-states and information-state transitions that compose tasks at one level into a task at the next higher level in the task structure. It specifies the data and control dependencies between the tasks at the lower level. Again, in Mosaic, a method such as External-Display really is a family of methods, whose members differ from each other only in specifics of the information in the state transitions, but not in the data/control dependencies among the lower-level tasks. As Figure 1 indicates, S(tructure) in SBF corresponds to the leaf tasks, the B(ehavior) to the methods, and the F(unction) to the tasks in the task structure.

8. The Task-Method-Knowledge Language

Autognostic [Stroulia and Goel 1994a, 1994b, 1995, 1996] is a semi-formal language for specifying TMK models of intelligent software agents. Its TMK language enables a specification of the tasks and subtasks of the agent, the methods used by the agent to accomplish a task, and the domain knowledge used by the methods. The tasks specify the input and output information states. The methods specify the subtasks of a task and the flow of data and control between the subtasks. The domain knowledge is specified in terms of domain concepts, relations and constraints. Thus the TMK language provides primitives for specifying the functional semantics of tasks (including the leaf tasks), the compositional semantics of methods, and also the "causal" (data/control) semantics of the interdependencies among tasks, methods, and domain knowledge. The following specifications of the TMK language are taken due to Stroulia [1994].

8.1 The Task

Each task is described as a tuple $T := (in,out,proto,by,proc,cond,sem)$, [Footnote: In addition to these slots, there are two additional slots which we have considered *subtask_of* and *instances* which are important to building a trace of a model but are not exactly parts of the model itself.] where *in* is the input of the task, *out* is its output, *proto* is another task of which the current task is an instance, *by* is a set of methods which can be applied to accomplish the task, *proc* is a procedure [Footnote: The term *procedure* here suggests that they are performing an internal manipulation of knowledge. Alternatively, the can be viewed as *operators*, i.e. as primitive actions which have a direct impact on some external world state.] which can be applied to accomplish the task, *cond* is a set of conditions under which the task should be performed, and *sem* is a set of semantic relations partially describing the role of the task in the task structure.

The slots *in* and *out* are information states. For some specific task tk, the information state $in(tk)$ consists of the types of information which are necessary for the task tk to be accomplished. Thus, for each type of information, $i \in in(tk)$, i is either required by some of the subtasks resulting from the decomposition of tk, by some method $m \in by(tk)$, or it is an input parameter of the procedure which implements $proc(tk)$. For the same task tk, the information state $out(tk)$ consists of the types of information which are produced by the task tk, and which are going to be used by other subsequent tasks.

The slots $cond(tk)$ and $sem(tk)$ are sets of relations. The relations in $cond(tk)$ describe the conditions under which the task tk contributes to the progress of reasoning. They are used as a criterion to decide whether to perform the task or not; thus they refer to the types of information available in $in(tk)$. The relations in $sem(tk)$ partially describe the information transformation that tk performs in the service of the overall reasoning. Essentially, the describe the function intended of the task in the overall context of reasoning. Thus each relation $sem(tk)$ is either a unary relation on some type of information in *out*(tk) or it relates to some type of information in *out*(tk) with some type of information in *in*(tk).

The *by*(tk) slot is a set of methods which are applicable to the task tk. When one of these methods is applied to tk, it sets up a set of subtasks such that if each subtask is accomplished, the overall information transformation intended by tk will be accomplished. The attribute *by* provides a partonomic hierarchy among tasks.

Finally, the *proto*(tk) slot is another task, tp, of which tk is an instance. It is simply a task which accomplishes an information transformation more general than or equivalent to the transformation accomplished by tk. If the task tk does not have associated with it a method or a procedure, i.e. $by(tk) = \emptyset \wedge proc(tk) = \emptyset$, then it can be accomplished with the methods or the procedures associated with $tp = proto(tk)$. The attribute *proto* provides a taxonomic hierarchy among tasks.

In Autognostic the SBF language is implemented in a schema-based framework. Table 1 depicts the schema for specifying a task.

8.2 The Method

Each method is described as a tuple $M := (app,cond,sub,ctrl)$, where *app* is the task to which M is applied, i.e., the task which M is intended to accomplish, *cond* is a set of conditions under which the method is applicable, *sub* is the set of subtasks that the method sets up for *app*, and *ctrl* is a set of control relations over these subtasks.

For any specific method, mk, the *cond*(mk) slot is a set of relations which describe the conditions under which the task mk contributes to the progress of the reasoning. They are used as a criterion to decide whether to perform the task or not; thus they refer to the types of information available in the information state *in*(tk).

When a method, mk, is applied to the task *app*(mk) it decomposes it into a set of simpler subtasks *sub*(mk). The order in which these subtasks must be accomplished is not completely specified by the method. Instead, the method, mk imposes only a partial ordering among them which is specified in the *ctrl* slot of the method schema. In the SBF language, there are three control operators *serially*, *in_parallel*, and *loop_until*. The control that the method imposes over the subtasks it sets up, *ctrl*(mk), is specified as a control operator imposed over a set of subtasks in the set *sub*(mk) or other control statements. When two subtasks are connected by the *serially* operator, they must be performed in the order they are mentioned. When they are connected by the *in_parallel* operator, they may be performed in any ordering. Finally, when they are connected by the *loop_until* operator, then they must be performed repeatedly until the condition annotating the *loop_until* operator is met.

Table 2 depicts the schema for specifying a method,

8.3 The Type of Information

The input and output of each task in the problem-solver's task structure consists of a set of types of information. A type of information is defined as a tuple $I := (obj,in,out)$, where *obj* is the type of domain concept of which this information type is an instance, *in* is the set of tasks which take the type of information as an input, *out* is the set of tasks which produce the type of information as an output. Note that for a given information type, i, it is required that $\forall\ tk \in in(i).\ i \in in(tk)$ and $\forall\ tk \in out(i).\ i \in out(tk)$.

Table 3 depicts the schema for specifying a type of information reasoned about by the problem-solver's task structure.

8.4 The Domain Concept

Domain concepts are the concepts which are relevant to reasoning in a particular domain. They may correspond to physical entries in the real world or they may be abstract. A domain concept is described as a tuple $DConc := (dom,attr,id)$ where *dom* is the domain in which this concept may take values, *attr* is the set of attributes characteristic of the concept, and *id* is a predicate that evaluates the identity between two instances of a concept.

The actual knowledge of the problem-solver about domain objects of this concept type is in the form of a set of instances that the problem-solver knows about. This set of instances constitutes the domain, *dom* of this object type.

Additionally, for each attribute in *attr*, the SBF language specifies its *name*, its *type*, and a function f which given a specific instance of the concept produces the value of the attribute for the given instance.

Table 4 depicts the schema for specifying a type of domain concept.

8.5 The Domain Relation

A domain relation is specified by a tuple $DRel := (in,out,truth,pred,invpred,idx)$. The elements *in* and *out* are the types of arguments that the relation takes as input and produces as output. The elements *truth*, *pred*, and *invpred* refer to the actual knowledge of the problem-solver regarding this domain relation. For a given domain relation rk, this knowledge may have the form of a truth table, *truth*(rk) which includes all of the pairs $(ij,oj) \in in(rk) \times out(rk)$ for which the relation is true. Alternative, there may be a predicate, *pred*(rk), which, when applied to any instance $ij \in$ *in*(rk), returns the one unique $oj \in$ *out*(rk) for which the relation is true and *nil* if none exists. In the latter case, the problem solver may also know of an inverse predicate *invpred*(rk) which when applied to any instance $oj \in$ *out*(rk), returns the one unique $ij \in$ *in*(rk) for which the relation is true and *nil* otherwise. A key difference between these two ways of evaluating rk is that in the first case the contents of rk are inspectable; the problem-solver may modify rk by simply adding or deleting tuples in *truth*(rk). In the latter case, since the predicates *pred*(rk) and *invpred*(rk) are non-inspectable "definitions", the problem solver cannot modify them.

Finally, the attribute *idx*(rk) characterizes the type of domain relation. The concepts in the domain of the problem-solver may be related through domain relations. Domain relations can be of two types: relations whose truth value depends on the state of the world, which we call *state-of-the-world* relations, and organizational relations which are "invented" to increase the efficiency of problem solving, which we call *convention* relations. A problem-solver may often create associations between different types of knowledge, which are all relevant at the same point in its problem solving, in order to be able to retrieve the appropriate pieces of information more easily. An important difference between these two kinds of relations is that the evidence necessary for the problem-solver to modify the convention relations is less than the evidence necessary to modify the state-of-the-world relations. In fact, the problem-solver may begin without any specific organization of its knowledge, i.e. with an undefined domain relation, and it can induce this relation by adding entries into its table on an as needed basis, i.e. every time it fails because of its incomplete organization.

Table 5 depicts the schema for specifying a domain relation.

8.6 The Domain Constraint

Constraints are higher-order relations that apply to the objects of a domain. A domain constraint is specified as a tuple *DCons* := (*if*,*then*,*where*) where *if* and *then* are domain relations and *where* is a predicate which connects the attributes of those two. If in some particular domain, there is a constraint ($r1(A,B)$,$r2(B,A)$) where $r1$ and $r2$ are relations in that domain and A and B are types of objects in that domain, then this constraint means that if for a particular tuple (a,b) the relation $r1$ is true, then the relation $r2$ is true for (b,a). If the types of attributes of $r1$ and $r2$ are not the same, it is still possible to specify a constraint between these two relations if there is a predicate *where* such that it applies tot he attributes of relation $r1$ and produces the attributes of $r2$ which do not belong in the set of attributes of $r1$.

Table 6 depicts the schema for specifying a domain constraint.

9. Model-Based Software Evolution

A common problem in evolutionary design of intelligent software agents is incremental addition of new functionalities desired of an agent. In this class of problems, a software agent delivers a set F(old) of functions f(1),...,f(n-1), but subsequently the software engineer (or a user) specifies a new set, F(new), containing an additional function, f(n). The specific problem of adding the functionality of supporting user-specified external viewers in Mosaic 2.4 is an instance of this class of problems. Often the additional function, f(n) in F(new), is similar and related to some function f(i) in F(old). For example, while Mosaic enables external viewing of Postscript files using Ghostview, a user may specify the additional functionality of viewing PDF files using the Acrobat Reader, where the new functionality is similar to the one already delivered. The design issue in this class of problems becomes the adaptation of the current structure of the software system, S(old), that delivered F(old), into a modified structure, S(new), for delivering F(new). To help maintain the global consistency of the behaviors of the software system, and to make the implementation of modifications easy, the adaptation strategy needs to follow the principle of minimal change, insuring that the modifications are as localized as possible.

The TMK model of the software agent enables multiple strategies for modifying S(old) to obtain S(new). Here we outline one strategy that we will call *within-problem-analogy*: (1) The function f(i) in F(old) that most closely matches the additional function f(n) in F(new) is identified. In the example of viewing PDF files, this may result in the identification of viewing Postscript files as the closest match. The representation of the functions in the SBF language enables this matching. (2) The components and the behaviors responsible for achievement of f(i) are localized and identified. In our running example, the components of Access-Display-Program, Execute-Display-Program, and Library-of-Display-Programs are identified, along with the behavior of External-Display that composes the functions of these components into the function Display-Interpreted-File for Postscript files. The functional, compositional and causal semantics of the SBF model enable this localization and identification. (3) The identified components and behaviors are abstracted into a Generic Teleological Mechanism (GTM) [Stroulia and Goel 1992; Bhatta and Goel 1993] by removing all structural information specific to display of Postscript files. For example, in the GTM, the components Access-Display-Program and Execute-Display-Program are specified only by their functions, and the functional specifications refers only to types of information they take as input and give as output, not to the specifics of information relevant to Postscript files. A GTM thus specifies a functional, compositional and causal design pattern. The SBF language provides the vocabulary for representing GTMs. (4) The abstracted GTM is instantiated in the context of the additional function f(n). In our example, the GTM for external viewers is instantiated in the context of viewing PDF files. This instantiation results in the placement of a new display program (the Acrobat Reader, specified as part of f(n)) in the Library-of-Display-Programs, and the introduction of a new method for viewing PDF files in the family of External-Display methods. The SBF model focuses the GTM instantiation. In addition, the original SBF model is now revised to reflect the modified structure for the software system. We call this strategy within-problem analogy because in solving a subproblem (e.g., external viewing of PDF files), it relies on making an analogy to the solution to another subproblem (e.g., external viewing of Postscript files).

10. Relationship to MORALE

The MESA methodology for software evolution is related to the MORALE project in three ways:

1. Architectural Description Language:TMK is a language for describing the functional architecture and domain knowledge of a software artifact. The TMK model of a software artifact is organized in a hierarchy of functional and compositional abstractions, and explicitly relates the behaviors of the artifact with the domain knowledge.
2. Model-Based Architectural Redesign:MESA provides some answers to the *why*, *what* and *how* questions we raised in the introduction. In particular, the TMK model of a software artifact explicitly represents what can be potentially modified, which ranges from the tasks, methods, domain knowledge, and flow of data and control. It also enables "how-to" strategies, such as within-problem analogy, that identify the needed modifications to the architecture of the software artifact.
3. Functional Modeling Method:The SBF method for functional modeling provides a set of functional, behavioral and compositional abstractions of software artifacts. These abstractions will help guide the process of reverse engineering of legacy systems.

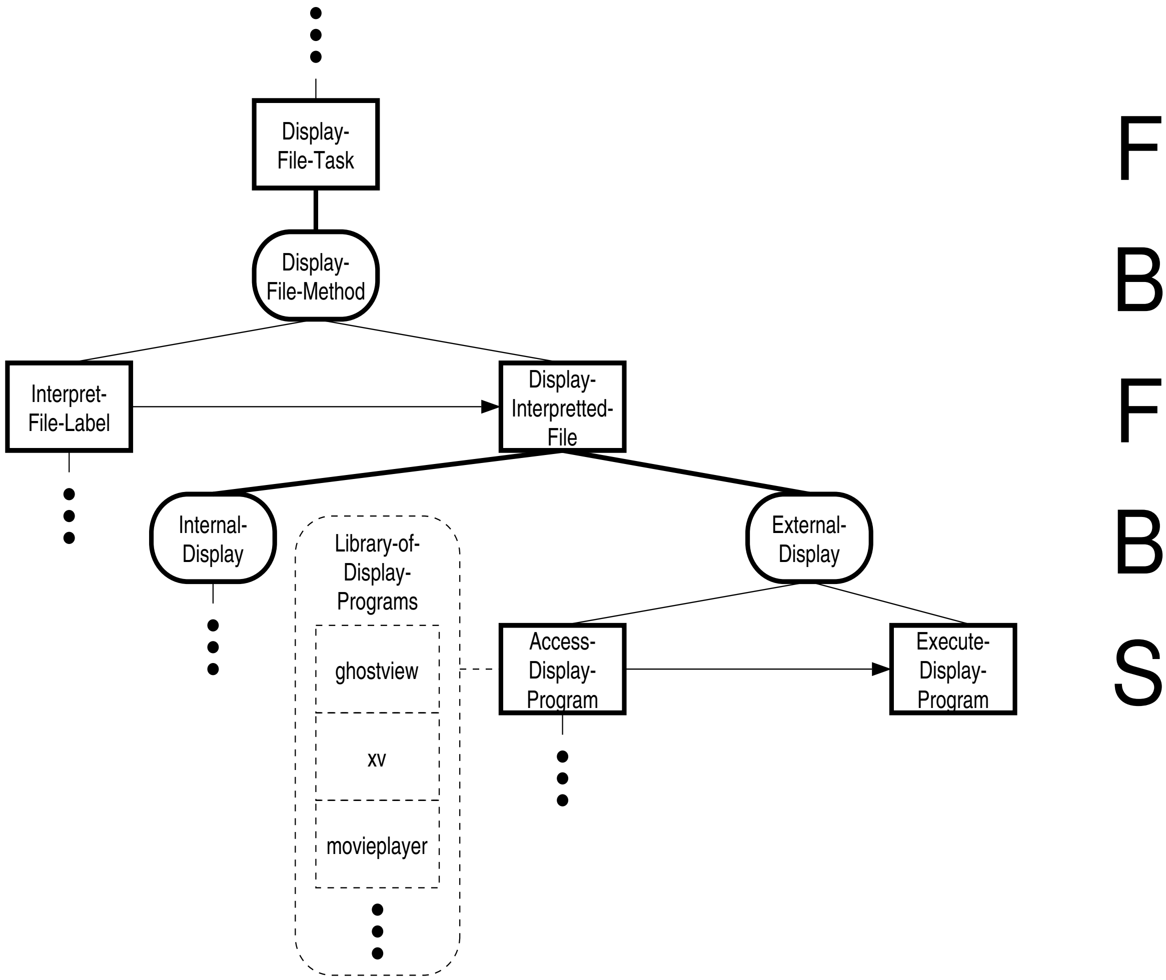


Figure 16 A partial SBF model of a Mosaic 2.4. Rectangular boxes indicate tasks while rounded boxes indicate methods. Knowledge sources are contained within dotted lines. Thick lines between boxes represent the availability of methods to accomplish a task; thin lines represent a method's composition of subtasks; Horizontal arrows indicate ordering of subtasks within a method.

task	the name of the task *t*
input	the types of information taken as input *in*(*t*)
output	the types of information produced as output *out*(*t*)
prototype	the prototype of which this is an instance *proto*(*t*)
by_methods	the set of methods possibly applicable *by*(*t*)
procedure	the procedure that directly accomplishes this *proc*(*t*)
under_condition	the conditions under which this is useful *cond*(*t*)
semantics	the information transformation performed *sem*(*t*)

Table 43 The Task Schema

method	the name of the method *m*
applied_to	the task for which this is applicable *spp*(*m*)
under_condition	the conditions under which this is useful *cond*(*m*)
subtasks	the set of subtasks that this sets up *sub*(*m*)
control	the flow of control among the subtasks *ctrl*(*m*)

Table 44 The Method Schema

information	the name of the information type *i*
input_to	the tasks that consume this as an input *in*(*i*)
produced_by	the tasks that produce this as an output *out*(*i*)
object_type	the type of domain concept of which this is an instance *obj*(*i*)

Table 45 The Type of Information Schema

concept	the name of the concept *c*
domain	the domain in which instances of this take values *dom*(*c*)
attributes	the set of attributes applicable to this *attr*(*c*)
identity_test	the predicate for identity between instances of this *id*(*c*)

Table 46 The Domain Concept Schema

relation	the name of the domain relation *r*
input_args	the types of concepts taken as input *in*(*r*)
output_args	the types of concepts produced as output *out*(*r*)
truth_table	the table enumerating the tuples are valid *truth*(*r*)
predicate	the predicate for deriving valid output from input *pred*(*r*)
inverse_predicate	the predicate for deriving valid input from output *pred*(*r*)
indexing_rel	whether or not this is used for indexing purposes only *idx*(*r*)

Table 47 The Domain Relation Schema

constraint	the name of the domain constraint *c*
if	one of the relations involved *if*(*c*)
if	the other relation involved *then*(*c*)
where	the predicate relating non-common attributes of the relations *where*(*c*)

Table 48 The Domain Constraint Schema

MORALE METHODOLOGY GUIDEBOOK
Model Oriented Reengineering Process for Human-Computer Interface (MORPH)

Melody M. Moore
College of Computing
Georgia Institute of Technology
melody@cc.gatech.edu

11. Introduction

MORPH is a technique for reengineering the user interfaces of text-based legacy systems to graphical user interfaces, and a toolset to support the process. This document is an overview of the MORPH reengineering process and a description of the MORPH toolset components and architecture.

11.1 Purpose

Information systems are critical to the operations of most businesses, and many of these systems have been maintained over an extended period of time, sometimes twenty years or more [JAY93]. With the advent of client-server technology, open systems, and high-powered graphical workstations, many businesses are choosing to reengineer their critical applications to better fit the needs of the organization and to take advantage of new technologies. In the information systems domain, this frequently entails reengineering applications that run on large mainframes, with character or form-based user interfaces, to environments that require graphical user interfaces.

It has been estimated that half or more of the code in an interactive system is devoted to implementing the user interface [SUT78]. User interface code tends to be very platform-dependent, often making it necessary to completely rewrite it when the system is migrated. If the user interface components can be extracted from the computational code and expressed in an abstract model, then maintenance and future migrations can be made much simpler, since only the user interface components would need to evolve. Forward engineering tools, such as a User Interface Management System could be employed to generate the new graphical user interface. Understanding the functional requirements of the user interface is an important part of reengineering, and adding automation to this process could substantially reduce the cost and effort for the migration, and allow large-scale systems to be migrated.

11.2 The overall MORALE architecture

(common across guidebooks)

11.3 Overview of method

MORPH is a process for reengineering the user interfaces of text-based legacy systems to graphical user interfaces, and a toolset to support the process. This section is an overview of the MORPH reengineering process. A description of the MORPH toolset components and architecture follows in a later section.

11.3.1 The MORPH Process

There are three steps in the user interface reengineering process [MOO94A]:

- *Detection* - Analyzing source code to identify user interface components from the legacy system.
- *Representation* - Building a model of the existing user interface from the detection step.
- *Transformation* - Manipulating, augmenting, or restructuring the resulting model to allow forward engineering to a graphical environment.

Figure 1 below describes the reengineering process:

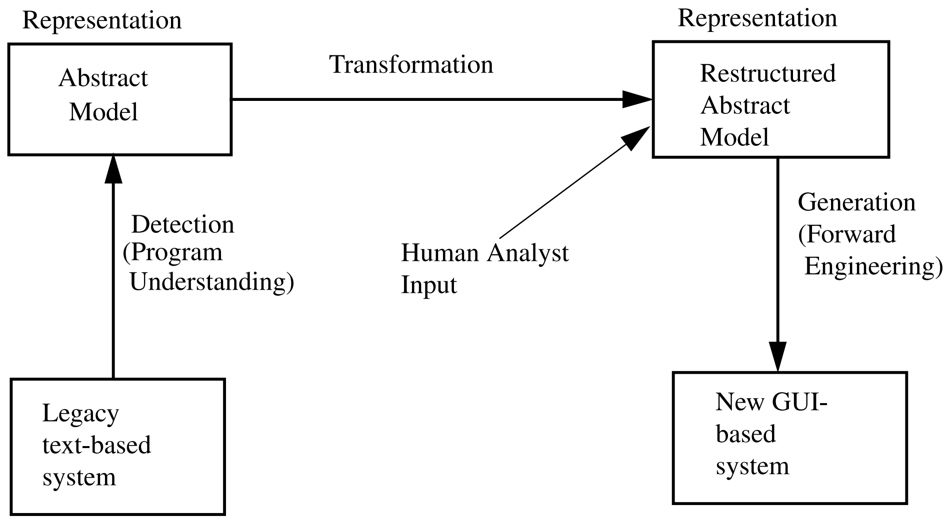


Figure 17 : A Reengineering Process for User Interfaces

The process begins with extracting the user interface from the computational legacy code, using program understanding techniques to build an abstraction, or model, of the existing interface. This path is depicted by the “detection” edge and the “abstract model” box in Figure 1. The legacy system model can then be transformed to a model structured to support a WIMP interface, shown by the “transformation” edge. Input from the human analyst is added at this stage, to define presentation details and to enhance the model. Once the model has been restructured, a forward-engineering tool, such as a UIMS, could be used to automatically generate a new graphical interface for the system.

Defining User Interface Abstractions

A key element of the MORPH process is the set of user interface abstractions that can be recognized from legacy code and represented in the user interface model. These abstractions start with the four basic user interaction tasks described in [FOL90]:

- Selection - the user makes a choice from a set or list of choices
- Quantify - the user enters numeric data
- Position - the user indicates a screen or world coordinate
- Text - the user enters text from an input device

Coding patterns that implement each of these basic interaction tasks can be used to detect these tasks in text-based user interfaces. Once the tasks are identified, then the code can be examined for further clues to possible attributes of the interaction object that will be most appropriate in the GUI. For example, if a selection task is identified, then the number of choices may determine whether that interaction object should map to a menu or a scrolling list. MORPH maintains a set of interaction objects abstractions, categorized under the basic interaction tasks. These abstractions are described in a knowledge base, to allow transformations between the abstractions and specific GUI toolkits to be accomplished by inferencing.

The Detection Step

The MORPH process starts with a static analysis step which applies a pattern recognizer to the application code in order to locate user interface implementation patterns. First, user interface variables are identified in the legacy code. Abstract syntaxt trees (ASTs) are generated from the legacy code, and coding patterns are then compared to the ASTs at definition and use points for the user interface variables, in order to detect interaction tasks. When an interaction task is detected, the patterns to determine attributes of that interaction task are applied.

The Representation Step

The recognized interaction task and its attributes are stored in the MORPH Knowledge Base as an interaction object, in the correct place in the abstraction hierarchy. Figure 2 below shows the organization of this conceptual hierarchy:

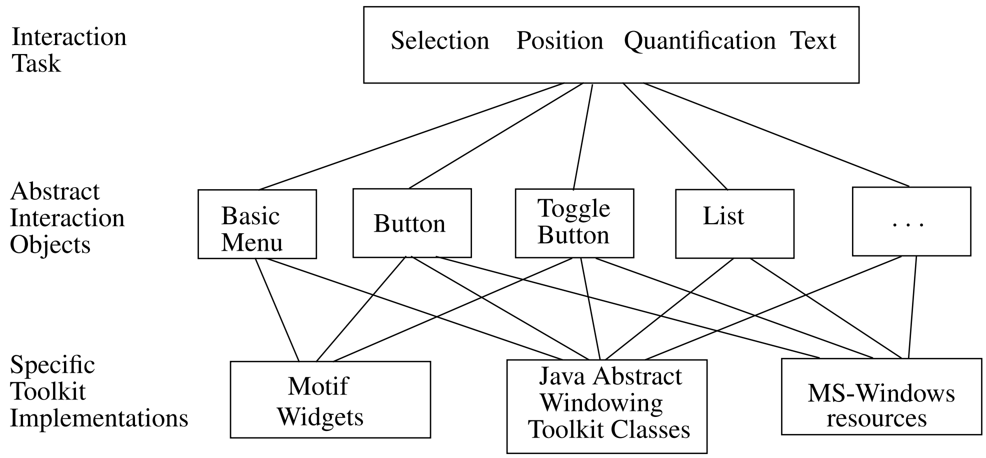


Figure 18 : MORPH Abstraction Hierarchy

The Transformation Step

Once the detection and representation steps are completed, the Knowledge Base can be employed to identify transformations from the abstract user interaction objects to specific toolkit widgets. If toolkit widgets are described in the Knowledge Base, inferencing can be used to find the best mapping from abstract object (by its attributes) to specific widget. When all of the abstract objects have been mapped to specific toolkit widgets, the new graphical user interface can be generated with an (existing) GUI builder tool.

11.4 Example Usage

(scenario)

12. Tutorial Example

13. Detailed Methodology Description

This section details the MORPH knowledge base, pattern set, and toolset.

13.1 Defining the User Interface Abstraction Knowledge Base

Interaction tasks define*what* the user accomplishes while interacting with a user interface - for example, selection, or entering data [FOL90]. *Interaction objects* are the controls (such as buttons, textboxes, or sliders) that define *how* the user interacts with the system [DEB92]. Interaction tasks are defined by the application requirements, and do not change when the system is migrated to a different platform. Interaction objects are defined by the implementation environment, and may be completely replaced during migration. Interaction objects can be classified by the interaction task they implement, differentiated by identifiable attributes. For example, a menu is an interaction object that implements the *selection* interaction task, with the attributes “small, static choice set”. A scrolling list is an interaction object that also implements the selection interaction task, but can be used to implement a “potentially large, dynamic choice set”.

Migrating an interactive system to a new environment requires three steps: program understanding, representation, and transformation. In the *program understanding* step, identifying interaction tasks from code enables construction of an abstract model of the user interface (expressed by a *representation*). From this abstract model, *transformations* can be applied to map the model onto a specific implementation. It has been shown that interaction tasks can be detected from character-oriented application code by defining a set of coding patterns and replacement rules [MOO96b]. This paper expands on these ideas to show that attributes of interaction tasks can also be detected from code, in order to construct a higher quality abstract model of the user interface. With more detail in the model, the transformation step can select more appropriate interaction objects for the implementation of the new user interface.

13.1.1 Declarative Models

In [DEB92], deBaar, Foley, and Mullet present a declarative approach to designing a user interface by specifying a data model, and then using an inference engine to generate the user interface from the data model. This method also has applicability to reverse engineering, since it is straightforward to detect a data model from code. Detecting interaction tasks and objects of a user interface from text-based code depends largely on the data structures involved in I/O. Identification of attributes of variables that are used in I/O with the terminal may be able to determine which components of a specific toolkit are appropriate to substitute in a GUI interface.

The addition of a knowledge base strengthens this approach. Frame-based (slot-and-filler) knowledge base systems are particularly suited for pattern recognition [FRO85]. If data structure and attribute patterns for interaction tasks and objects can be identified, then a frame-based knowledge representation could be used to describe them. This knowledge base could then be used to aid in the detection of user interface components from legacy code, and then used to identify the appropriate replacement interaction objects within a specific toolkit such as MS-Windows or Motif.

This section details attributes of interaction tasks that define semantic differences in the associated interaction objects, in order to build a knowledge base of user interface abstractions for character-oriented interfaces. The results have been formalized by coding in the CLASSIC [RES93] knowledge representation language.

13.1.2 Interaction Tasks

The basic interaction tasks outlined in [FOL90] are the foundation for organizing a taxonomy of abstract interface objects [MOO96a]. These high-level abstractions are not particular to any specific toolkit or widget set. The interaction styles described in [HIX93] are classified by the interaction task that they implement, and are included as abstract interaction objects in each task category. The implementations of interaction objects included in a particular toolkit vary widely; however, care has been taken to choose those whose semantics can be implemented in most toolkits. The four basic interaction tasks are:

- *Selection* - allows the user to choose from a set of choices.
- *Text* - provides data entry and output messages to the user.
- *Quantify* - allows the user to specify a numeric value.
- *Position* - allows the user to specify a position in screen or world coordinates.

13.1.3 Attribute Taxonomy

This section details each interaction task, and provides tables showing defining attributes of each interaction object that could be detected from text-based legacy code.

Selection

Selection, allowing the user to make a choice from a set of alternatives, is one of the most-used basic interaction tasks, especially in character-oriented applications. Selection choices can be commands (file save, edit) or setting attributes and states (turn on boldface type). Selection interaction objects range from buttons of various kinds to text entry and menus. Attributes that are important in determining the appropriate selection mechanism include:

- Action when selected - When a selection mechanism is invoked, it either causes an action to occur (procedural), or it changes a state in the user interface or in the application.
- Number of States - All selection objects have a range of states or choices, from one to infinite. This column describes the number of states available for that selection object.
- Choice List Type - whether the number of choices in the choice list is fixed, or whether it can be added to dynamically (variable).

The following decision table details selection semantics, but not presentation feedback. (In other words, if a pushbutton reverses color when the cursor is moved over it, this is not considered semantics but rather presentation feedback, and not salient in this analysis).

Table 49:

Abstract Interaction Object	Action When Selected	Number of States	Choice List Type
Pushbutton	procedural (action)	One	Fixed
Toggle Button	State Change	Two	Fixed
Basic Menu	procedural (action)	Few (< 15)	Fixed
Option Menu	State Change	Few (< 15)	Fixed
Selection List	State Change	Many - (> 15)	Variable/ Dynamic

The Grouping Concept

In the "toggle button" selection object, there is an emergent phenomenon which occurs when the objects are grouped: they become mutually exclusive. That is, selecting one toggle button deselects all others in the group, as with radio buttons. The behavior of one toggle button affects the behavior of, potentially, all others in the group. The toggle buttons may be grouped organizationally (putting them all in the same window) without these semantics; it is possible to have multiple selection of toggle buttons. However, in these circumstances they are considered to be individual buttons and the grouping is purely presentation information. Grouping toggle buttons to make them mutually exclusive affects their semantics, not just their presentation.

Text Data Entry and Output

Text entry allows the user to input character strings to the application. Text data that is validated by comparing against a list of values is considered a selection mechanism (as described in the "selection list" abstract object). Text data that is not validated, however, such as in entering a name or address that is passed directly to the application, is considered to be pure data. Text data that is validated by checking against a regular expression (checked for pattern correctness, such as in entering a date, which would be checked for a particular date format and legal values) is also considered text data entry. Text output is also included here, since messages from the system can be output read-only and need to be detected in legacy code. There are two attributes that are important in choosing a Text Data Object:

- Modifiability - indicates whether the field can be edited (input), or if it is read-only (output).
- Length - determines the number of characters that can be entered into the textfield or displayed by the textfield.
- Validation allowed - can there be validation of the input, such as error checking, or pattern matching?

Following is the attribute table for text data, which was adapted from [DEB92]:

Table 50:

Modifiability	Length	Validation Allowed	Interaction Object
Editable	Short (< 80 characters)	yes	Single Line Text Field
Editable	Long (> 80 characters)	no	Multiline Text Field
Read Only	Short (< 80 characters)	no	Read-only message

Table 50:

Modifiability	Length	Validation Allowed	Interaction Object
Read Only	Long (> 80 characters)	no	Read-only multiline message

Quantify

The Quantify interaction task allows the user to enter a numeric value in a prescribed range [FOL90]. In a text-based application, this is usually accomplished by the user typing in a number. In rare cases, quantify could be implemented by terminal control semantics by moving the cursor up and down a scale, similar to a numeric slider. Numeric input has the following attributes:

- Modifiability - Indicates whether the numeric field can be edited, or is it display only (read-only).
- Upper and Lower Bounds - Determines whether the minimum and maximum values for the numeric field are known or unknown.
- Range Size - Taking into account the upper and lower bounds, how many values are in the range of the numeric field?
- Precision - Whether an approximate relative value is required (low), or a precise value (high).

The following tables were adapted from [DEB92]:

Table 51:

Modifiability	Upper and Lower Bounds	Range Size	Precision	Interaction Object
Editable	unknown	large	don’t care	Textbox (numeric field)
Editable	known	small	low	Slider
Editable	known	small	high	Textbox (numeric field)
Editable	known	large	high	Slider with numeric field
Read-only	unknown	don’t care	don’t care	Read-only message
Read-only	known	don’t care	low	Read-only Gauge
Read-only	known	don’t care	high	Read-only message

Similarly, we can define attributes for numeric values with real data types:

Table 52:

Modifiability	Upper and Lower Bounds	Range Size	Precision	Interaction Object
Editable	unknown	any	any	Textbox (numeric field)
Editable	known	small	low	Slider
Editable	known	small	high	Textbox (numeric field)
Editable	known	large	any	Textbox (numeric field)
Read-only	any	any	any	Read-only textfield

Position

The Position interaction task allows the user to specify a coordinate position (x,y) on the screen for interpretation by the application program [FOL90]. In a GUI, this is typically accomplished by moving the mouse to the desired screen location and clicking, or otherwise moving a cursor and using a selection mechanism (such as a pushbutton) to indicate the choice has been made. Another method that can be employed in both GUI and text-based interfaces is typing the screen coordinates in a textfield. Text-based systems can also use terminal control semantics to move the cursor to a particular area of the screen - although care must be taken to ensure that selection is not mistaken for position. Position tasks in text-based systems are always discrete, never continuous, because of the nature of the keyboard input device. Attributes for the Position interaction task are:

- Resolution - how important the actual specified values are to the application.
- Coordinate system - which values need to be specified for the coordinates, either a 2-D coordinate (x,y) or a 3-D coordinate (x,y,z).

Table 53:

Resolution	Coordinate System	Interaction Object(s)
High	x,y,z	Textbox (numeric input)
High	x,y	Textbox (numeric input)
Low	x,y,z	Textbox (numeric input)
Low	x,y	point to drawing window

13.1.4 The CLASSIC Knowledge Representation Language

The addition of a knowledge base strengthens the declarative approach. Frame-based (slot-and-filler) knowledge base systems work well with pattern-based detection methods [4]. If data structure and attribute patterns for interaction tasks and objects can be identified, then a frame-based knowledge representation can be used to describe them. This knowledge base can then be used to aid in the detection of user interface components from legacy code, and then used to identify the appropriate replacement interaction objects within a specific toolkit such as MS-Windows or Motif.

The CLASSIC Knowledge Representation from AT&T Bell Laboratories [13] is a frame-based system that is particularly well suited for our purposes. CLASSIC focuses on objects rather than logical sentences, allowing interaction tasks and objects to be described in an object oriented fashion. CLASSIC is a deductive system based on the definition of *concepts*, which are arbitrarily complex, composable descriptions of object classes. *Roles* are properties that can be used to define attributes and restrictions on the concept descriptions, and *individuals* are instantiations of objects.

The power of CLASSIC for the MORPH knowledge base is in the organization of concepts and individuals into a hierarchy. More abstract, or general, concepts will be higher in the hierarchy, while more specific concepts and individuals will be lower. This allows individuals to be matched with classes that they satisfy. For example, the “selection task” is a high-level concept, while the concepts for the abstract objects “menu” and “pushbutton” are underneath “selection”. Concepts and individuals for specific toolkit components, such as the “Choice” in Java AWT, are underneath the abstract objects in the hierarchy.

CLASSIC also provides a variety of inferencing capabilities. Its completion inferences include contradiction detection, combination, inheritance, and propagation. It also provides *classification*, which allows concepts to be identified if they are more general or more specific than a given concept, and *subsumption*, which answers questions comparing the relative order of two concepts in the hierarchy. Classification in particular can be used to infer the most appropriate specific toolkit component from a description of an abstract interaction object.

13.1.5 MORPH Representation

This section illustrates the construction of the concept hierarchy using the *selection* interaction task as an example. Selection, allowing the user to make a choice from a set of alternatives, is one of the most-used basic interaction tasks, especially in character-oriented applications. Selection choices can allow the user to submit commands, such as save or edit, or allow the user to set system attributes and states (such as turning on boldface type). Selection interaction objects range from buttons of various kinds to text entry and menus. Attributes that are important in determining the appropriate selection mechanism include:

- Action when selected - When a selection mechanism is invoked, it either causes an action to occur (procedural), or it changes a state in the user interface or in the application.
- Number of States - All selection objects have a range of states or choices, from one to infinity. This attribute describes the number of states available for that selection object.
- Choice List Type - whether the number of choices in the choice list is fixed, or whether it can be added to dynamically (variable).

The attributes of these selection tasks are formalized in Classic as roles, with the following concepts defining their possible values. The concept definition names the possible values of a choice set by incorporating a higher-level concept, SELECTION-PROPERTY. CLASSIC provides operators, such as *one of*, that allow the concept to specify a choice:

```
(cl-define-concept  'SELECTION-ACTION
      '(and SELECTION-PROPERTY
      (one-of
        Procedural-Action
        Visible-State-Change) ))

(cl-define-concept
      'SELECTION-NUMBER-OF-STATES
      '(and integer (min 1)   ))

(cl-define-concept
      'SELECTION-VARIABILITY
      '(and SELECTION-PROPERTY
            (one-of Fixed Variable)  ))

(cl-define-concept
      'SELECTION-GROUPING
      '(and SELECTION-PROPERTY
            (one-of Grouped Not-Grouped)  ))
```

The Selection interaction task itself is defined by the composition of the various roles. The *at-least* operator asserts that the named role must be filled in any individual created from that concept:

```
(cl-define-primitive-concept  'INTERACTION-OBJECT 'classic-thing)

(cl-define-concept
     'SELECTION-OBJECT
     '(and INTERACTION-OBJECT
         (at-least 1 action)
          (all action  SELECTION-ACTION)
          (at-least 1 number-of-states)
          (all number-of-states
         SELECTION-NUMBER-OF-STATES)

         (at-least 1 variability)
         (all variability
            SELECTION-VARIABILITY)
         (all grouping
            SELECTION-GROUPING)
      )
)
```

The definition of INTERACTION-OBJECT is at the top of the CLASSIC concept hierarchy; all of the MORPH definitions are subsumed by it, allowing the CLASSIC inference engine to work. Notice that all roles except for SELECTION-GROUPING are required; this is because the only interaction object that refers to grouping is the radio button, which is used to implement a selection from a mutually exclusive set of choices.

The interaction objects are defined and differentiated by restrictions on the roles, as illustrated in the definition of radio button and basic menu:

```
(cl-define-concept 'MORPH-RADIO-BUTTON
      '(and SELECTION-OBJECT
       (fills action
          Visible-State-Change)
       (fills number-of-states 2)
       (fills variability Fixed)
       (fills grouping Grouped)
       )
)

(cl-define-concept 'MORPH-BASIC-MENU
       '(and SELECTION-OBJECT
         (fills action
            Procedural-Action)
         (all number-of-states
               (and integer
                 (min 2) (max 15)))
         (fills variability Fixed)
         (fills grouping Not-Grouped)
       )
)
```

13.1.6 Transformation by Inferencing

The next task is to perform transformations from the MORPH abstractions to specific toolkit instances. For our test, the Java Abstract Windowing Toolkit (AWT) [12] was added to the knowledge base. Following are the CLASSIC descriptions for the “Grouped Checkbox” (corresponds to the MORPH-RADIO-BUTTON) and the “Choice” (corresponds to the

MORPH-BASIC-MENU). In order to get a fair test of the inferencing capabilities, no assumptions are made about the hierarchy and the parent concept is INTERACTION-OBJECT, not SELECTION-OBJECT:

```
(cl-define-concept
     'AWT-Grouped-Checkbox
     '(and INTERACTION-OBJECT
            (fills action
              Visible-State-Change)
             (fills number-of-states 2)
            (fills variability Fixed)
            (fills grouping Grouped)
       )
)
(cl-define-concept 'AWT-Choice
     '(and INTERACTION-OBJECT
      (fills action Procedural-Action)
      (all number-of-states
          (and integer (min 2) (max 10)))
      (fills variability Fixed)
      (fills grouping Not-Grouped)
     )
)
```

The inferencing test is performed by providing a MORPH abstract concept to CLASSIC and asking which higher-level concepts subsumes it, which gives us the mapping to the specific Java component AWT-CHOICE. The screen capture below shows the interactive CLASSIC session to perform this query, and the response from the CLASSIC knowledge base on the next line:

```
Classic> (cl-concept-parents
             @morph-basic-menu)
(@c{AWT-CHOICE} @c{SELECTION-OBJECT})
```

CLASSIC returns AWT-CHOICE as the best match for MORPH-BASIC-MENU, and also indicates that the interaction task is SELECTION-OBJECT. Even though the Java AWT components were not directly described as SELECTION-OBJECTS, the CLASSIC system makes the match because of the values of the role fillers. In this way, the MORPH abstract model objects can be mapped to any specific toolkit that is described in the knowledge base.

13.1.7 Design Critics

Another advantage of using a knowledge representation for transforming user interfaces is that design and usability heuristics can be incorporated into the knowledge base. A "design critic", or guideline, is shown in the range of 2..15 defined in the MORPH-BASIC-MENU definition. A decision was made that any selection with a large number of choices should be handled by another interaction object, a scrolling list. Therefore only selection tasks with choice lists smaller than 10 will ever be mapped to MORPH-BASIC-MENU, and larger lists will be mapped to MORPH-LIST. This enhances the usability of the resulting user interface.

13.2 The MORPH Pattern Set for User Interfaces

(patterns paper contents here)

13.3 The MORPH Toolset

The MORPH tools implement the MORPH process. This section describes the overall system dataflow and each component of the toolset.

13.3.1 Conceptual Dataflow

As shown in figure 2, legacy code is processed first by a recognizer, which identifies the pieces of the legacy code that implement the user interface. When the recognizer detects a user interaction object, it reports this to a modeler, which represents the abstract design of the user interface. Human input from the analyst/designer can modify and refine the user interface model, and when it is completed, it is fed into a transformer. The transformer maps the user interface abstractions into a specific GUI toolkit or widget set, and produces a new user interface. Details of each component are described below.

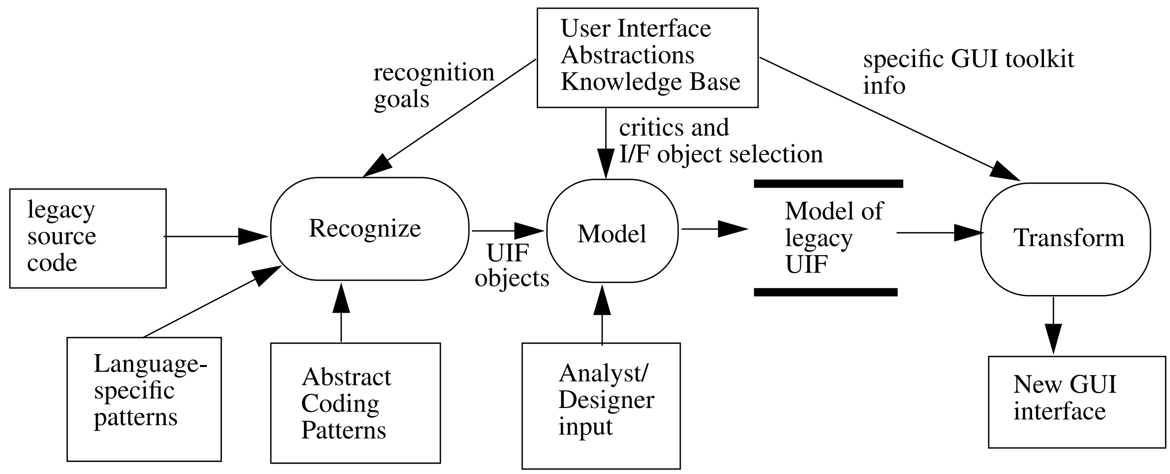


Figure 19 : MORPH Dataflow

13.3.2 System Components

Knowledge Base

The Knowledge Base is a central component of the MORPH toolset. It contains information about user interaction task abstractions, user interface object abstractions, and also specific GUI toolkit information. The Knowledge Base is used in all phases of the MORPH process as well, from detection to modeling to transformation.

- Catalog of User Interface Abstractions - User interface objects are described according to the basic interaction task they implement, plus defining characteristics and attributes.
- User Interface Object selection heuristics - Design critic rules are incorporated into the knowledge base in order to use inferencing to select the best interaction object for a particular task.
- Specific GUI toolkit descriptions - in the transformation step, information about specific GUI toolkit objects and widgets can be used to map from the abstract model to the concrete implementation of the GUI.

Recognizer

- Abstract Pattern Detection Rules - these non-language-specific rules describe patterns that implement basic interaction tasks and attributes of interaction objects in text-based code. The recognizer uses these rules to detect user interface components from the code.
- Language-specific patterns - A set of language-specific constructs are provided to allow MORPH to look for particular patterns in the legacy code (for example, a definition of the I/O routines for a specific language).
- Knowledge base interface for recognition goals - The knowledge base partially drives the detection process by narrowing the list of possible patterns that can be recognized by context. For example, if a selection mechanism has been detected, the knowledge base can infer that further selection mechanisms may indicate a menu hierarchy.

Modeler

- Representation of the generated model - Once the interaction objects and the dialogue control pieces have been detected by the recognizer, the modeler stores and displays the model as it is being built.
- Human judgement input - model editing allows the analyst or designer to make changes to the model as it is being developed.
- Knowledge base critics and object selection - heuristics for choosing an appropriate interface object can be used by the modeler to produce a higher-quality user interface.

Transformer

- Specific target GUI toolkit information is used to transform the abstract model into a concrete implementation in a particular GUI technology. For example, the abstract interaction objects could be mapped to the closest match in Motif widgets by using inferencing if the Motif widgets were also described in the knowledge base.

13.3.3 MORPH Toolset Architecture

The MORPH system is built using Refine and CLASSIC. Following is an architectural description:

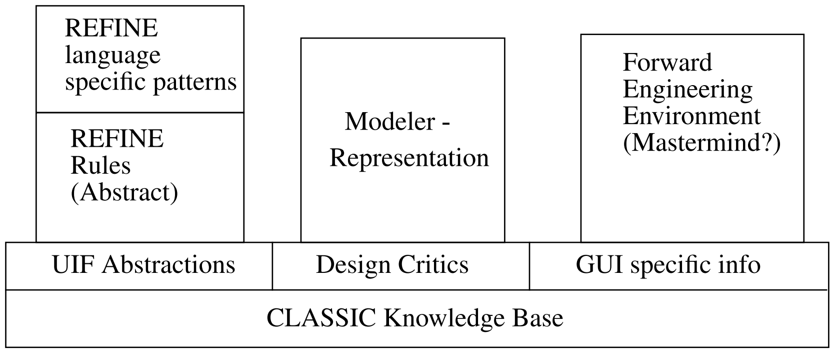


Figure 20 : MORPH Architecture

The cornerstone of the MORPH toolset is a knowledge base implemented in CLASSIC, which is an object-oriented knowledge representation language [RES93]. The Software Refinery [REA??] is used to implement the abstract pattern recognition rules and the language-specific patterns for the detection step. The Modeler is an editor that allows the analyst to view and modify the model that is being built. The Forward Engineering step will be tied into an existing GUI-building toolset such as Mastermind or Amulet.

13.4 Input formats

There are several important inputs in the MORPH technique:

- The Knowledge Base, which contains descriptions of user interface abstractions
- The set of code patterns that implement the user interface abstractions for the detection step

13.5 Output formats

13.6 Steps

13.7 Evaluation

14. Glossary

analyst - an individual who performs program understanding tasks on an existing system.

basic interaction task - (see interaction task) an indivisible interaction task - if the interaction task is decomposed, it is no longer meaningful to an application. [FOL90]

composite interaction task - Aggregate basic interaction tasks - combinations of basic interaction tasks into combined tasks that have a new meaning to an application. [FOL90]

data entry - input from the user to the computer via an input device (keyboard, mouse, etc.) that is not interpreted by the application program, but treated as information to be stored. For example, entering a typed command is not data entry, whereas entering a user's address is data entry.

designer - the software engineer responsible for designing and implementing a new application.

gesture - (as used in [HEN90]) - A sequence of user interactions with an application that can be recognized as belonging to a set of symbols that have meaning to the application. For example, textual handwriting recognition can be accomplished by a gesture recognizer that analyzes the input and matches it to known alphanumeric symbols.

input device - a piece of computer hardware that allows a user to interact with a computer. The concept of logical devices [FOL90] groups input devices into five equivalence classes: locator (position), pick (selection), valuator (numeric), keyboard (strings), and choice (actions).

interaction objects - as defined in [DEB92], are "controls such as buttons, sliders, or menus that can be manipulated directly by the user. Every interaction object in the user interface is associated with an action or attribute in the application data model". As defined in [HIX93], interaction objects are precoded objects in toolkits that can be used in implementing a user interface.

interaction styles defined in [HIX93] are described as "a collection of interface objects and associated techniques from which an interaction designer can choose when designing the user interaction component of an interface. They provide a behavioral view of how the user communicates with the system." In [SHN93], primary interaction styles are menu selection, form fill-in, command language, natural language, and direct manipulation.

interaction task is the entry of a unit of information that is meaningful to an application by the user [FOL90]. Interaction tasks classify the fundamental types of information entered with the interaction techniques. Interaction tasks are defined by what the user accomplishes, not how it is accomplished. The four basic interaction tasks are:

- Selection - allows the user to choose from a set of choices.
- Text Entry - provides unvalidated data entry and output messages to the user.
- Quantify - allows the user to specify a numeric value.
- Position - rarely used in text-based interfaces, allows the user to specify a position in screen or world coordinates.

interaction technique as defined in [FOL90] is a construct that "implements the hardware binding portion of a user-computer interface design". An earlier informal definition adds that interaction techniques are "ways to use input devices to enter information into the computer". Interaction techniques are also defined as primitive building blocks from which a UIF is crafted.

interaction technique toolkits [FOL90] are subroutine libraries of interaction techniques, mechanisms for making a collection of techniques available for use by application programmers (for consistent look and feel).

interactor - (Amulet) In a GUI presentation, interactors are invisible objects that attach to graphical objects in order to allow them to respond to input. [

legacy system - An application that has been developed and maintained over a period of time, typically its original designers and implementors are no longer available to perform the system's maintenance. Often specifications and documentation for a legacy system are outdated, so the only definitive source of information about the system is the code itself.

reengineering - The use of design recovery techniques to understand the code of a legacy system in order to modify an existing system. Typically reengineering is performed to correct errors, restructure a system to improve maintenance, to add functionality, or to port the system to another platform. Also called renovation or reclamation [CHI90].

reverse engineering - The process of analyzing a software program in an effort to create a representation of the program at a higher level of abstraction than source code. Also called design recovery, the process can consist of data extraction, architectural specification, control flow and data flow understanding. [PRE92]

terminal control semantics - in some text-based applications, the cursor can be moved around the screen using keyboard commands or keystrokes. If the application ascribes meaning to the location of the cursor, then the terminal control has semantics to the application. For example, if the screen is split into two areas, and moving the cursor from one area to another determines the meaning of a data value that the user enters, then the application has terminal control semantics.

text-based application (interface) - An application that accepts user input only from a keyboard. Typically a text-based application only displays textual data as well, however this is not a requirement. Alphanumeric characters, as well as function keys and arrow keys can be used to interact with a text-based application.

text entry - Any alphanumeric characters or strings entered by the user. Can be used as data entry (see definition) or as a selection mechanism, as in typing the name of a command.

user - the individual that the software application was designed and implemented for - the "end user" of the system.

user interface elements - [Mastermind term] Pedro's presentation model paper

user interface software - [HIX93] The means for implementing the code that instantiates the user interaction component. This is the "system" or "application" view of a user interface.

user interaction component - [HIX93] The interaction component of an application is its "look and feel", the application's response to what a user sees and hears and does while interacting with the computer. This is the "user view" of a system.

widget - [HIX93] another name for user interaction objects, specifically used to describe X Windows interaction objects, but sometimes used as a general term to describe objects in user interface toolkits. Examples of widgets are scrolling lists, pushbuttons, and cascade menus.

15. Annotated References to the Literature

(How this method relates to other methods, discriminators, links to further information)

Merlo Macroscope

Van Sickle

Commercial methods

16. References

[CHI90] Chikofsky, Elliott, and Cross, James. "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software,* January 1990.

[DEB92] deBaar, Dennis, Foley, James D., and Mullet, Kevin E. "Coupling Application Design and User Interface Design", Proceedings of CHI '92, May 3-7, 1992.

[FOL90] Foley, James D., van Dam, Andries, Feiner, Steven K., and Hughes, John F. *Computer Graphics Principles and Practice*, Second Edition, Addison-Wesley Publishing Company, Addison-Wesley Systems Programming Series, 1990.

[FRO86] Frost, Richard. *Knowledge Base Systems,* MacMillan Publishing Company, New York, NY, 1986.

[HEN90] Henry, Tyson R., Hudson, Scott E., and Newell, Gary L. "Integrating Gesture and Snapping into a User Interface Toolkit", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST)*, Snowbird, Utah, Oct 3-5, 1990.

[HIX93] Hix, Deborah, and Hartson, H. Rex. *Developing User Interfaces - Ensuring Usability Through Product and Process*, Wiley Professional Computing Series, John Wiley and Sons, Inc., 1993.

[MOO96a] Moore, Melody. "A Taxonomy of User Interface Abstractions", PhD Working Paper, College of Computing, Georgia Institute of Technology, November 1996.

[MOO96b] Moore, Melody. “Rules to Detect Interaction Tasks and Presentation Attributes in Character-Oriented Legacy Code”, PhD Working Paper, College of Computing, Georgia Institute of Technology, Oct 22, 1996.

[MYE96] Myers, Brad, Ferrency, Alan, McDaniel, Rich, Miller, Robert, Doane, Patrick, Mickish, Andy, and Klimovitski, Alex. *The Amulet V2.0 Reference Manual*, School of Computer Science, Carnegie Mellon University, 1996.

[PRE92] Pressman, Roger S. *Software Engineering - A Practitioner’s Approach, Third Edition*, McGraw-Hill, 1992.

[RES93] Resnick, Lori Alperin et al. CLASSIC Description and Reference Manual for the COMMON LISP Implementation Version 2.1, AT&T Bell Laboratoris, Murray Hill, N.J., May 15, 1993.

[SHN93] Shneiderman, Ben. *Designing the User Interface, Strategie for Effective Human-Computer Interaction*, *Second Edition*, Addison Wesley, 1993.

Table of Contents