MORALE
Mission Oriented Architectural Legacy Evolution

Gregory Abowd, 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

1. Introduction

1.1 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 behavior. 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 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.

1.2 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.

1.3 Approach

There is an unexpected symmetry between requirements analysis and reverse engineering that can be exploited to improve the process of mission-driven system

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.

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, using both static and dynamic analyses. 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 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.

2. The MORALE Software Evolution Method

2.1 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

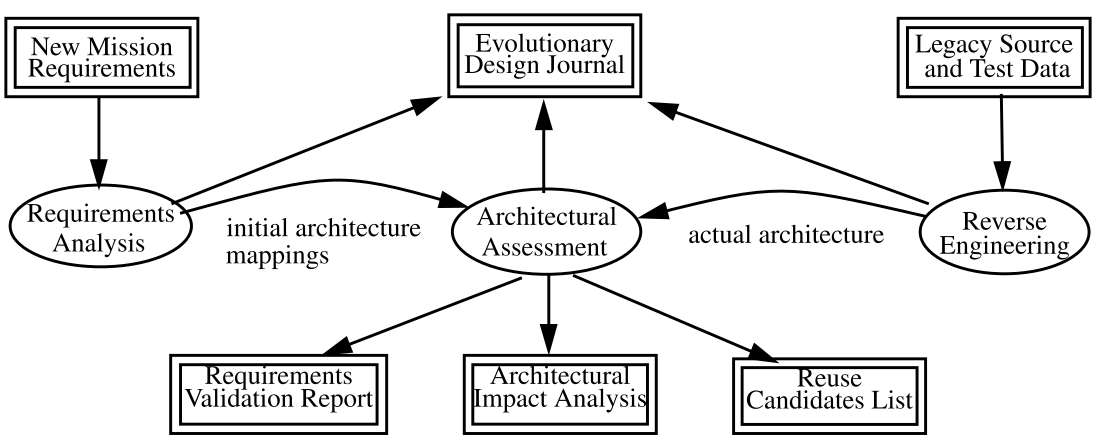

Figure 1: MORALE inputs, activities and outputs

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 to 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 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.

2.2 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 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.

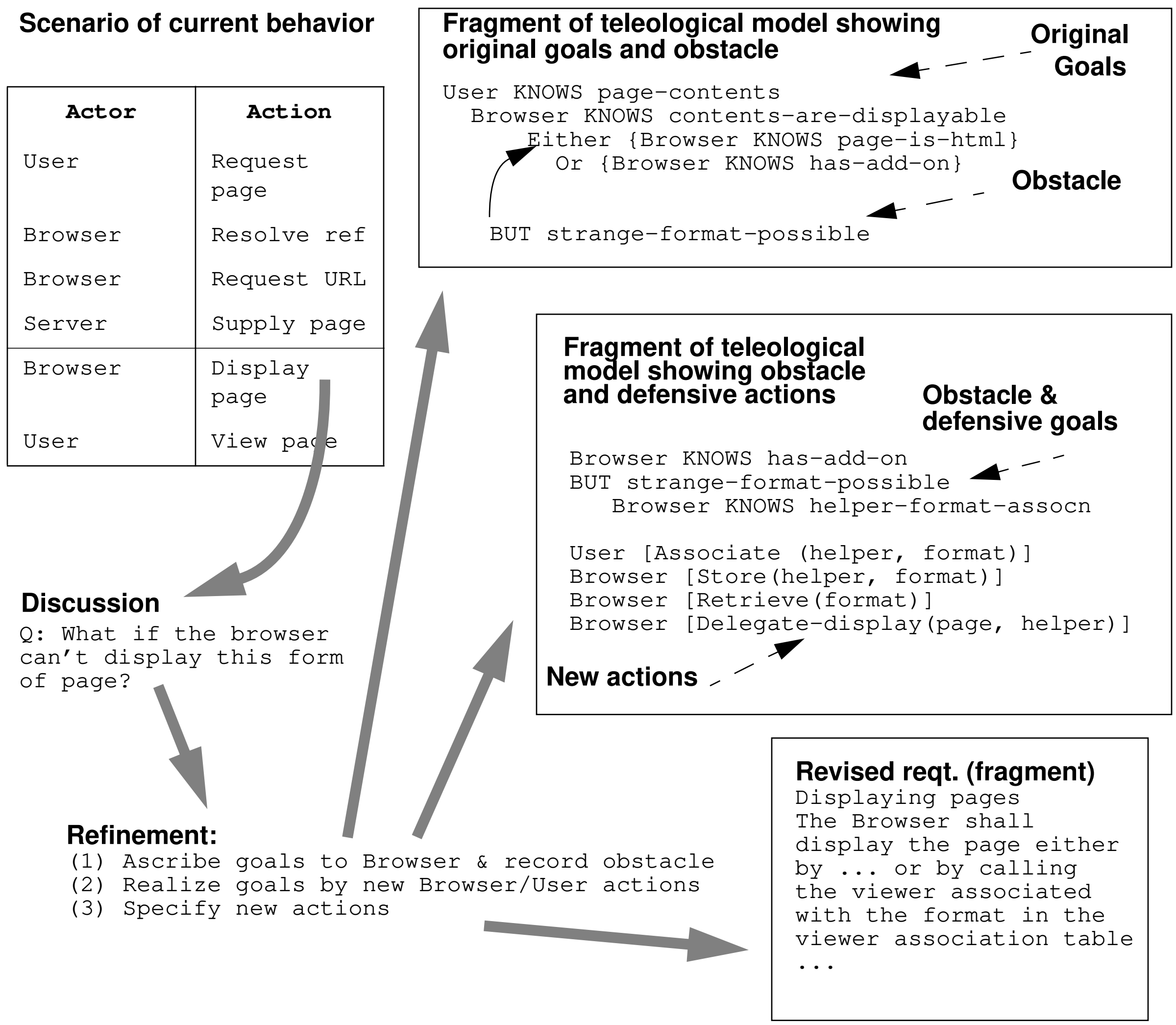

Figure 2: Illustration of ScenIC being applied to the addition of a functional requirement
Exploring a scenario results in a question that is answered by refining the teleological model and specifying a new functional requirement.

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 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.

2.3 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.
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,

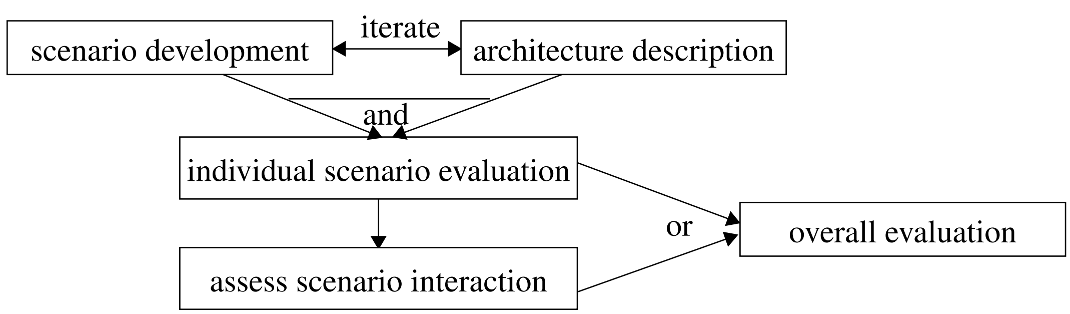

Figure 3: Activities and dependencies in SAAM's scenario-based analysis

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 subject of changes from this new requirement.

2.4 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

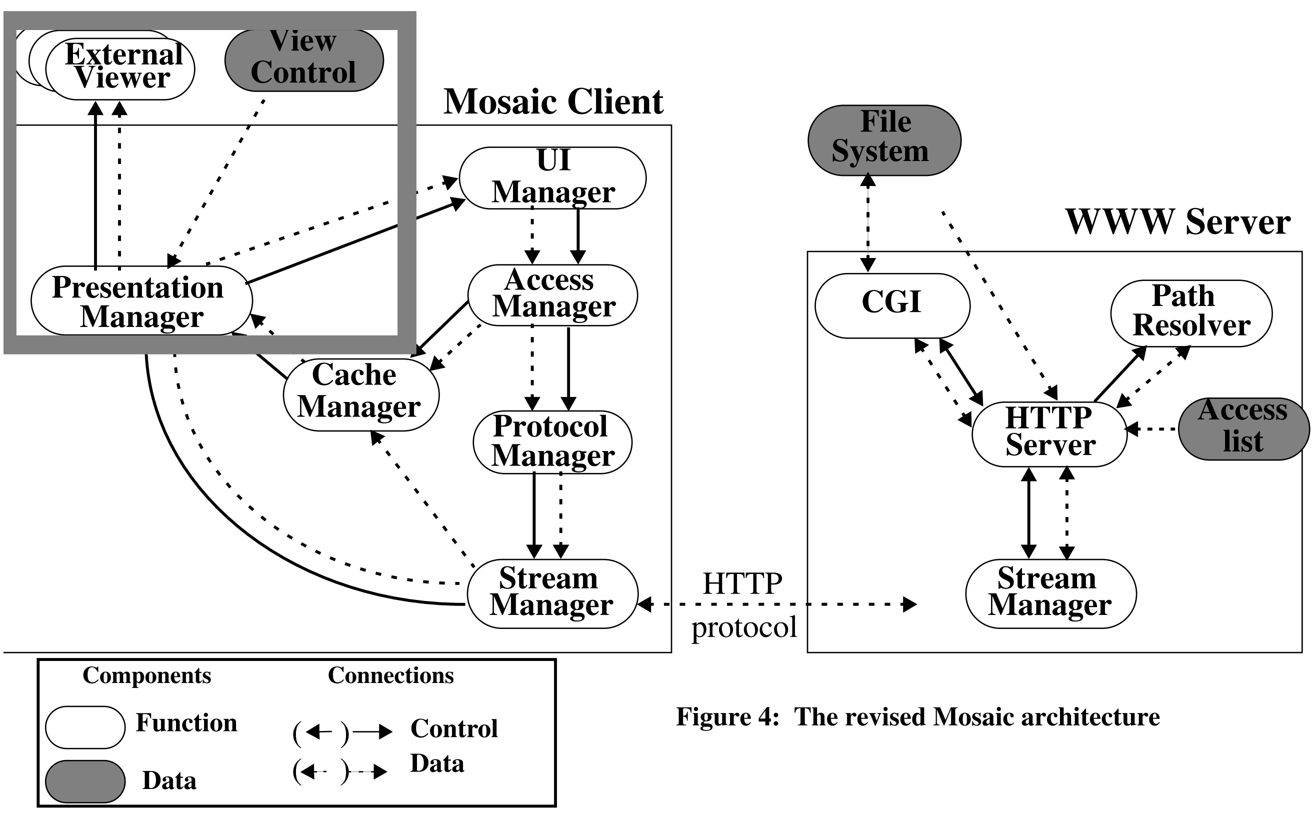

Figure 4: The revised Mosaic architecture

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 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) [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

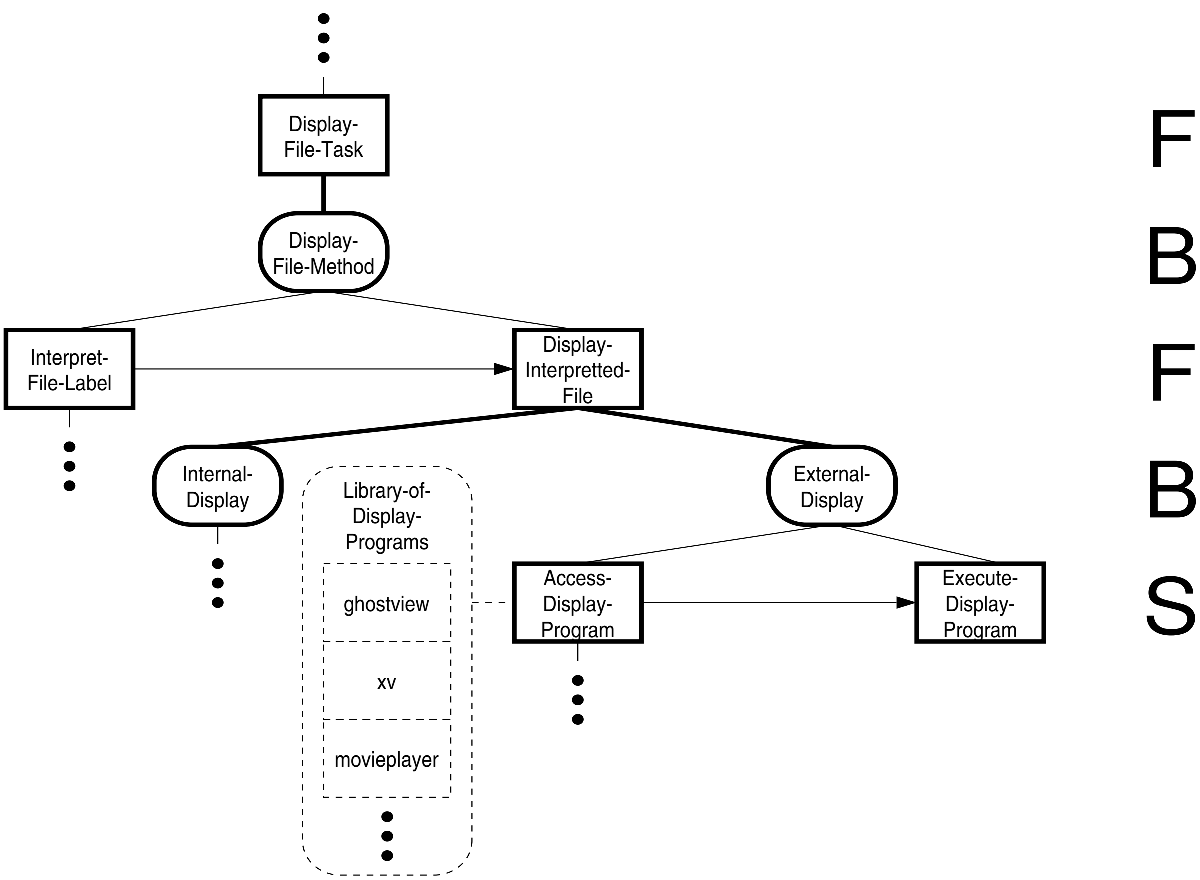

Figure 5: A partial SBF model of a Mosaic 2.4.

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.

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.

2.5 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.

Detecting the user interface model from 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 alterna-

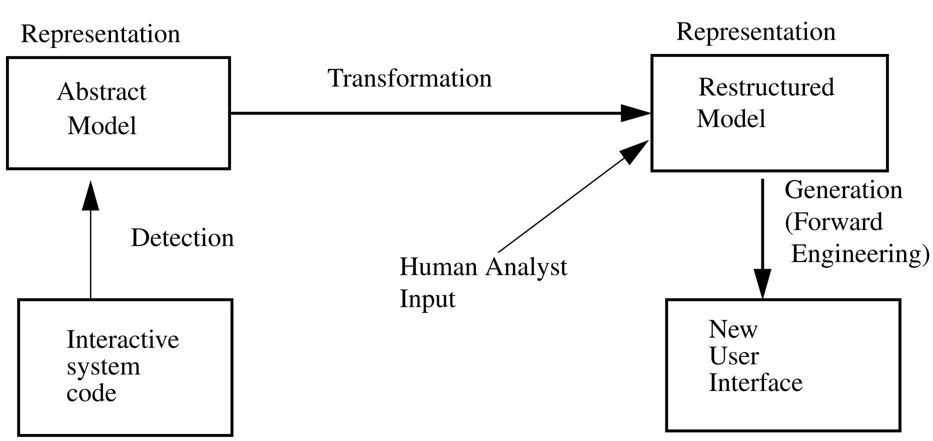

Figure 6: The MORPH Reengineering Process for User Interfaces

tives 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.

3. 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.

3.1 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].

3.2 ISVis

ISVis (Interaction Scenario Visualizer) [10] is a tool that supports software engineering tasks requiring dynamic program understanding. The tool uses static analysis data to instrument a subject system, so that when run, program execution traces are generated. These traces are read by ISVis, and views of the actors and interactions in the traces are 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.

3.3 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.

3.4 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].

3.5 SAAMTool

The SAAM process for impact analysis can involve many 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.

4. 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 only loosely integrated. We need to develop a common data model to support interoperation and a unified user interface.

Acknowledgments

Effort sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-2-0229. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

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] Dean Jerding, John T. Stasko, and Thomas Ball. "Visualizing Interactions in Program Executions." *Proceedings of the International Conference on Software Engineering*, 1997, to appear.

[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.