

Getting to Why¹

**Kenneth S. Rubin
Adele Goldberg
ParcPlace Systems
999 East Arques Ave.
Sunnyvale, CA 94086
(408) 481-9090**

The Problem Statement

One fundamental measure of maturity of a technology is whether it can be used to create large systems that can be validated and verified. A key to reaching this level of maturity is traceability. *Traceability* connotes connectivity, whereby artifacts created during the development of a system are linked in meaningful ways.

Traceability is bi-directional. Tracing forward means that from premises it is possible to explain conclusions. Tracing backward means that conclusions can be tied back to assumptions, supporting facts, and the goals and objectives that led to the conclusions. When a system is constructed such that its artifacts are traceable, it is possible to say *why*:

- Why goals and objectives are met with the proposed system (verification)
- Why elements in the resulting system are necessary (validation)

A complete discussion of traceability in the context of object-oriented technology requires a description of how traceability is applied across all activities associated with object-oriented development. In this paper, we focus on how traceability impacts the formulation of an analysis methodology. This is not only the natural starting point, but also the most critical because ignoring traceability during analysis subverts its usefulness throughout the entire life cycle. To understand why this is so, we begin by overviewing the nature of analysis. We then discuss the importance of traceability in validating and verifying the developed system. Next, we discuss how traceability is supported in the Object Behavior Analysis (OBA) Methodology [Rubin92].

The Nature of Analysis Methodologies

Analysis is the study and modeling of a given problem domain, within the context of stated goals and objectives. Analysis focuses on what a system is supposed to do, rather than how it is supposed to do it. An analysis methodology is a set of concepts, techniques, notations, and tools that help guide an analysis process. Having defined these terms, let's examine the nature of analysis and supporting methodologies.

Assume that a customer has detailed knowledge of a problem domain. Furthermore, the customer has certain goals and objectives surrounding this domain—perhaps the customer wants to automate part of the domain. The simplest form of analysis is when the customer, as owner of the understanding of the problem, uses this understanding to directly construct a system to meet the desired goals and objectives. Take, as an example, a business person who understands a particular financial problem and directly utilizes a spreadsheet to express a solution. In many such cases, the problem is small—methodologies and tools are well known and closely related to the problem. There is no need to involve a third party in the analysis of the problem or, for that matter, in the design and implementation of the solution.

¹Paper to appear in the special insert on object-oriented methodologies in the July/August issues of the *Journal of Object-Oriented Programming*, *Object Magazine*, and the *C++ Report*.

Customer as developer may work well for small problems, but it rarely scales to larger systems. We are interested in the development of these larger systems. Figure 1 illustrates a methodology model for conducting analysis of such systems. Although this model is independent of any particular technology, we will point out where the model benefits from use of object-oriented technology.

In the most generic sense, analysis involves two individuals (actually roles) working through three information spaces: Concept, Articulation and Model². The first person is a user or customer (an expert)³ who has a particular understanding of the problem to be solved. We assume the problem is quite complex, and is maintained in the expert's Concept Space. Physically speaking, Concept Space is the head of the expert. By definition, this understanding is multi-faceted and highly associative.

When the expert who owns the concept is the one developing the system, the principle manifestation of the concept outside of his or her head is likely to be a direct solution (e.g., a spreadsheet as in the prior example). The real issues of analysis (and therefore traceability) begin when the expert must communicate the concept to other people—to analysts, as represented in Figure 1. Since the concept is often very rich and expansive, it is generally not possible for experts to adequately communicate their entire understanding in a single, holistic expression. As a result, they are forced to provide multiple articulations of various aspects of the concept. The goal of the analyst is to prompt for and capture these articulations (we call this Articulation Space), and piece them together into a coherent model (Analysis Model Space).

The Modeling Concepts and Analysis Techniques indicated in Figure 1 are particular to a methodology, and guide the appropriate use of Articulation and Analysis Model Spaces. The Modeling Concepts are the foundation. They dictate the core ideas that are used to represent the expert's concept inside of Analysis Model Space. In object-oriented analysis, these Modeling Concepts are the core concepts of object-oriented technology, such as object encapsulation, inheritance and polymorphism, as they are applied to both static and dynamic object models. Since there is a transformation from Concept to Analysis Model Space, the goal is to reduce the distance between the two spaces.⁴

In the best of all worlds, the identity transformation would be used to move from Concept to Analysis Model Space—each aspect in the expert's concept model would have a direct correspondent in the analysis model. One of the benefits of object-oriented technology is that the transformation from Concept to Analysis Model Space is quite natural and efficient. In other words, object-oriented technology provides a means of modeling a problem that closely corresponds to the way experts think about the problem.

²These ideas were developed in concert with Gurdon Blackwell of Gemini Consulting, Morristown, New Jersey.

³Henceforth we refer to this person as the expert to connote that this is the person with the expert understanding of the problem to be analyzed.

⁴Any transformation introduces the possibility of errors when moving from one space or representation to another. This is a failing of Structured Analysis and Structured Design—a significant number of large transformations are required when moving through the steps of the process.

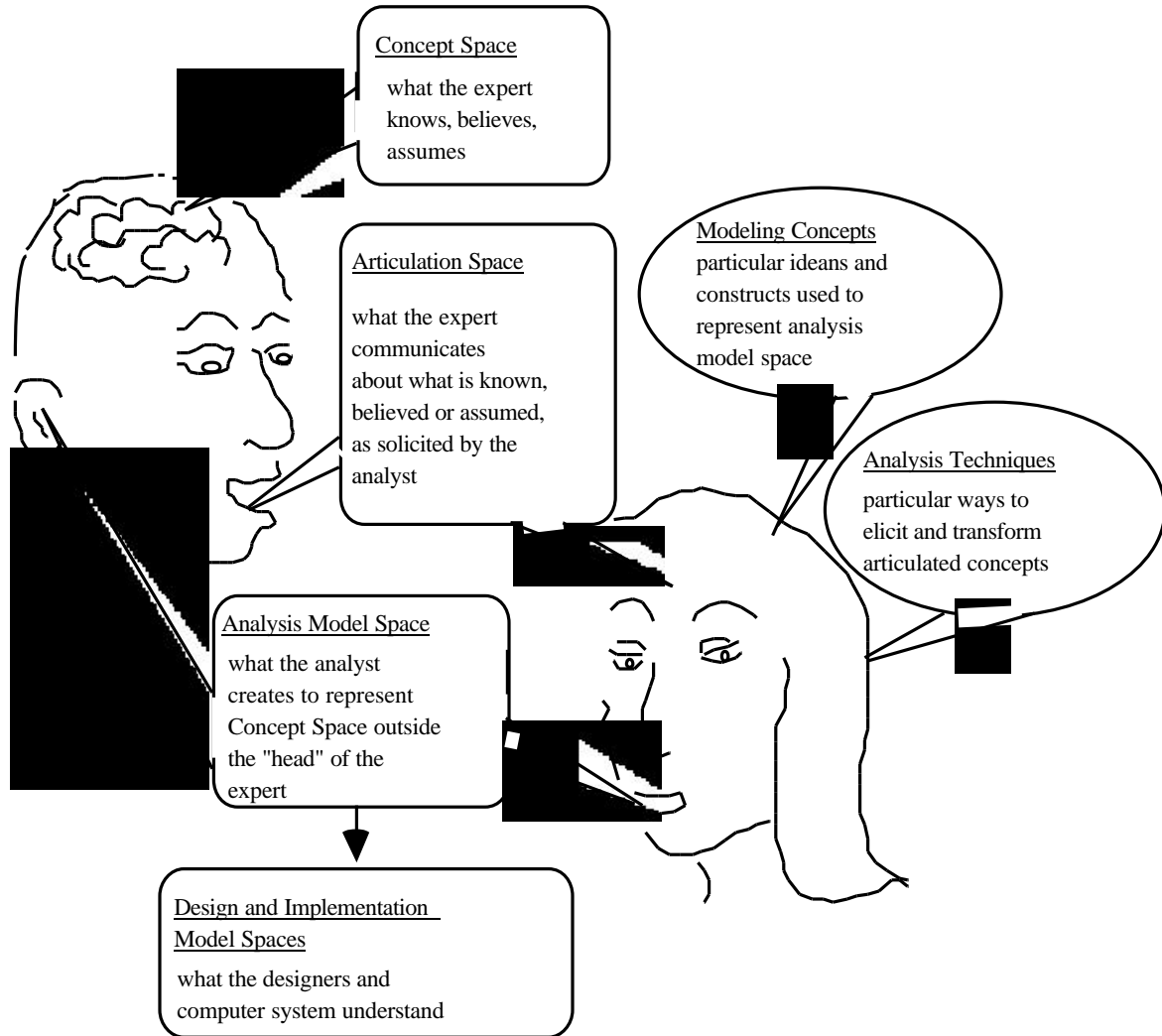


Figure 1. A Model of Methodology

The power of keeping the transformation between Concept and Analysis Model Space small is illustrated in Figure 1. Notice that the expert reviews the analysis model. This is done to determine whether the analyst's model of the problem accurately depicts the expert's concept of the problem. If the distance between the analysis model and the concept is large, it will be quite difficult, if not impossible, for the expert to determine accuracy. So, one of the principle requirements of any analysis model is that it be understandable to the expert. Again, the extent to which this is possible is directly related to the Modeling Concepts' ideas and constructs.

With the Modeling Concepts in place, an analysis methodology must also specify a set of Analysis Techniques. These techniques are employed by the analyst in the context of both Articulation and Analysis Model Space. It may be helpful to think of Articulation Space as a convenient place for expert and analyst to rendezvous. Here they agree on how they will talk with one another (possibly including the structure of sentences and definitions of words to be used). The precise nature of the rendezvous depends on the chosen Analysis Techniques. For example, within OBA methodology, experts are asked to articulate their understanding in the form of scenarios—descriptions of how activities take place. These scenarios are converted into scripts by the analysts, where scripts have properties including pre- and post-conditions, initiators and participants in actions, and services provided by participants to carry out actions. In OBA, scripts are used to create both static and dynamic models.

After a model of the problem has been created in Analysis Model Space, it is transformed into Design and Implementation Model Spaces.⁵ During each of these transformations, additional constraints are added to the model so that it can meet stated resource and quality objectives. We will examine these spaces in the context of traceability.

The Importance of Knowing Why

The thesis of this paper is that knowing why—being able to trace both forward and backward through a collection of artifacts—is essential for evaluating that we have built the right system, and for evaluating that we have built the system right. The previous section presented a model of analysis that we use to illustrate this point. Using Figure 1, we described how analysts assist experts in articulating their concepts into an analysis model. The first issue of traceability is:

- Why do we think that the analysis model is correct?

Since as analysts we are not able to get into the heads of the experts and see their concepts, how do we trace our model back to their concepts? In practice there is no way to do this, so the best that we can hope for is to establish confidence that we have modeled the concept to an appropriate level of fidelity and accuracy. It is here that we rely on object-oriented modeling to reduce the transformation distance between Concept and Analysis Model Space. As a result, the expert is able to understand the Analysis Model Space and map it back to the concepts he or she understands.

Next we wish to build confidence that the proper system is being built. We can only do this at the level of system features, and mainly by relying on close customer verification of the deliverables. We must be able to verify the contents of each model space. To do so, it must be possible to answer why:

- Why does a particular feature or part of the system exist?
- Why will a change, for example in specification, create a particular impact?

The issue of why something exists relies on backward traceability. It is the ability to point at an artifact in the system and ask—why is this here? What are the goals and objectives that are supported by this artifact? It is quite likely that an artifact will not trace directly back to a goal or objective, without first

⁵We do not mean to imply that all of analysis must be completed before design begins. On the contrary, we advocate an incremental style of development. But in the development of a particular incremental part of a system, it is logical to do analysis before design.

tracing through intermediary artifacts. This implies that we have a transitive traceability network—artifacts are connectable through a series of intermediate artifacts. When the network provides full coverage, any artifact can be traced back to some foundational artifact, such as a goal or objective.

The issue of why some change impacts a particular analysis, design or implementation artifact relies on forward traceability. In particular, we are interested in knowing which artifacts in the system will change if a particular system goal or objective is changed. Complementary to the discussion on backward traceability, we are interested in determining the impact of the change down to the lowest-level meaningful artifact. For example, if a requirement changes, we may be interested in knowing how the code and/or test cases are affected.

We next examine both of these issues—backward and forward traceability—in more detail in the context of OBA, describing how OBA provides a transitive traceability network.

Traceability Model for Object Behavior Analysis

OBA is a specific approach to analyzing problem situations based on the model presented in Figure 1. OBA has been under development for more than four years, and continues to evolve from feedback provided by its users.⁶ OBA's development goals are:

- Provide a set of techniques to assist the analyst in extracting and modeling the customer's concepts
- Capture arbitrarily-ordered customer articulations that describe multiple aspects of the concepts
- Construct an analysis model that can be reviewed by the customer
- Avoid reliance on any specific notation
- Be able to trace results both backward and forward

Many of these development goals are detailed in [Rubin92]. Although the OBA solutions to several of these goals distinguish it from other methodologies, it is OBA's support for traceability that is most prominent.

Figure 2 illustrates the OBA traceability model. The foundation for the model is the set of the goals and objectives for the system under development. *Business goals* identify the specific business reasons for building a system. As analysts, we do not reason about the appropriateness of these goals and objectives. We require them as the base from which, and to which, we can trace subsequent analysis artifacts. These goals and objectives are also used to measure our progress and success. Where possible, a list of system features should also be identified. Objectives differ from goals in that they are time-targeted, measurable descriptions of all key aspects of the project. There are several categories of objectives, notably resource and quality. *Resource objectives* define the people, time and money budgeted for the project. *Quality objectives* are quantitative descriptions of qualitative results. Typical quality categories are performance, reliability and reusability.

The next aspect of the traceability model shown in Figure 2 involves core activity areas. These are the major areas of the system that require analysis. Identifying these areas will provide a basis for the scripting process, and for work-partitioning and parallel development. Core activity areas are not necessarily subsystems. Within our software development strategy, a subsystem is a design artifact used to describe a coordinated set of objects that work together to provide a set of services. The collection of subsystems and how they relate defines the architecture of the system. Core activity areas are analysis artifacts whose realization at design time may span one or more subsystems. To better understand the difference, imagine that the system must support transactions that query a database and print results. Transactions that perform similar types of queries are classified into the same core activity, whereas each transaction at design time may require the services of one or more subsystems (e.g., database and printing subsystems). Each core activity area must trace back to one or more goals or objectives.

⁶This paper updates information published in [Rubin92], with some change to nomenclature and structure of artifacts.

Once the core activity areas have been defined, the next step is to determine what the system is supposed to do, and for whom and with whom it is supposed to do it. Our basic approach is to specify usage scenarios that cover all possible pathways through the system functions. We refer to this particular Analysis Technique as *scripting*. The scripting principles we use are adapted from those used in Cognitive Psychology and Artificial Intelligence. The basic premise is that one way people store their understanding (in Concept Space) is by episodic memory. This is essentially a sequence of events that make up a story or episode. Our goal is to extract this description (via articulations in Articulation Space) and model it (in Analysis Model Space). However, we wish to model it using object-oriented concepts. To do so, we assume there are a collection of entities in the system, each of which provides a set of well-defined services that can be used by other entities.

Work gets done when one entity communicates with another to notify that an event has taken place, to provide information, to request information, or to request a service. Scripts are designed to capture this information in the context of a particular use scenario that defines a sequence of service requests that accomplish some overall task. The notation in Figure 2 indicates that

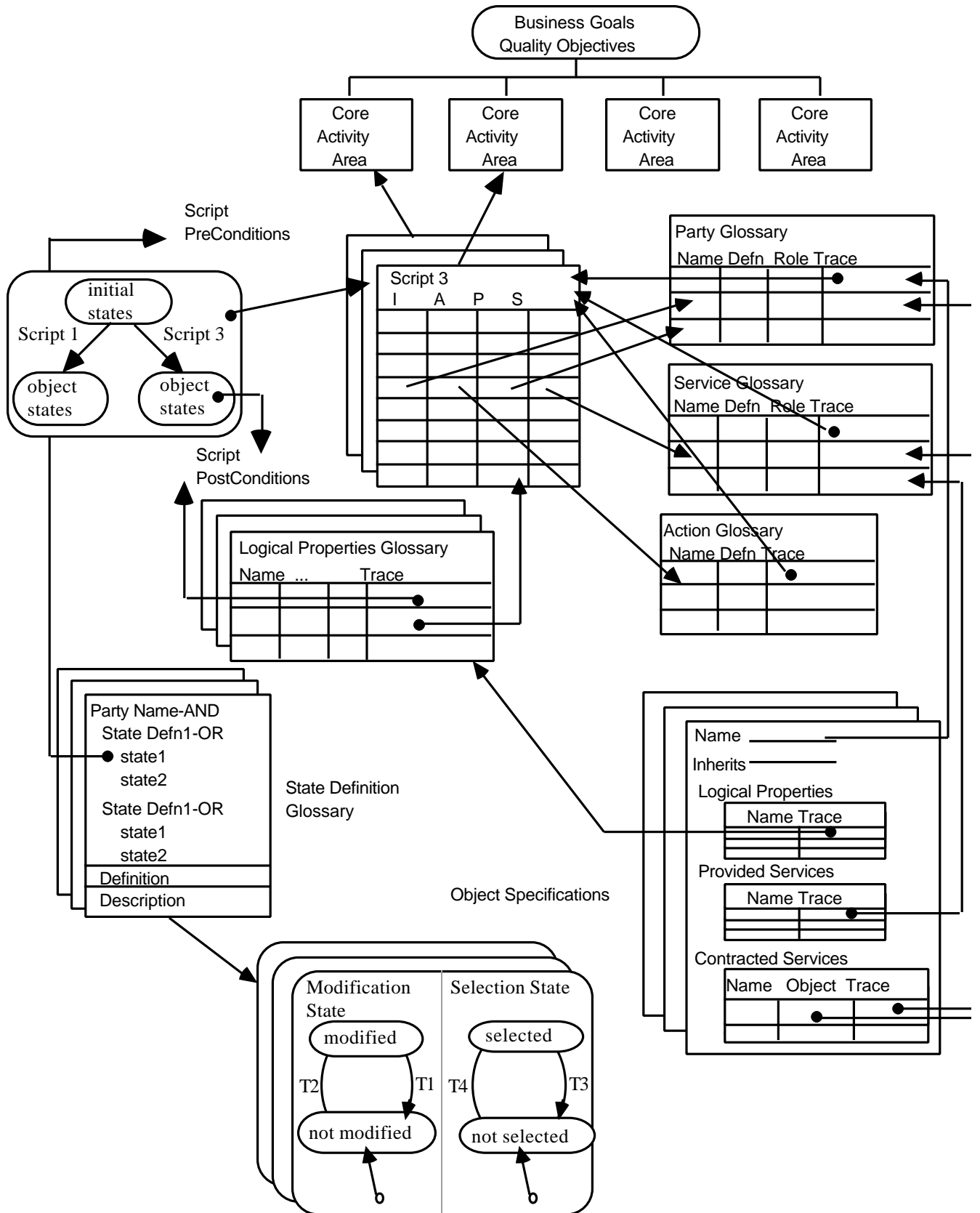


Figure 2. OBA Backward Traceability

Getting to Why

scripts contain sequences of contracts, where each contract consists of an initiator (I) of an action (A) that involves a participant (P). In order for the participant to carry out the initiator's requested action, the participant provides a service (S).

It is important to know why we choose to script some activities over others. In Figure 2, we see that a script traces to one or more core activity areas. This trace justifies the existence of the script. It is also used to identify which scripts are affected by a change in core activity area requirements.

Associated with each script is a set of preconditions and postconditions. The preconditions denote what is true of the state of the system in order for the script to be applicable. The postconditions denote what is true of the state of the system as a result of carrying out the script to completion. This information is used to generate both the object- and system-level dynamic models.

The object-level dynamic models are shown in Figure 2 as state definition glossaries and Harel Statecharts [Harel87] (two ways of representing the same information). An object-level dynamic model exists for every object in the system that has interesting state. An *interesting state* is any state associated with an object that causes it to alter its behavior. For example, an overdrawn bank account behaves differently than a bank account in good standing. The representations show all possible state changes of an individual object. Each state in an object-level dynamic model traces back to one or more script pre- or postconditions. These conditions, in turn, trace to the scripts they describe, and ultimately trace back to goals and objectives. Thus, each state can transitively be traced back to goals and objectives.

The system-level dynamic model is also shown in Figure 2. The purpose of this model is to indicate the flow or potential execution relationships among scripts. Recall that a script can execute when its preconditions, defined by a set of object states, have been satisfied. After a script has executed, its postconditions, defined by a set of object states, are satisfied. States on the system-level dynamic model are defined in terms of script pre- and postconditions. The transitions between states on the system-level dynamic model occur when a particular script has executed. Using this model, it is possible to understand the various ways that scripts may precede or follow one another during system execution. In addition, it is possible to assess completeness by detecting scripts that may never execute because the system never enters a state in which their preconditions are satisfied.

Scripts provide quite a bit of information within the system. In particular, three glossaries of information—Party, Service and Action—are derived directly from scripts. The Party Glossary defines each initiator or participant name in terms of the role that that party plays in the system. In addition to the role definition, each entry for a party includes traces back to the scripts in which the party is referenced. Associated with each trace is an indication as to whether or not the party played the role of an initiator (I), participant (P), or both (I/P). The Services Glossary contains a definition for each service name, based on the script action from which the service is derived. Each service entry includes the list of participants who exhibit the behavior, and traces to all the scripts in which the service was identified. The Action Glossary contains a definition for each action, along with the traces to all the scripts in which the action was identified. Glossaries are created incrementally, and provide a means of normalizing the system vocabulary.

There is also a Logical Properties Glossary for each Party. It contains information necessary to support the definition of the Party's states, as well as the data-capturing requirements. As an example of supporting a state, suppose we are modeling a bar in which a person can only be served liquor if he or she is an adult. The precondition of the bartending script for serving liquor is that the drinker be an adult. As analysts, we ask for the definition of "adult" and are told that an adult is someone age 21 or older. In this context, we can choose to recognize the logical property "age," and use it to define the state of adulthood. Alternatively, the expert may describe a script in which certain information must be captured. In the bar script, the patron has to pay the cost of drinks ordered, that is, the value of the bar bill. This value is referenced in terms of a logical property.

Associated with each glossary entry is a set of traces that indicate scripts from which the entry was derived. This makes it possible to find all scripts that contain a particular party, service or action. Traces

from the Logical Properties Glossary are either back to script pre- or postconditions, or to the scripts themselves. Following the script traces, we can determine the system goals or objectives met by a given artifact.

Object Specifications

We no longer have to guess which objects represent the problem domain. A full specification of these objects is derived from the scripts and glossaries. This is illustrated in Figure 2. An object specification consists of six critical pieces of information. The name of the object corresponds to a party name. The logical properties and services of the object are found in the corresponding glossaries. There are two kinds of services: provided and contracted. Provided services are services that the object allows other objects to invoke. Contracted services are the services that this object requires of other objects in order to carry out its behaviors. The object specification maintains the traces back to the glossaries from which the information was derived.

Initially, the trace for why an object specification exists is to an entry in the Party Glossary. An additional step of OBA re-examines the various object specifications. The purpose is to use several well-known object techniques—factorization, generalization and specialization—in order to reorganize objects. Permitting reorganization complicates the construction of the full traceability network. Object specifications can be created that were not mentioned in any script. For example, a generalization may create a new object that does not occur as a party in any script. Its services and logical properties exist because they were moved from the object specifications that were generalized. In this way, object specifications are created that inherit from other object specifications. Information about the generalization must be recorded with the new object specification in order to justify its existence. Versioning of object specifications is not necessary for such justification.

Regardless of how one tries to formalize the information-capturing process through scripts, there are good reasons for including services and other artifacts solely on the basis that the user or analyst says so. OBA allows the analyst to add user-defined artifacts such as parties or services. User-defined serve two purposes. First, they provide a way to create abstractions of information captured during analysis. And second, they allow the user to capture information at the time the idea occurs, deferring justification. The reason for the addition of these user-defined artifacts must be recorded by the analyst. Any tools that support OBA must be capable of distinguishing user-defined artifacts from those derived more formally.

Summary

Traceability is a very powerful capability. We ask *why* in order to verify that a result is needed and is appropriate. And we ask *why* in order to create a safer context in which to make changes.

Our approach should be compared with analysis approaches in which objects and services are identified by underlining nouns and verbs in a requirements specification. Such approaches typically determine that any object with a physical realization in the problem is an object in the model. However, these approaches are not able to justify why this is so. In fact all nouns may not denote objects in the model. Nor are these approaches capable of identifying intangible objects implied or derived logically. In OBA, one can easily determine the justification by following traces from any artifact back to goals and objectives.

This exposition on the OBA traceability model should make it clear that getting to *why* requires detailed bookkeeping, best provided by tools for information capture and manipulation. This is work in progress at ParcPlace Systems.

Acknowledgments

Our grateful acknowledgments for criticism and ideas go to the OBA tools development team and members of the ParcPlace professional services organization—among them are John Schwartz, David

Pelligrini, Patrick McClaughry, and Vicki Katzman. In addition, we are indebted to the excellent editing of Brian Alexander.

References

- [Harel87] Harel, D., "Statecharts: a Visual Formalism for Complex Systems," *Science of Computer Programming*, North Holland, Vol. 8, No. 3, 1987, pg. 231-274.
- [Rubin92] Rubin, K.S. and A. Goldberg, "Object Behavior Analysis," *Communications of the ACM*, Vol. 35, No. 9, Sept. 1992, pg. 48-62.