An Object-Oriented Framework to Support Architectural Design Development

Doris L. Carver
Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803

David W. Cordes
Department of Computer Science
University of Alabama
Tuscaloosa, AL 35487

Abstract

The architectural design phase represents the phase in the software life cycle where the information acquired in the requirements specification phase is mapped into a high-level design that represents the major structural elements of the system. Interest in object-oriented architectural design paradigms is on the increase. In this paper, we present a requirements specification methodology that analyzes domain information from an object-oriented perspective. The methodology is based on an environment that provides knowledge-based assistance for the generation of a requirements specification model. The model is used to derive a framework for an object-based architectural design. A key feature of the methodology is that it supports traceability of the information contained in the architectural design.

I. Introduction

Object-oriented software development represents a development approach in which the real world problem is bounded by the identification of the objects, their properties, their actions, and the relationships among the objects. The structuring of a system around the real-world objects supports the desirable traits of abstraction and information hiding. It provides a stable foundation for software development and enhances the maintainability of the system due to the localization of the objects, properties and actions. Systems that are developed with this approach tend to be flat instead of hierarchical. Each module denotes an object or class of objects. Object-oriented development builds on the concept of abstract data types. An operation on an object may be classified as a constructor which alters the state of an object, a selector which evaluates the state of an object, or an iterator which permits parts of the object to be visited. Each object may be viewed externally by other objects from its specification or internally from its implementation details. The clear advantage is that the implementation details of an object can be changed without impacting the rest of the system, thereby increasing maintainability. Another major benefit of this approach is that it allows the evolution of the system in terms that are understandable to the user.

This paper focuses on an object-oriented system development methodology. We describe a framework, initiated during the requirements specification phase, that supports the development of an object-oriented architectural design. The architectural design phase is the phase in the software life cycle where the information that is retrieved and analyzed in the requirements specification phase is mapped into a design that contains a blueprint of the major structural elements of the system. This structural design is then refined as the detailed design is developed. Architectural design methods should support object-oriented concepts and modularity by showing interfaces and visibility [34]. Existing architectural design methods include object-oriented design [8], General Object-Oriented Software Development (GOOD) [29], and Object-Oriented Structured Design (OOSD) [34]. Booch's object-oriented design paradigm consists of the identification of objects and their attributes, identification of operations performed on or by the objects, identification of the visibility of the objects in relation to the other objects, solidification of the boundary between the inside and outside view by establishing the interface, and finally the implementation of the object. GOOD uses a conventional dataflow diagram to organize the information in the requirements phase. The dataflow diagram is used to extract the objects used in the architectural design. OOSD is based on the structured design approach. The structure chart notation is expanded to represent object abstractions such as classes, superclasses, and visibility. The effectiveness of each of these paradigms is heavily dependent on the knowledge extracted in the requirements phase.

The mapping from the requirements specification to the architectural design requires a conversion of the "what" to a top level abstraction of the "how". This conversion is generally not a straightforward mapping from the specification to the design. Efforts to provide automated assistance for the conversion process are ongoing [18]. However, heuristic decisions remain a significant part of the conversion process. For an object-oriented architectural design, a conventional structured analysis approach to requirements may not provide the most efficient approach to the requirements specification phase. A requirements specification methodology that analyzes problem information from an object-oriented perspective should provide a stronger mapping to the architectural design than other non-object driven requirements methods.

In this paper, we describe an object-driven requirements methodology. The specification resulting from the application of the methodology can be mapped to a top level object-based modular architecture. In Section II we review existing requirements specification methodologies, both conventional and knowledge-based. In Section III, we describe our object-oriented specification methodology named Knowledge-Based Assistant for Program Specification (KAPS). In Section IV, we provide an example of the resulting specification and discuss its relationship to the initial modular structure of the system. Finally, in Section V, we present a summary of the methodology.
II. Existing Requirements Specification Methodologies

Effective transference of knowledge from the real world to the requirements specification process is a prime factor affecting the quality of a software system. Thus, the requirements analysis and specification phase is a crucial part of the entire software development process. Within the requirements specification phase, the user's needs are identified and analyzed. The result of the analysis is a precise description of the intended behavior of the system. This description is a definition of the functions the system will perform. Many problems that are uncovered at the time of system delivery can be traced to the requirements phase. Functions that are expected by the user but are not identified during requirements definition are often not exposed until acceptance testing or some period of time after system installation. As software development proceeds through the remaining phases, the cost to fix errors increases dramatically.

In a similar manner, the delivered system should contain exactly the functions stipulated in the requirements. Every function that is present in the delivered system should be traceable to its origin. If a function were introduced during requirements specification, in textbook fashion, each function in the delivered system should be linked to the specification. The introduction of functions later in the development process is a more realistic situation. However, the functions should still be traceable to their origin. This situation broadens the scope of a requirements specification methodology that attempts to provide traceability throughout the development process. The methodology should allow for additional information to be added during the development process.

The requirements analysis phase is typically initiated with a document, prepared by the user in narrative format, in which the real world behavior of the system is described. This document becomes the primary data source for the initial transformation of real world system information into the software development process. Research efforts that address the initial phase of the information transfer process are increasing. Benefits of improved techniques in this area include enhanced communication with the user, broader understanding of the real world problem, and the production of accurate system documentation. In addition to these benefits, a major benefit of successful research efforts in this phase is the potential of identifying techniques that support not only the requirements phase but also form a framework to support the ensuing phases of the software development process. Research efforts that result in paradigms that enhance traceability throughout the software development process hold potential to significantly contribute specifically to the advancement of the requirements specification process and generally to the advancement of the entire software development process.

Conventional Specification Techniques

Existing techniques for defining the functional requirements of the system are diverse. Numerous conventional approaches to represent requirements analysis information exist. These approaches include formalized methods to represent both static and dynamic system behavior. The representation techniques for static behavior include regular expressions, algebraic axioms, and recurrence relations. Common methods to represent the dynamic description of systems include state-oriented techniques such as decision tables, transition diagrams, event tables and Petri nets [11].

Other conventional techniques used to represent information from the requirements analysis process are based on data flow and data structure models. The data flow approach places emphasis on flow of the information rather than on flow of control in the system. The most commonly used representation scheme for the data flow approach is the data flow diagram (DFD). A DFD is a graphical representation of the information flow and the transformations performed on the information from the input to the output. Data structure representation techniques are based on the assumption that the information has a hierarchical structure. Common representation schemes include War- nier diagrams [33], HIPO charts [17], and structure charts. A combination of data flow and data structure is typically used to represent the requirements information.

The benefits of automated support for the requirements process have precipitated efforts to provide such support for these conventional approaches. The PSL/PSA system was one of the first such systems to have automated support [31]. The PSL (Problem Statement Language) is based on a model of objects and their properties. The model uses a pre-defined syntax to describe information flow, system structure, data structure, data derivation, system size, and system properties. The PSA (Problem Statement Analyzer) analyzes the PSL input and produces analysis, data-base, reference and summary reports.

A second major effort toward automated support is found in SREM (Systems Requirements Engineering Methodology)[1]-[2]. It is based on a finite-state modeling approach where each state represents a unique phase of execution within the system. It provides facilities for incorporation of time elements.

A third system that has automated support is the SADT (Structured Analysis Design Technique) [26] - [27]. It has a data flow basis and uses a graphical representation of actigrams and datagrams that can be decomposed as the specification and design process evolves. Numerous additional systems with automated support are available. A comparative study of conventional specification techniques is found in [10].

The requirements methodologies such as those just described lack the structure and formality necessary to allow mathematical analysis to support the production of a complete, unambiguous, noncontradictory, requirements specification. A specification written in a formal specification language provides these benefits. Numerous formal specification languages exist including Gist [5], Ina Jo [28], PAISley [36], and Larch [16]. The technical nature of formal specifications precludes their use as a viable communication interface with the user. They also do not possess the mechanisms found in many informal techniques for decomposing the real world problem. As a result, the use of formal specifications is typically preceded by the application of an informal methodology.

Knowledge-Based Specification Techniques

As the field of requirements specification matures, many of the above techniques are being used as a basis for the application of knowledge-based systems to the requirements specification phase. The global goal of automatic programming, which is the automatic generation of a system based on stated requirements, is a futuristic goal of software engineering. Much of the ongoing work towards this goal has its basis in artificial intelligence. Early work in automatic programming is found in [5] and [6]. In [6],

350
the automatic implementation of abstract algorithms is discussed and in [7] application of domain specific knowledge to the automatic programming process in the $\Phi$-mix project is described. Balzer's early work in automatic programming dealt with transformation of informal specifications to formal specifications [3] - [4]. The dually difficult problems of transformation from informal to high-level formal specification and high-level formal to low-level formal were divided. The formal specification language Gist [30] was developed for use as a formal input to the process of automatic transformation to low-level form. The continuing research on a framework for the development and evolution of CommonLisp systems is discussed in [5].

The "divide and conquer" approach has been followed by most researchers. Some researchers are working on the automatic generation of code from a specification [5], [12], [15], while other researchers are working on the formalization of information extracted from an informal specification [5], [9], [14], [19], [25], [32]. The need for informality in specifications is described in [4] and the inherent problems with that informality are discussed in [21].

In [25], the Requirements Apprentice, a part of the Programmer's Apprentice, is presented. The goal of the Programmer's Apprentice is to provide intelligent assistance to all aspects of the programming process. The primitive element of the Apprentice is the cliché, which is a combination of familiar elements in a domain. A cliché is made up of roles and constraints on the roles. A key component of the cliché concept is the relationship among the clichés. The formal representation, called the Plan Calculus, is composed of a graphical notation and formal semantics for representations. The Requirements Apprentice addresses the transformation of an informal system description to a formal system description. The target user for the system is the analyst, not the end user.

Gist is a formal specification language developed to emulate the natural language construction of specifications [5]. It is a textual language based on object types, operations, and relationships of the object types. Gist has a syntax for expressing constraints on the behavior of the objects. A Gist specification is an operational specification that permits formal analysis. In order to enhance the readability of the Gist specification, a paraphraser recreates a natural language description of the specification to complement the validity checking process.

The knowledge-based system Kate [14] provides assistance for the transformation of informal requirements to a formal specification. It is a domain-specific analysis techniques utilizing the Glitter [13] system to represent information needed to refine a specification. The refinements of informal requirements are criticized by using domain knowledge and usage scenarios. In [14], a description of a critic for a formal specification is given. It is based on a model which represents policy issues in a specific domain.

Lubars [19] and Tsai [32] present systems based on dataflow input representations. Lubars describes and intelligent environment, IDEA, for the specification and design phases of software development. IDEA uses the dataflow diagram as its unifying representation model. It refines the dataflow specification based on refinement rules in the knowledge base. User input is required in the refinement process. The concept of reusability is addressed by an abstract graph representation of designs that can be selected to produce a specific design [20].

Tsai [32] describes the Specification Transformation Expert System, STES. It provides intelligent assistance for the requirements phase by generating a structure chart design from a dataflow specification input. Heuristic rules expressing desirable design features such as coupling, cohesion, fan-in, and fan-out are contained in the knowledge base. These rules, coupled with designer interaction, are used to refine the specification into a design in a structure chart representation.

Each of these systems addresses some aspect of the requirements specification process. The Requirements Apprentice, Gist, and Kate have a common goal to develop a formal specification from various types of informal input. IDEA and STES represent knowledge-based systems that add formalism to the specification process but do not actually produce a formal specification. IDEA refines an informal DFD description and STES transforms a DFD description to a structure chart representation.

This paper describes our work towards the development of an environment that provides knowledge-based assistance to the requirements phase from an object-oriented perspective. The goal is to add formalism to the specification process. It is not our goal to produce a formal specification. Benefits of capturing the requirements from an object-oriented perspective include the bounding of the primary system entities, the representation of the system in a manner that is amenable to communication with the user, and the development of an object-oriented framework to support architectural design.

III. KAPS: Knowledge-Based Assistant for Program Specifications

KAPS is a knowledge-based assistant that is designed to enlarge the user's role in the requirements analysis process in order to provide improved communication between the user and the specifier. The system accepts as input the user's natural language description of the system behavior. It requires a grammatically correct document containing no pronouns. It performs a syntactic parse of the document using interactive resolution to assist in parsing syntactically ambiguous sentences. The parser is based on augmented transition networks (ATNs). The on-line dictionary is a key component of this process. Each word in the dictionary has the word type, possible word ending and other word features associated with it. The word type defines the grammatical class to which the word belongs. The ATN recognizes adjectives, adverbs, determinants, modals, negations, nouns, prepositions, pre-verbs, conjunctions, disjunctions, and verbs. This parser relies heavily on user interaction to resolve parsing decisions that cannot be made directly by the parser. The dictionary is extensible by an interactive feature that allows the addition of word definitions. The requirements document used for the examples in this paper is given in Figure 1 [24].

The output of the parse module is produced on a sentence by sentence basis. It is a standard LISP S-expression which contains information regarding the parsed sentence. This information includes the basic sentence type (transitive, intransitive or linking) and information about the objects and actions found in the sentence. A detailed description of the parsing process is found in [9].

Using the parsed sentence structures, a set of domain specific facts is asserted to the knowledge base. Initially, each sentence is asserted as a fact. Based on the sentence fact, each object and action in the system is assigned a unique identification number that is used when the object
or action is used in other facts. Specific formats for events and properties are defined. The event format contains information about the actor, action, object, and recipient. The properties are subdivided into simple, object, and event. Simple properties contain single word modifiers. Object properties are properties that are expressed in terms of other objects. Event properties pertain to events that the object initiates. The specific attributes of these property formats vary; however, each format contains an attribute for expressing object restrictions such as time dependency.

In addition, the identification number associated with the formats of other objects. Event properties pertain to events that the fact. The conversion algorithm used to convert the sentence structures to the facts is found in [9]. Partial output for the sentence "All aircraft must have a transponder" is given in Figure 2.

A problem encountered in this conversion is the problem of aliasing. For example, if the document refers to a payroll system and then later refers to the system, we are unable to guarantee correct recognition of the dual reference to the same object. We make heuristic decisions about the resolution of such occurrences and verify the correctness of the decision interactively with the user. Recognizing the limitations, the application of the conversion algorithm results in a set of facts which contains the domain-specific knowledge about the objects, properties, and actions found in the original document. These facts also contain the information that provides a traceable path from the information in each fact to its origin in the original document.

These domain-specific facts are used to perform an analysis of the document, regenerate a document for the user, and produce a technical specification model. The document analysis includes partial checks for internal completeness, consistency, and necessity. Completeness warnings are produced for objects that have no properties, events with no actor, and actor/action pairs with no events. Consistency is analyzed by listing all properties of each object and interactively asking the specifier for consistency checks. Necessity is addressed by the identification of objects that do not participate in any system objects or have little or no interaction with other objects.

KAPS currently produces an evaluation report containing the error analysis as well as a cross reference by objects and actions. A portion of the report for the transponder requirements is shown in Figure 3. KAPS also produces a regenerated object-oriented document containing a structured summary of objects, relationships, and properties. These documents are intended to provide the user with a mechanism to validate the information from which his system will be developed.

IV. KAPS Specification Models

In addition to the document analysis, a specification model for each object is created using a detailed algorithm. The model is represented in an Ada package-like format containing actions, recipients of the actions, and properties of each object. Details of the model generation process are found in [9]. The model generation process includes the identification of solution objects based on their role in the domain. For each solution object identified, the definitions of active or passive status, properties, events, derived objects, and internal objects are determined. This model information is derived from the information stored in the knowledge base. The information is traceable to its source by the use of a traceability id slot in the frame. A sample module specification generated by KAPS for the transponder problem is given in Figure 4. In these modules, each object is identified as an abstract data type or an abstract state machine. An abstract data type represents a passive object and an abstract state machine represents an active object. Objects that must be visible to a given object are contained in the requires clause. Objects that require separate modeling because they initiate actions but that are visible to only a single object are represented in the contains clause. Each attribute of an object is listed in the property clause. Objects that are abstract data types that are not required of more than one object are shown as an internal object. Finally, actions of each object are shown in the model as procedures.

Using these specification models as the representation of the real world problem from an object viewpoint, we can extract a high-level representation of the modular architecture. We use a graphical representation to express the object topology of the system. Rectangles are used to represent objects, labeled directed lines represent actions initiated by or to objects, and nested rectangles represent objects contained within other objects. Using the models in Figure 4, we derive the architectural design shown in Figure 5. The object controller queries the objects ground station monitor and aircraft. Ground station monitor checks and maintains the database and queries the aircraft. Ground station monitor contains an internal object graphic display which it generates and updates. Database maintains flight information which is an internal object. The object aircraft contains a transponder. The transponder transmits aircraft position which is an object that is internal to transponder.
the transponder transmits aircraft position to the ground station monitor
the monitor can query an aircraft for flight information
the monitor keeps a database that maintains this information
a graphics display is generated from the current information
the ground station monitor updates the graphics display frequently
the monitor checks for dangerous situations
the controllers may query the monitor for additional flight information
controllers may also query the aircraft for this information

Diagnostic Messages
ERROR Messages
Missing actor for action generated in sentence s_005
The property below is defined for 'ground station monitor'
but there is no other reference to the corresponding action 'check'
-> ground station monitor checks for dangerous [two or more aircraft in the same air space] situations

WARNING Messages
The action/object pair 'query - aircraft' has multiple actors defined
in sentence s_003 the actor of the pair is ground station monitor
in sentence s_009 the actor of the pair is controller

No properties defined for: aircraft position
No properties defined for: controller

No hierarchy relationships defined for: aircraft
No hierarchy relationships defined for: aircraft position

Aircraft has properties defined but participates in no operations

Cross Reference Listing
Listing By Sentence
...
Sentence: s_003 the monitor can query an aircraft for flight information
Objects used:
aircraft
object of action query
flight information
restriction to action query
ground station monitor
actor of action query

Actions used:
query

Listing by Object
...
ground station monitor
Operations include:
s_003: actor of action query
s_004: actor of action keep
s_006: actor of action update
s_008: object of action query
Attributes defined in:
s_007: event property defined for this entity
No hierarchy ordering defined
Visible to operations/objects in:
s_002: restriction to action transmit

Listing By Action
query
Used in sentences: s_003 s_008 s_009
### Figure 4
Specification Model

<table>
<thead>
<tr>
<th>package</th>
<th>aircraft model</th>
<th>Abstract Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>requires</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>contains</td>
<td>transponder</td>
</tr>
<tr>
<td>property</td>
<td>transponder:object is separate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- all aircraft has transponder</td>
<td></td>
</tr>
<tr>
<td>property</td>
<td>position indicator:object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- all aircraft has position indicator</td>
<td></td>
</tr>
<tr>
<td>property</td>
<td>radio:object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- all aircraft has radio</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>aircraft</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>package-part</th>
<th>transponder -- belongs to aircraft model</th>
<th>Abstract State Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>requires</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>contains</td>
<td></td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td>transmit [aircraft position:internal]</td>
</tr>
<tr>
<td></td>
<td>-- transponder transmit aircraft position to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- ground station monitor aircraft position</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>internal transponder</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>package</th>
<th>controller model</th>
<th>Abstract State Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>requires</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>contains</td>
<td></td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td>query [ground station monitor:external]</td>
</tr>
<tr>
<td></td>
<td>-- controller query ground station monitor for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- additional [data not normally displayed by</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- the monitor] flight information</td>
<td></td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td>query [aircraft:external]</td>
</tr>
<tr>
<td></td>
<td>-- controller query aircraft also for flight information</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>controller</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>package</th>
<th>database model</th>
<th>Abstract State Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>requires</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>contains</td>
<td></td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td>maintain [flight information:internal]</td>
</tr>
<tr>
<td></td>
<td>-- database maintain flight information</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>database</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>package</th>
<th>ground station monitor model</th>
<th>Abstract State Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>requires</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>contains</td>
<td></td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td>query [aircraft:external]</td>
</tr>
<tr>
<td></td>
<td>-- ground station monitor query aircraft for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- flight information</td>
<td></td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td>keep [database:external]</td>
</tr>
<tr>
<td></td>
<td>-- ground station monitor keep database that</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- maintains flight information</td>
<td></td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td>generate [graphic display:internal]</td>
</tr>
<tr>
<td></td>
<td>-- ground station monitor generate graphic display</td>
<td></td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td>update [graphic display:internal]</td>
</tr>
<tr>
<td></td>
<td>-- ground station monitor update graphic display</td>
<td></td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td>-- frequently [10 to 12 seconds]</td>
</tr>
<tr>
<td></td>
<td>procedure</td>
<td>check [database:external]</td>
</tr>
<tr>
<td></td>
<td>-- ground station monitor check for dangerous</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- [two or more aircraft in the same air space]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- ground station monitor check database regularly</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- [every 2 seconds]</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>ground station monitor</td>
<td></td>
</tr>
</tbody>
</table>

| 354 |
This graphical representation serves as a first-pass at the topological structure of the system. This architectural design is designed to provide the system designer with an initial blueprint of the objects and their interactions with each other as expressed in the requirements. As the system design evolves and new objects are introduced to refine these high-level objects, this structure must be modified accordingly. In addition, as the design evolves, the implementation language becomes a factor in the refinement of the design. However, at this level, this representation can serve as the basis for a system that will have an Ada implementation or it can serve as the basis for a system that will be implemented in an object-oriented language such as C++ [35], Flavors [23], or Eiffel [22]. A key feature of this methodology is that the information contained in the specification model, and thus in the graphical representation, is traceable to its point of origin.

V. Summary

The application of an effective requirements analysis paradigm is a prerequisite for the development of a viable architectural design. In particular, an object-oriented design should benefit from a requirements analysis method that provides an analysis of the application domain from the perspective of the identification of objects, actions, and relationships among the objects. In this paper, we presented an object-oriented analysis technique that produces a specification in a representation format that is designed to allow for ease of conversion to an architectural system design. The methodology uses as input a preprocessed requirements document in a natural language format. This document is submitted to a series of routines that parse the sentences, convert the information to facts stored in a knowledge base, and produce a regenerated document.
This document, organized by object, is for the primary purpose of providing a communication tool with the user. After validation of the content of the document by the user, the system processes the information in the knowledge base in order to derive the object-oriented specification. Numerous decisions are made in order to select the objects that require separate modeling. These decisions are made based on the actions and visibilities surrounding each object. For each object that is identified as requiring a model, the system produces the model derived objects, and internal objects. From this representation scheme, a high-level architectural design is derived. A key feature of this paradigm is that it provides for traceability of the decisions that produced it are traceable to their origins.

REFERENCES


