The Domain Theory for Requirements Engineering

Alistair Sutcliffe, *Member*, *IEEE*, and Neil Maiden, *Member*, *IEEE*

Abstract—Retrieval, validation, and explanation tools are described for cooperative assistance during requirements engineering and are illustrated by a library system case study. Generic models of applications are reused as templates for modeling and critiquing requirements for new applications. The validation tools depend on a matching process which takes facts describing a new application and retrieves the appropriate generic model from the system library. The algorithms of the matcher, which implement a computational theory of analogical structure matching, are described. A theory of domain knowledge is proposed to define the semantics and composition of generic domain models in the context of requirements engineering. A modeling language and a library of models arranged in families of classes are described. The models represent the basic transaction processing or 'use case' for a class of applications. Critical difference rules are given to distinguish between families and hierarchical levels. Related work and future directions of the domain theory are discussed.

Index Terms—Requirements engineering, domain modeling, software reuse, analogical reasoning, CASE tools.

1 Introduction

Requirements engineering is a complex, error prone process [57] which could benefit from intelligent support for capturing and validating specifications. Cooperative critics have been proposed by Fischer as one approach to validation by engaging designers in problem exploration [17], [18], whereas others have suggested more automated software design critics [63]. Intelligent tools for assisting the requirements engineer can point out flaws in specifications [52], [37] and domain-oriented design environments can help in suggesting improvements to specifications [17]; however, all intelligent validation/critiquer tools depend on embedding considerable domain knowledge within the tool.

The role of domain knowledge in software engineering has been recognized as a critical factor in obtaining complete, consistent, and accurate requirements. Unfortunately, acquisition of domain knowledge is a time consuming and expensive process [50] which becomes a bottleneck that may prevent widespread application of domain-specific intelligent design assistants. Such tools are inevitably limited by the necessity of building a domain knowledge base de novo for each application area.

An alternative approach is to use generic models of applications, for instance the Requirements Apprentice [55] proposed clichés or reusable abstractions of domain classes. This theme has been adopted by others in the form of Generalized application frames [11], reusable patterns [22], and design architectures [63]. Generic models of applications based on analogical structure matching theory [19] were proposed by Maiden and Sutcliffe [39], who demonstrated experimentally that reuse of such models can improve requirements engineering [66]. These models extend the concept of generic knowledge beyond that afforded by domain-oriented design environments to raise the possibility of wider ranging, interdomain reuse through which many application domains may be realized.

Knowledge can be reused between dissimilar domains. (e.g., car hire and library loans) if an appropriate abstract model (e.g., loaning resources) can be transferred. Cognitive [6] and computational models [21] of analogical reasoning describe the processes by which people recognize abstractions, similarities, and differences between domains. In software engineering, analogies are intuitively recognized and used by experts when they come across familiar problems; moreover, evidence from cognitive studies of programming shows that experts develop "deep structures" of similar problem domains [20], [48]. This raises the question as to whether a finite set of such abstractions may exist for software engineering problems and, furthermore, at what level of abstraction generic models of domain knowledge should be described.

In software reuse, domain analysis follows a pragmatic approach to exhaustively analyze all the applications within an organization's field of operation [50]. Clearly some more theoretically principled answers are required. Domain theories have been proposed in different senses, such as the abstract computational theories of domains [63], [64], and design architectures [62]. However, our focus is on problem rather than on solution models. We intend to describe domains in meaningful terms to facilitate communication between users and requirements engineers.

The paper is organized in five sections. Section 1, the utility of using generic models in the software critiquing is demonstrated with a case study scenario. Section 2 describes the matcher-retrieval tool, AIR (Assistant for Intelligent Reuse), which is based on analogical structure matching

• *A. Sutcliffe and N. Maiden are with the Centre for HCI Design, School of Informatics, City University, Northampton Square, London EC1V OHB. E-mail: {a.g.sutcliffe, n.a.m.maiden}@city.ac.uk.*

Manuscript received 5 May 1994; revised 17 July 1996.
Recommended for acceptance by L. Clarke.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 101154.

theory. Section 3 reviews the knowledge base of domain models used by the matcher and the theory upon which they are based. The paper concludes with a review of related work and future prospects for this approach.

2 Use of Domain Knowledge in Requirements Engineering

This section describes a requirements engineering example in a library domain from the viewpoint of the user system dialogue, as well as illustrating how the matcher helps the requirement engineer understand, validate, and elaborate a developing requirements specification [41].

The system uses a library of generic models to either critique a new requirements specification or to act as templates for its development. This highlights the need for a computational matching process which can take a small set of input facts describing a new application and match them to appropriate models in the system library. This approach is based on the cognitive theory of analogy [19], [27] which asserts that problems of a certain class share a deep structure, i.e., a common abstraction.

Structure matching theory proposes that people recognize and remember high-level abstractions for a set of related problems; for instance, an abstract model of a satellite revolving around a central object controlled by centripetal forces maps to the concrete domains of the sun and planets in the solar system and electrons orbiting the nucleus of an atom [19]. Computational matching processes have been created for ad hoc analogies [27], but these rely on exhaustive pairwise matching between facts in two models. This approach becomes computationally intractable as the number of potentially matchable models increases. The challenge is to produce a matching algorithm to detect different applications which share the same abstraction over many different families of analogies (e.g., loan type abstractions, logistics, inventory, etc.).

2.1 The Domain Matcher

The domain matcher process matches a limited set of input facts describing a new application with a library of domain abstractions, termed Object System Models (OSMs). It determines the goodness of fit between the input facts and one or more domain models stored in its knowledge base.

2.1.1 Structure of the Knowledge Base

Object system models are a network of cooperating objects that serve some purpose, similar to the "use case" concept of Jacobson [29] or reusable patterns espoused by Wirfs-Brock et al. [71] and Coad and Yourdon [10]. A detailed description of the OSM library and the modeling language is given in Section 3. The models are equivalent to a class hierarchy in any object-oriented language, but abstraction is at the level of a group of cooperating objects rather than isolated classes.

OSMs are specialized at each level by addition of different knowledge types. Structural facts (e.g., objects and their functional relationships) are the most important constructs for discriminating between models at higher levels in the hierarchy. The knowledge types used for discriminating between classes, in descending order of specialization, are:

- state transitions with respect to structural facts
- activities which are needed to attain goal states
- relationships between structural facts and state transitions
- goal states
- events and stative conditions
- object properties
- activities in the object system
- labels which are linguistic expressions of action instantiating state transitions

An important model concept which forms part of the structural facts is structure objects which represent approximation to the physical domain and express containment or possession of other objects, e.g., a library may be modeled as a structure object which contains books. Further details of the modeling language are given in Section 3 when the library is described. The structure of the OSM hierarchy is illustrated in Fig. 1.

Models in the first and second levels are defined by state transitions, objects and relationships which provide the basic analogical structure for matching. Goal states, events, and object properties are useful discriminators at lower levels.

2.2 Validating Requirements by Matching to Generic Models

The system provides domain knowledge in the form of generic models to guide requirements specification and to

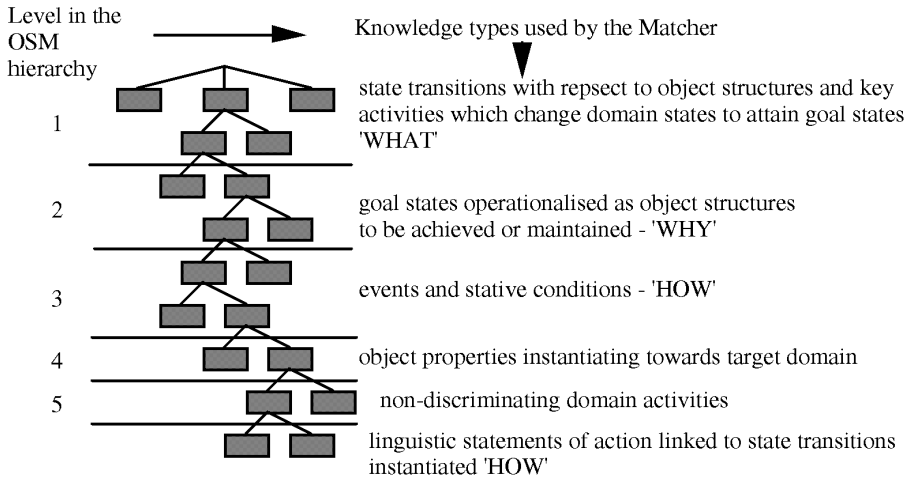


Fig. 1. Hierarchical structure of object system models with the knowledge types used for matching.

enable critiquing/explanation. To illustrate the system in action we describe a typical session with the toolset. A library scenario is used to demonstrate active guidance and critiquing during requirements engineering. The critic explains retrieved domain abstractions to the requirements engineer to assist development of a specification for computerized stock control facilities in a library. The dialogue controller manages a mixed initiative dialogue to provide effective intervention and guidance [41]. The dialogue controller has configurable rules to trigger system initiative so explanation and critiquing by the system can be customized to users' preferences. Fact capture is via a form filling dialogue. The system maintains a lexicon of fact types which is extensible. This is used in combination with the knowledge representation ontology, described in Section 3, to classify facts by type as they are entered. If the system cannot find a suitable classification it prompts the user to assign the new fact within the structure of its lexicon displayed as a menu hierarchy. Alternatively, the lexical checking may be turned off, so the system will accept any identifier the user enters for a specific fact type. This enables the system to be independent of domain specific detail.

The requirements problem. The requirements engineer has conducted an initial interview which revealed the need for a stock control system. The system must identify missing or damaged books which are no longer in the library. It must also permit a stock-take of books within the library and ensure that books cannot be purchased without the librarian's authorization. The scenario has three critiquing cycles which uses three OSM models, Object Containment (OC), Object Returning (OR), and Object Supply (OS) in descending order of specialization.

Fact capture and matching in cycle-1. First the requirements capturer acquires facts to enable retrieval of a high-level OSM. Active guidance in the fact capture dialogue ensures that fact types necessary for the system's matching strategy, e.g., state transitions and Structure objects, are entered. The system prompts the user to enter facts in fields labeled with the fact types it expects, e.g., object, state transitions, events, as illustrated in Fig. 2. The dialogue illustrated shows the lexical checking options turned off so the user may enter any identifier he/she chooses in the fact-type input field. Text explanation helps the user decide where facts should be entered. Once the system has acquired a minimal fact set, in this case the objects library, book, and borrower with the transition deliver, the first round of matching is initiated. The system captures state transitions by reference to the objects, e.g., deliver is communicated to the system by the user linking the transition from the library to the borrower. In this manner the system does not need to maintain a valid lexicon of the state transitions. The system searches the top level families of models in its library and retrieves the closest match which is the OC object system model. In the example the user has entered facts which are incomplete and inconsistent, however, the power of the domain matcher ensures that it retrieves the OC model as the best fit for stock control.

Explanation in cycle-1. The requirements critic explains the retrieved model to improve the user's understanding. The retrieved OC model is explained using visualization and text-based descriptions of the abstraction with its mappings to the current application, as illustrated in Fig. 3a. The mappings show the user entered terms for the objects (book, library, and borrower) with the types the system has used to classify these objects (resource, container, and sink, respectively). The problem classifier has detected one problem illustrated on the notepad, see Fig. 3b. This is caused by entry of the object, a borrower of books, which does not fit the OSMs available at this level of abstraction. The Matcher flags this fact as an inconsistent categorization, but allows the dialogue to continue and stores the extra fact for future matches. The requirement engineer is prompted to confirm or reject the current match. In the scenario the user confirms the match, and the system then prompts the user to enter further facts.

Model selection and explanation in cycle-2. The user enters further facts describing a supplier structure object and a state transition representing book delivery from the supplier. This allows the matcher to search further down the OC family and retrieve the OS (object supplying) subclass. The dialogue controller selects active guidance to present a set of windows in an ordered sequence, illustrated in Fig. 4a, 4b, and 4c. The critic explains the retrieved model using informal graphics supported by text-based descriptions to draw attention to fundamental differences between the input model and the OSM structure and behavior (see Fig. 4a), with an example as shown in Fig. 4b. The other OSM (OR object returning), which was not selected in the OC family, is also explained to help the user understand the system's rationale for discriminating between models, see Fig. 4c and 4d. Gradual exposure to the candidate abstractions and prototypical examples is controlled by the dialogue manager. The result of this bout is an extended, but still incomplete requirements specification. The user can request prototypical examples of the object supply model to supplement earlier explanations and browse the list of detected problems including incompleteness, inconsistencies, and ambiguities on the problem notepad.

Critiquing in cycle-3. The users enters further facts describing a relationship between the supplier and the customer, but has not identified that the customer in this application is in fact the library. The dialogue controller selects active critiquing because the problem classifier detects an inconsistent fact that it cannot reconcile with either the current OSM or with any subclass models, see Fig. 5a and 5b. Reconciling this problem is achieved either by deleting the fact or by indicating that the customer and library objects are the same. The user chooses the latter option. Further requirements acquisition of activities is followed by explaining associated information system models, such as circulation control reports for functional requirements. Guided explanation of information systems with prototypical examples is given using text descriptions, diagrams, and animation, as shown in Fig. 5c. The requirements engineer can request critiquing at any time by rematching the requirement specification, hence iterative explanation and critiquing occurs until the user is satisfied.

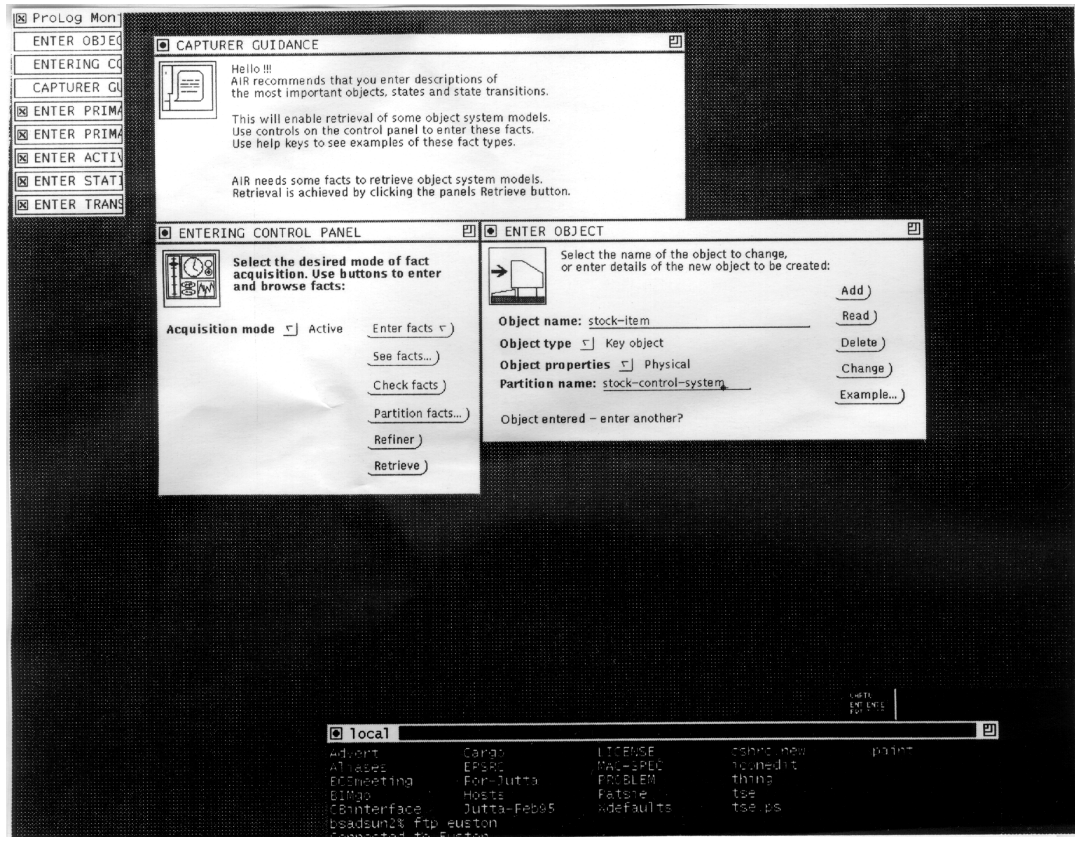


Fig. 2. Fact capture screen in early part of the dialogue, illustrating the dialogue controller and fact entry.

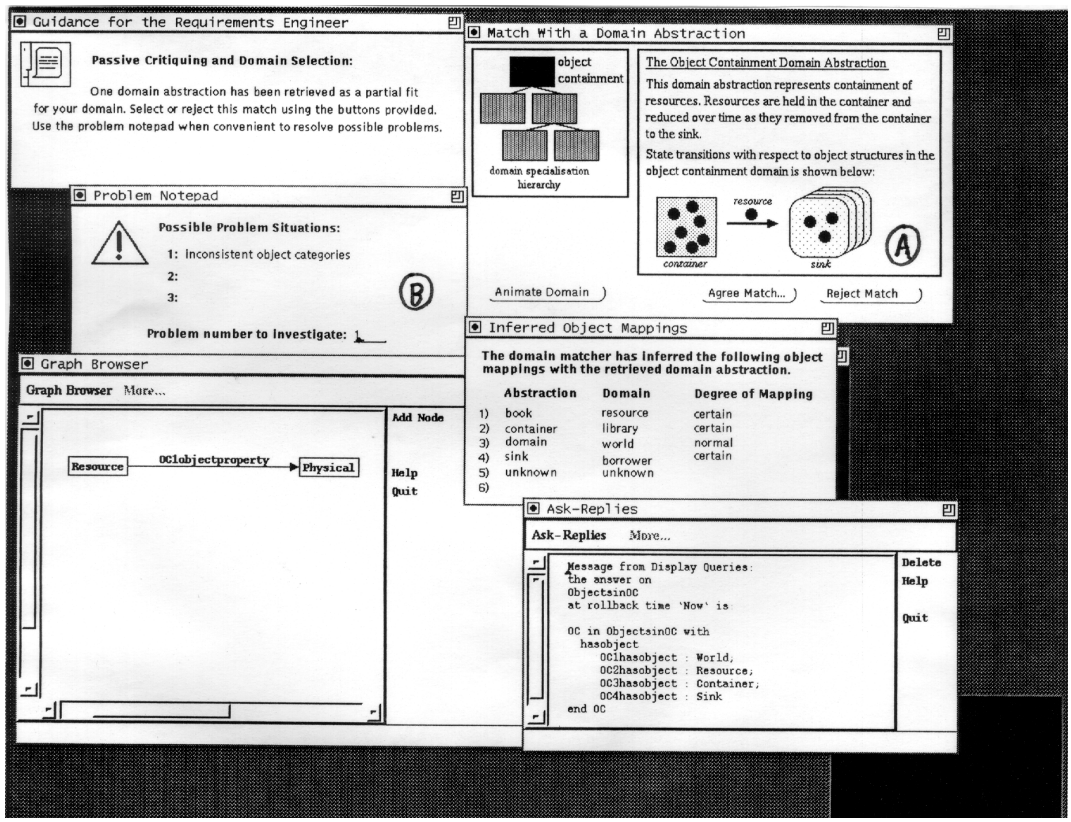


Fig. 3. Illustration of the first cycle matcher dialogue showing windows for (a) explaining the retrieved generic model and (b) the problem notepad.

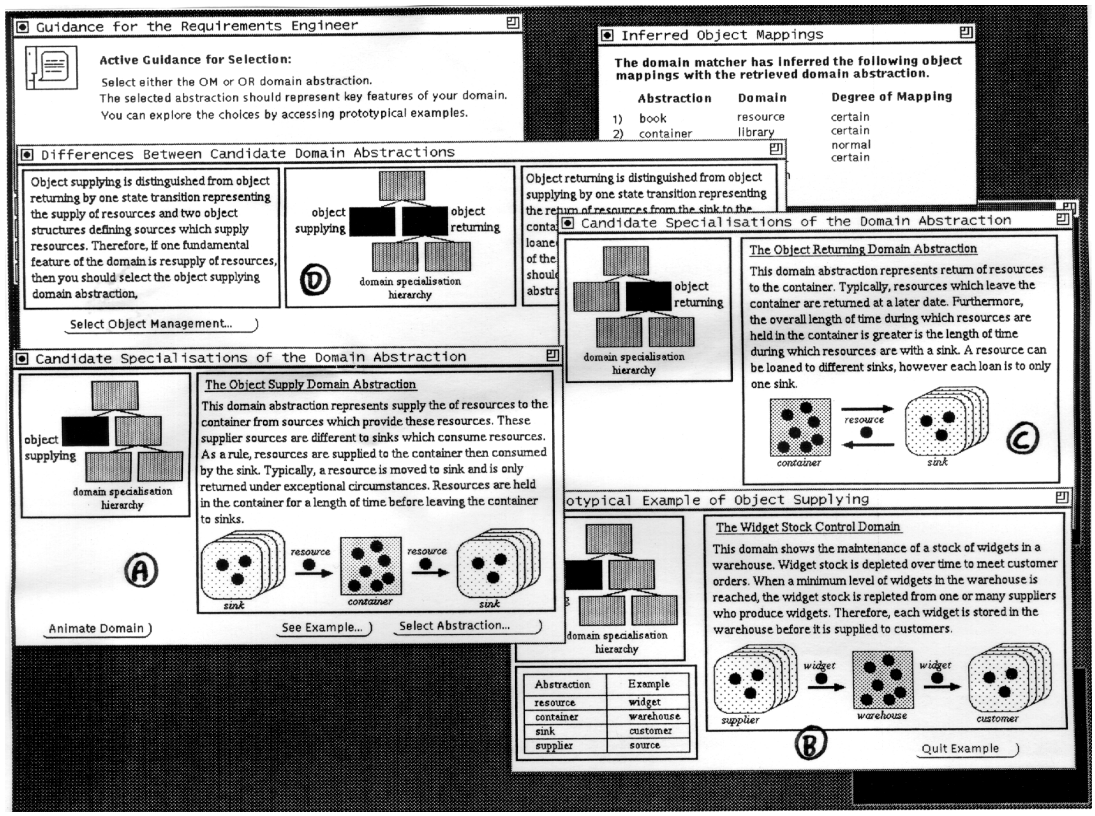


Fig. 4. Explanation dialogue (a, b) for the retrieved “object supply” model after further fact entry with details of the system rationale for selecting the object supply model (c, d).

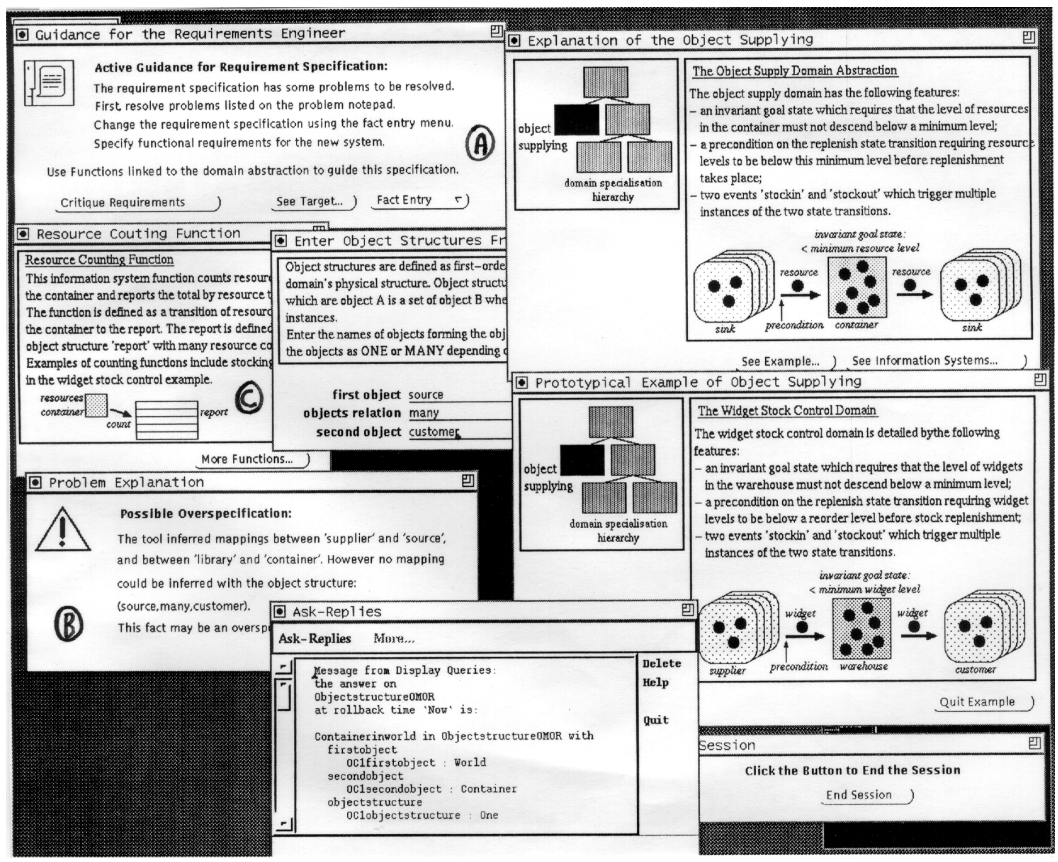


Fig. 5. Critiquer dialogue with active guidance for explaining inconsistent facts (a, b), with further explanation later in the dialogue of an information system model (c).

The user can then request a second pass to match facts, so far unused, e.g., the role of the borrower. To model the requirements for book repair, an object property is entered to subtype the key object as augmented when the dialogue is at the early stage having identified the OC family of models. This causes the matcher to retrieve the OR model rather than the OS which was retrieved when the supplier structure object was entered. If circulation control for book stock were required and a return transition delivering a book from the borrower to the library were entered, this would lead to retrieval of the OH (object hiring) OSM to describe borrowing type applications. In this case the matching is between an owning structure object, the library, a client (the borrower), a resource (a book), and two state transitions one for the outbound loan (from owner to borrower) and the return in the opposite direction. In this manner the Matcher can help refine requirements for large scale applications which are composed of several OSMs.

For further details of the heuristics, guiding strategies and the critiquing/validation dialogue, the reader is referred to Maiden and Sutcliffe [41]. Successful matching is dependent on a system knowledge base of OSM models covering a range of applications which the user may encounter. In the next section we describe the functionality of the tools for matching and critiquing and explain the mechanism of the analogical structure matching upon which they are based.

2.3 Example of the Matching Process

This example demonstrates the matcher process following the scenario of requirements specification for stock control in a library (for further details see [32]).

After initial entry of facts about the target domain, structure matching is used to map the input model to a high-level object system class. First local mappings calculate the matching feasibility with each candidate high-level OSM class (see Section 3 for details of these classes). Local mapping is a simple filter using type checking to ensure that the input model and a candidate OSM share a number of facts types above a preset, but configurable, threshold. This eliminates unnecessary matches with unsuitable OSM families. The OC model contains six facts which can be mapped to the target model (four structure object, one key object and one state transition). Other high-level OSMs do not attain the same score so they are rejected.

The structure matcher detects common connections between state transitions, structure object and object subtypes in the OC system and the input model, as illustrated in Fig. 6. Mappings are indicated by equivalent positions in the two figures.

The structure mapper has to determine whether facts in the input model have the same neighboring fact types in the corresponding location within the OSM. Each input fact is considered in turn. The mappings are quantified as the goodness of fit between the OSM and the input model for every component and their respective neighbors. The structure mapper maps each state transition and structure object in the OSM and input model; for instance, the structure <*world-one-container*> is mapped to the input structure <*domain-one-library*> according to a shared type and mandatory one-to-one relationship; see Fig. 6.

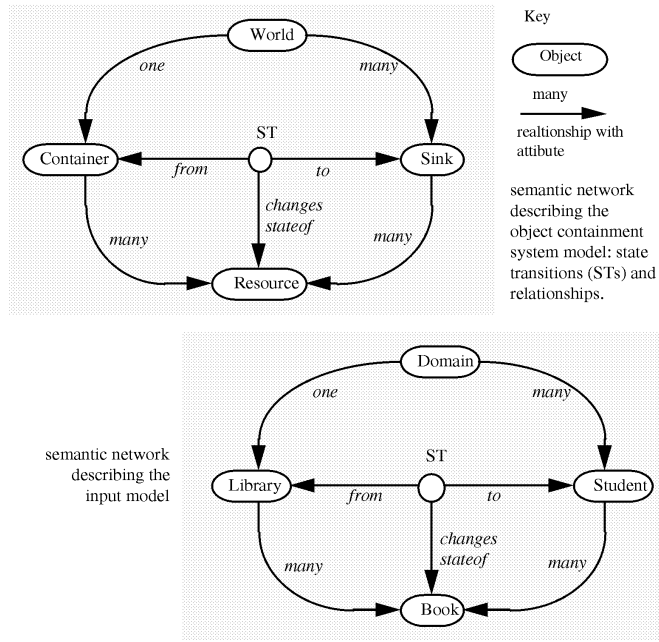


Fig. 6. Matching between input facts and the OC system model.

The structure mapper selects the best-fit from a set of possible mappings between the input model and a target OSM. Three structures in the input model can map to the structure<*container-many-resource*>; as depicted in Fig. 7. The *container* object is associated with two other components and a state transition. The structure <*library-many-book*> is the best fit with the Object System Model because it is connected to four compatible neighbors (i.e., the facts have same types: container = library, resource = book, same cardinality = many and a shared state transition); while the other two candidate structures have only two compatible neighbors. This matching process is repeated for each state transition and structure object within the Object System Model.

The best-fit mapping is calculated from the total score of component-pair mappings between the models. For example, the objects *library* and *container* score 3, 4, 4 and 4, focusing on world, container, resource and the state transition and counting the number of shared nearest neighbor objects with the same type. Hence container and library get a score of 4 because they share domain = world, book = resource, the state transition linking book and library and finally library and container have the same type (Structure objects). The total score accounting for the dyadic permutations of the object-pair <library, book = container, resource> mapping is 15, which is greater than for any other object combined with *container*.

The structure matcher scores the degree of similarity between the target and system models. In the first pass, the matcher visits each OSM family in turn and tests each input fact successively with its neighbors to calculate a goodness of fit score. It then ranks the OSMs and selects the family with the highest score, in this case the OC model. The match controller records the current match then attempts to specialize it to each OC subclass.

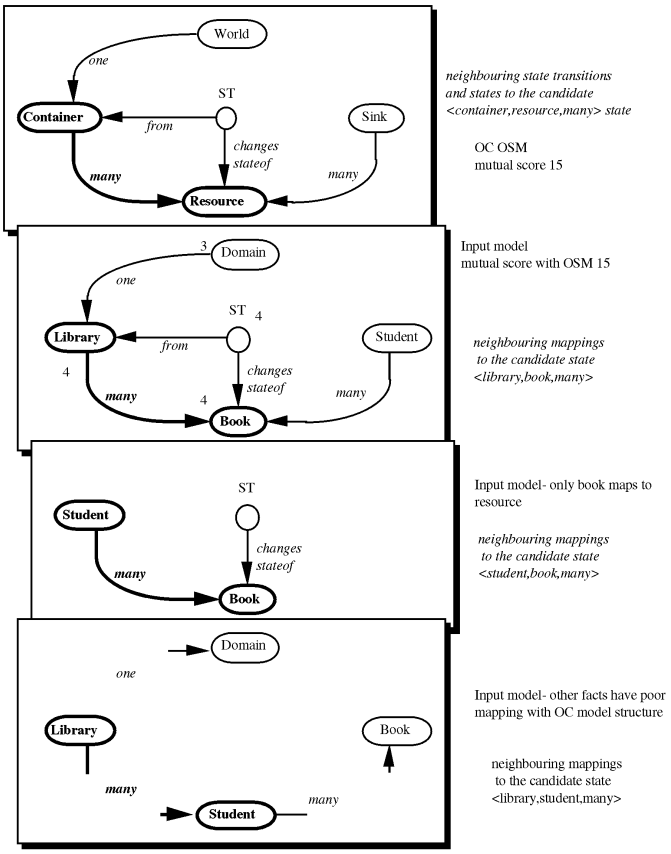


Fig. 7. Possible structure mappings between the structure <container-many-resource> and three Structure objects in the input model. The best-fit is with <library-many-book>.

2.3.1 Matching to the Lower-Level OSMs

Structure matching is used to map further input facts to the OR and OS subclasses using the same algorithm. It detects additional mappings with the *return* state transition and relationships between Structure objects and state transitions.

The rule-based matcher specializes the structure match from the OR class using goals, events and stative conditions. Following the example illustrated in the dialogue (Figs. 2, 3, 4, and 5), the OS class is detected by further structure matching as the user has entered additional facts describing the supplier and the transition representing delivery of goods, in this case transfer of books from publishers to the library. The OS model is a leaf node in the OC tree so the matcher calculates an overall goodness of fit score for the input model and the OS model.

Taking the second pass, not illustrated in the dialogue, the matcher selects first the OR model as the additional facts describing the borrower and a transition representing a loan have been entered. Matched goal states are inferred by unification using goal categories and fact-pair mappings which describe the desired goal states. For instance, the requirements statement specifying that books must be returned within a certain period is a goal which leads to a state of books-in-library and a relationship between the transitions loan and return. The matcher looks for mappings with states representing the desired goal (book in library) and the relationship defining a loan (i.e., a loan state transition is followed by a return and both are associated with a duration attribute). Similar rules map correspondences between input and system models for events, stative conditions, relationships and object properties. The OH model has no further subclasses; therefore, the rule-based matcher calculates the overall fit between the input and OH model.

The process involves an iterative cycle of user input and inferred mapping, so as more facts are entered the matcher progresses down the OSM hierarchy.

2.4 System Architecture

The architecture of AIR (Advisor for Intelligent Reuse) has six major components; illustrated in Fig. 8.

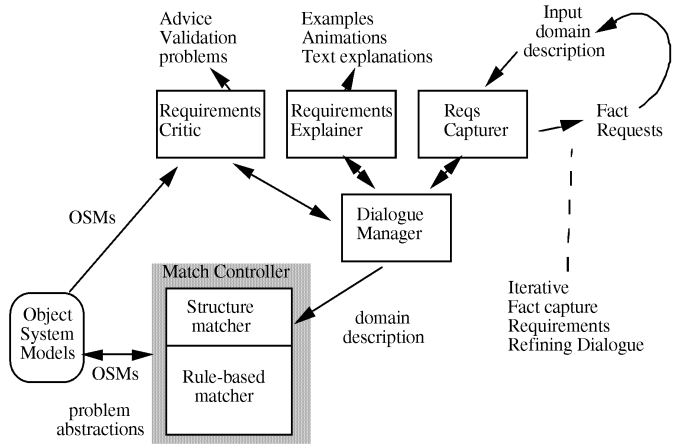


Fig. 8. Architecture of the AIR toolset.

The domain matcher, composed of the match controller, structure matcher and rule-based matcher, is implemented using ConceptBase [31] with Telos [45] and BIM Prolog. The tools and dialogue manager and the three tools (Requirements Capturer, Critic, and Explainer) are also implemented in BIM Prolog using the X Windows user interface environment. Each component is described in more detail, as follows.

2.4.1 The Matcher Controller

The match controller controls two mechanisms for retrieving OSM classes. At the beginning of the matching process, facts are acquired from the user in a system directed dialogue. These form the input model of the new application. The structure matcher uses a breadth first strategy for matching input facts to high-level OSMs then the rule-based matcher follows a depth first search down an OSM tree in an iterative cycle of fact acquisition and matching until one or more leaf nodes in the OSM family tree are reached.

Input facts are requested in the order of the OSM specialization, so the system asks for the fact-types it requires according to the depth of search in the hierarchy. Rule-based matching is interleaved with further fact acquisition directed by the needs of the matching process.

2.4.2 The Structure Matcher

Previous analogical matching mechanisms [14] were computationally complex and had excessive runtime [27], so the structure matcher attempts to overcome these problems by identifying structural patterns using a small number of

related fact types. The structure matcher uses a sophisticated, yet pragmatic search strategy to reduce the search space and avoid the exponential explosion of combinatorial search. Structure matching occurs in two phases:

1) Local mapping acts a filter to select the best candidate OSM for structure matching. Local mappings are controlled through a semantic lexicon, and detect instances of the same knowledge type; for example, the input fact book can be mapped to *resources*-object since both are typed as *objects*. All pairwise combinations of the input model and each candidate OSM are locally mapped by type to calculate a local match value which must exceed a preset threshold.
2) Structure matching takes the input model and one or more OSMs, depending on the local mapping filter, and determines a quantitative score of similarity between the input model and each OSM. The nearest neighbor mapping algorithm carries out an exhaustive search of pairwise combinations of knowledge types, constrained by the current locus of search. Structure matching is refined by weighting the importance of certain mappings. State transitions with respect to structure objects are central to discriminating between domains, therefore these mappings are assigned a higher score.

2.4.3 The Rule-Based Matcher

Full structure matching at subclass levels would involve redundant mapping, as the structural similarity already detected is shared by the subclasses. The rule-based matcher therefore maps isolated facts within the context of an existing structure match. The process dynamically swaps knowledge types according to the level of search within an OSM tree; moreover, the fact acquisition dialogue requests facts from the user appropriate for search strategy. Rule-based matching has two roles:

1) to infer knowledge type mappings needed to distinguish between candidate OSM subclasses
2) to match an entire domain model once the specialization has been selected. Output from the rule-based matcher is a quantitative score of matching with a leaf node OSM.

Rule based mapping uses 110 rules to test permissible neighboring mappings for each knowledge type. This rule set constrains the expected connectivity for any one fact, so fact pairs can only be mapped if they conform to relationships in the metaschema (see Section 3). Local mappings are achieved through types and properties (e.g., the similarity and quantity relations). The local mapping rule fires if and only if a fact describing the target domain and an OSM fact are instances of the same knowledge type and have the same attributes listed below. The knowledge types, with their *properties in italics*, are used to strengthen the detectability of candidate mappings:

- *frequencies* of primary state transitions
- *types* of secondary state transitions
- *object relations* in primary states
- *types* of secondary states
- event *cardinalities*
- *category attributes* for objects
- *attributes* for stative conditions
- *types* of goal expression for goal states

The system is implemented in BIMprolog, so description of the process as a procedural algorithm, which is illustrated in Appendix A, can only be an approximation to the fact unification and pattern matching process. The Prolog format of a sample local mapping rule is also illustrated in Appendix A.

2.4.4 Requirements Capturer

The requirements capturer acquires facts which discriminate between OSM classes. A mixed initiative dialogue enables iterative fact capture, retrieval of domain abstractions and guided development of a requirements specification. The capturer has two approaches for acquiring new facts from the requirements engineer. First, the dialogue requests fact types which are powerful discriminators, thereby enabling matching and selection of appropriate OSMs. Secondly, unguided requirements capture is controlled by the requirements engineer. Input is via a form filling dialogue which has a limited error repair capacity for facts that do not match the system's lexicon. In both approaches, the aim of the requirements capturer is to acquire facts about the target application which permit retrieval and explanation of domain abstractions.

2.4.5 The Requirements Critic

The requirements critic aids domain understanding and requirements validation by comparing the input model and retrieved OSMs, detecting problems and reporting these to the requirements engineer. The critic has three major features:

- a problem classifier which reasons about mappings inferred by the domain matcher to detect and classify problems in the input model representing the user's requirements. The classifier can detect incompleteness, inconsistencies, and overspecification in requirements and explains its rationale for detecting these problems
- critiquing strategies for requirements validation determined by the detected problem
- a controller for intelligent selection of strategies

Strategies are selected to encourage specification of complete requirements for an application class and to assist detection of inconsistencies, ambiguities, and wishful thinking in the specification. The requirements critic combines guidance and critiquing strategies in a mixed initiative dialogue.

2.4.6 The Requirements Explainer

Critiquing is aided by explanation of retrieved OSMs using guided exposure to prototypical examples, visualization, animation of retrieved models and descriptions of analogical mappings.

Diagrams representing OSM structures and state transitions are annotated and supported by text descriptions to aid recognition and understanding of domain abstractions [42], because people learn analogies more effectively if they

are presented with spatial diagrams illustrating critical determinants of the problem [19]. Prototypical examples are also used to aid understanding of the OSM abstractions, as people often understand new concepts using prototypical examples [58], [59]. Finally OSMs can be animated to illustrate domain behavior. Animations are interactive and permit playback with pause facilities to facilitate exploration.

2.4.7 Dialogue Management

The dialogue controller selects explanation/critiquing strategies based on properties of the requirements specification and simple states of user-system dialogue. Guidance is controlled by a set of heuristics derived from a task model of requirements analysis and validation. Active guidance in explaining/critiquing input using OSM abstractions is interleaved with user driven browsing of the OSM hierarchy. The requirements critic intervenes when serious omissions or inconsistencies are detected to explain retrieved OSMs and detected problems. Passive critiquing is used when the problem classifier detects less critical problems which are posted on a notepad as issues to be resolved when the requirements engineer chooses.

This concludes the description of the toolset. We now turn to the domain theory which provides the ontology for building the system's library of generic models.

3 Concepts and Models of the Domain Theory

The domain theory consists of three major components:

1) a metaschema or modeling language which defines the semantics for generic models of application classes;
2) a set of generic models organized in a class hierarchy;
3) a computational matching process, developed from analogy theory [27] for retrieving generic models appropriate to a set of facts describing a new application; as described in previous sections of this paper.

The theory has two viewpoints. First it is theory of abstraction in that it defines a schema for generic domain knowledge and secondly it is a theory of naturally occurring expertise. The latter view asserts that generic models of domain knowledge are naturally occurring phenomena held by expert software engineers and, to a less precise degree, by users. The theory proposes these models exist as a result of reasoning about related classes of problems leading to formation of generic abstractions by human cognitive processes of analogical reasoning [19]. The level of generality is broader than the domain knowledge used in domain-oriented environments [17] as we wish to reuse knowledge over a wider range of applications sharing a common abstract model, e.g., loans applications in car hire, library loans, video rentals, etc. Domain models are structured in a class hierarchy drawing on cognitive theories of memory [3], [6], [7], memory schemata [3], [56], and natural categories [58], [59] which asserts that human memory is organized as an informal hierarchy. There is good evidence that memory for several different types of knowledge (e.g., objects, procedures, and plans) is organized hierarchically [34]. Furthermore, generalization, specialization, and inheritance of objects are familiar concepts in software engineering. The theory also rests on empirical findings that experienced software engineers tend to recall and reuse mental knowledge structures or abstractions when specifying new systems [24]. A more detailed rationale for the theory and its motivation is given in Appendix B.

3.1 Models of Domain Knowledge

Generic models of domain knowledge fall into two subclasses: Object and Information Systems Models.

- Object System Models describe the essential transaction of the application in terms of a set of cooperating objects and their behavior (cf. use cases [29]). We introduce the concept of 'Structure objects' as a bridging abstraction which preserve some manifestation of the physical structure found in the real world. This enables OSMs to express facts about the real world domain which would not normally be included in conceptual models such as entity relationship diagrams, e.g., environmental phenomena such as buildings, rooms, warehouses.
- Information System Models contain processes which report on and provide information about an Object System Model. Object system models define the essential model of the problem [43], i.e., the application objects or entities and behavior related to achieving the system's purpose, while Information System Models describe processes which provide reports, ad hoc queries and other information requirements. This distinction is a development of Jackson's [28] ideas on separating the fundamental entity model of a system from the functional processes which acquire information from that model.

The focus of the domain theory is abstraction of the problem space rather than design models of computational functions, algorithms, and abstract data types. Models at this level of abstraction have been proposed by Shaw [62], Harandi and Lee [26], and in more detail by Smith [63], [64]. Before describing the OSM models in detail we introduce the metaschema, or our ontology for domain modeling.

3.2 Metaschema of Domain Knowledge

The metaschema defines eight knowledge types which form the primitive components of OSMs. The structure of the metaschema is shown in Fig. 9 using the format of an object-relationship diagram.

The semantics of each component are defined in Appendix C using the Telos knowledge representation language [45]. Telos itself has object-oriented semantics with inheritance, and its semantic primitives are mapped to first-order logic definitions, thus providing a formal knowledge representation language for the domain theory. The models have been implemented in the computational manifestation of Telos, ConceptBase, a deductive database [31]. First-order predicate logic provides a formal basis for reasoning to assist understanding of linguistic expressions of states, goal states, and behavior during requirements definition.

3.2.1 Key Objects

Objects are specialized into key objects, agents, or structure objects. Key objects are the subject matter of the essential

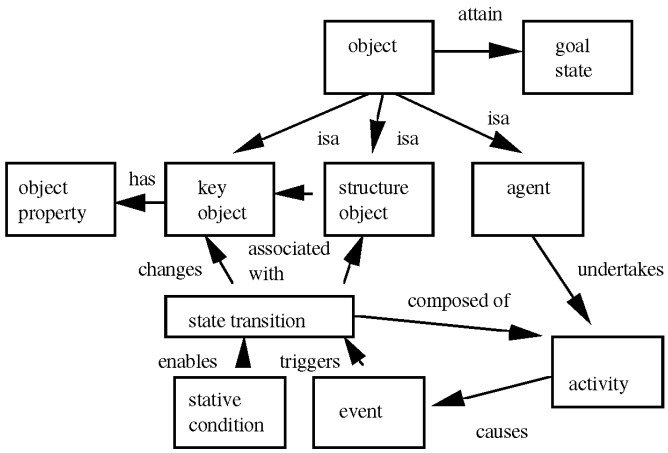


Fig. 9. Metaschema of knowledge types for domain modeling.

system transaction and therefore undergo state change. Key objects may be physical and subject to laws such as gravity and momentum, or financial or conceptual objects without physical instantiation.

3.2.2 Agents

Agents have properties and methods. Agents are specializations of objects and carry out activities which may then create events initiating state transitions. Agents can be subtyped as human or automated agents to describe a required computer system or a manual operator in a paper-based system:

3.2.3 Structure Objects

Structure objects preserve some physical aspects of the real world domain. They bridge the gap between domain-specific models [4], [17] and modeling languages [12], [44] which provide a general set of semantics. There are two motivations for introducing this concept. The first is to bridge between linguistically based knowledge representation [65] and more formal semantics. Secondly, existing modeling languages such as KAOS [12] do not contain the structure necessary for analogical computational matching, therefore we add structure as a semantic primitive. Structure objects represent passive objects and environmental facts which would not normally appear in data models, e.g., warehouse, library, air-corridors in air traffic control. Specializations in OSM families may add further Structure objects, for instance, a distributed warehouse application would be modeled by several container Structure objects representing branch warehouses.

Structure objects model approximations to the real world entities which must be persistent, have spatial properties and express containment or possession of key objects, for example, a library contains books. The relationship between key and structure objects is equivalent to a state. Objects change state with reference to a structure, e.g., an object is sold and physically moved out of the warehouse structure to a customer structure. This is logically equivalent to a state transition from in-stock to issued-from-stock.

Structure objects can model physical entities and states in two ways:

1) by creating an internal structure within a structure object. This is used in Object Allocation and Object Sensing OSM families in which the structure object is conceptually divided into an arbitrary number of parts. These parts are then used to record state of a key object, e.g., in Object Sensing aircraft(x) is-in structure-object subpart(A1)... air lane sector.
2) by creating additional structure objects when specialising an OSM family, for instance in Object supply, further structure objects may be added in a subclass to model a central warehouse and branch-warehouses. This would be advised if there are significant state transitions between the two structures, e.g., stock is taken from the central warehouse and distributed to branches before being issued. If, however, the purpose of the system is just to record where stock is located without controlling its distribution, a subdivided internal structure would suffice. The choice is left to the analyst.

3.2.4 State Transitions with Respect to Structure Objects

State transitions with respect to Structure objects are central to the domain theory for two reasons. First, they provide the structural association necessary for analogical retrieval and secondly they model behavior which achieves the system's goal. A state transition changes the state of one object by transferring its membership between Structure objects to achieve desired goal states.

The power of state transitions to discriminate between models is best illustrated by an example in Fig. 10. In the first OSM, stock is dispatched from the warehouse to the customer while books are lent from the library to the borrower in the second OSM. Both systems have structures representing objects held in and leaving a container but the library is distinguished by an additional state transition returning the outgoing object to the container.

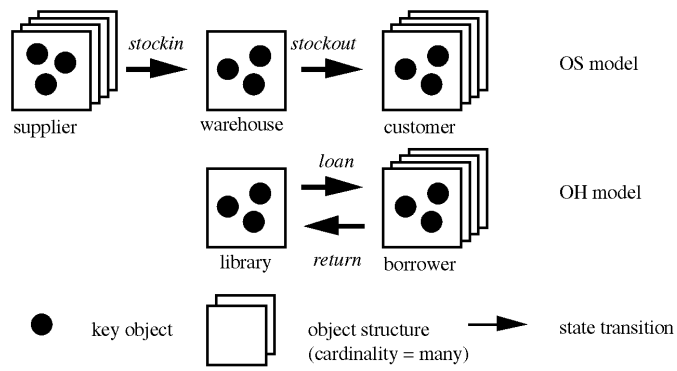


Fig. 10. Key components of Object System Models.

3.2.5 States

States are described in two ways. Primary states record the containment or possession of objects in structures. Secondary states belong to objects independently of structure and describe states such as being reserved or scheduled. The distinction between primary and secondary states is a consequence of Structure objects. The combination of Structure objects and state transitions provides good discrimination

between domains, whereas secondary states are less powerful. For instance, object hiring (library lending) is distinguished from a stock control class by a return state transition; however, object reservation (a secondary state) does not distinguish between classes since reservation occurs in many domains including stock control and library lending.

The following subtypes of secondary state are used for discriminating subclasses within an OSM family:

- *suspension.* The key object is, for a period of time, no longer a part of the object system; for example, a student does not attend classes while on industrial placement.
- *inhibition*. The object exists in the system but it may not exhibit behavior because its state prevents it from doing so, e.g., a student cannot continue with the course having failed the exams.
- *marked*. The object is marked to denote the result of some activity such as planning, scheduling, reserving, etc. All these activities result in the key object becoming available for a subsequent state transition, e.g., a reserved book may be loaned.
- *checked*. The key object is checked by an agent, for example a vehicle is checked by the garage to ensure its safety. Validation and testing type activities change an object's state from unchecked to checked. This is similar to the "marked" state but we differentiate checked to represent the outcome of quality assurance activities.
- *damaged*. The key object is damaged so that it cannot fulfill its purpose, for example a book is damaged and cannot be loaned to borrowers. Damaged objects are changed to an undamaged state by repair activities;
- *augmented*. The key object is augmented by changing the internal state of the object so that it is improved with respect to attaining the purpose of the system. For example, software is augmented through upgrading to a new version of an operating system .

3.2.6 Goals

Goal states describe a future, required state which the system should satisfy, maintain or sometimes avoid. Requirements are usually expressed linguistically as intentions for goals, aims, and objectives. These statements can be refined into more formal expressions of a future state which the system should (or should not) attain. Such state expressions link linguistic statements of goals to Object System Models. Goals are specified by OSM components in three ways:

1) goals describing states which the object system must achieve. These requirements are satisfied by state transitions on key objects
2) functional requirements describing algorithms and processes which must be carried out to satisfy the goal state. The completion of the activity causes a state transition to attain the state
3) functional requirements will necessitate production of information which are satisfied by activities in the Information System Model

Each OSM has a goal state definition which is part of the essential model of the system purpose, and this is vital for discrimination.

3.2.7 Activities

Activities belong to agents and are processes which normally run to completion resulting in a state change within an Object System Model. In addition, activities may model human tasks and procedures in the Information System Models. Specific activities are equivalent to primitive functions in Data Flow diagrams, or algorithm level detail in other specification languages. Specific activities are abstracted to define a generic set of activities, e.g., reserve-resource, match-objects, and repair-object. These activities change primary or secondary states, for example the generic reserve-resource activity can be instantiated to reservation of library books by borrowers. Activities generally are only weak discriminators between OSMs, as one activity may occur in many different types of applications, e.g., algorithms for scheduling, allocation, etc. occur in many different systems. In view of this we treat activities as a descriptive facet (cf. [51]) attached to OSM models, and their use in matching is restricted.

3.2.8 Object Properties

Properties define characteristics of key objects which constrain their behavior. Three subtypes are distinguished:

- *physical* objects which are subject to physical laws such as gravity or momentum. Examples of physical objects include library books in the lending library domain and aircraft in the air traffic control domain. These objects can undergo transitions resulting in physical change.
- *financial* objects represent currency or some monetary value. Examples of financial objects include cash in an automatic telling machine and the currency in a foreign exchange transaction.
- *conceptual* objects are not physical or financial objects, so this subtype acts as a "bucket" category for the first two. Examples of conceptual objects include transaction related documents, e.g., purchase orders, tickets, vehicle licence, etc.

3.2.9 Events

Events are defined in two subtypes; domain and time events. An event is a single point in time when something happens, following Allen's [2] conception of temporal semantics. Events are treated as semaphores which initiate a state transition. Temporal semantics are also expressed in state conditions (see following section). Events are implicit in state transitions. To avoid redundancy we only model initiating "triggers" as events, i.e., those events which start state transitions. For instance, the dispatch of goods from the warehouse to customers is triggered by a customer order (a request event). Triggering events are not needed to "kick-start" behavior in all object systems as autonomous behavior can be a normal property; for example, aircraft move between sectors in an air space without any initiating event.

Domain events model the result of some external activity in the real world while time events emanate from a clock. Events have a cardinality attribute expressing a relationship

between the event and state transitions it triggers. An event may trigger one instance of a state transition; many state transition instances from and to the same structure object; or many state transition instances from and to different Structure objects.

3.2.10 Stative Conditions

Stative preconditions and postconditions are tests on primary and secondary states. Several types of conditional tests are used for discriminating between lower order OSM classes:

- tests on primary states which must be successful for the state transition to be initiated, e.g., a book must be in the library before it is loaned
- tests on the number of key objects contained in a structure object, e.g., stock is replenished when the number of stock objects in the warehouse reaches a minimum-level
- tests enabling transitions to secondary states, e.g., a theater-goer's request must match the attributes of an available set a before the place can be reserved

3.2.11 Relationships

Relationships add further structure information to Object System Models which is used to discriminate between lower-level classes. Relationships are subtyped as:

- *cardinality* relations which express the expected membership of key objects within a structure object, e.g., many aircraft can be in one airspace
- *temporal* relations which indicate that an object A should persist in structure A′ for a significantly longer time than object B does in structure B′. It is defined as A longer B
- *scale* relations which indicate that a structure object A has a larger set of key objects than another structure B. It is defined as A greater than B.

Relationships provide additional structure to enable analogical matching and discriminate between OSMs as illustrated in Fig. 11 for a generic and specific model.

Relationships may also link state transitions to define sequential ordering, so that one transition precedes another in changing the state of an object instance. For example, in the lending library, loan of a book instance must precede the return of the same book. Relationships complete the knowledge metaschema for modeling Object System Models.

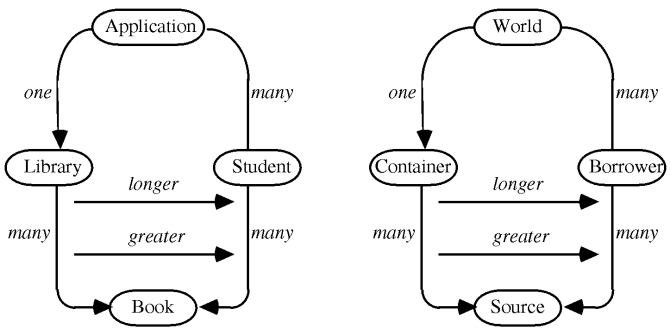


Fig. 11. Matching using relationships.

3.3 Object System Models

Any one application is composed of an aggregation of object system and information system models. The matching process (see Section 2) retrieves an appropriate set of models for a particular target application, so systems of different sizes and complexity can be handled.

The system knowledge base for the matcher is populated by a class hierarchy of Object System Models and an orthogonal set of information system abstractions. The overall Object System Model hierarchy is illustrated in Fig. 12. The top levels of the tree have been pruned, as models at such high levels of abstraction do not contain sufficient knowledge for structure matching and are not informative. The hierarchy, therefore, starts with nine separate trees, which we term OSM families. A brief description of each top-level class is given with lower-level detail of the OC family which was used to illustrate structure matching in Section 2.

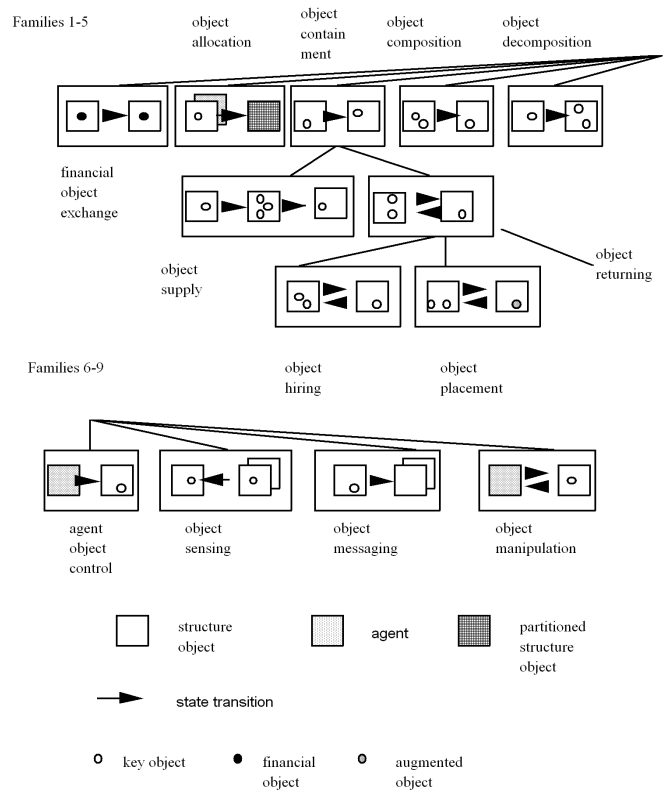


Fig. 12. Object system model hierarchy showing subclasses of the Object Containment Model.

Level-1 class: Object Containment. This class is characterized by the unidirectional transfer of key objects from an owning structure to a client structure. More familiar examples are all sales order processing systems and specializations in this tree also describe inventory and loans type applications. The model has two Structure objects, clients and resource owners, one key object and an event which models the request for the transaction. The common purpose of this class is the orderly transfer of resource objects from an owner to a client who requires the resource. The goal state asserts that all requests for key objects must be transferred to the requesting client. This class only models the transfer of objects, while a separate

abstraction is used for the monetary aspects of sales. Hence the object containment family models a wide variety of inventory type applications and may be specialized to many nonsales systems where resources are transferred, e.g., internal supply, donations of resources, etc.

Level-2 class: Object Supply. This specializes the OC object system by adding another structure object to model the source of new objects, and a further state transition to represent the transfer of objects from a source to the owning structure object. This specialization corresponds to a variety of inventory management systems with outbound and inbound transfer of objects. Purpose in this case is not only orderly transfer of objects to a client but also replenishment of resources from suppliers; hence the goal state is that all outbound requests by clients and requests initiated by the system for inbound transfer of key objects must be satisfied.

Level-2 class: Object Returning. This model is the other subclass from the OC model. In this case the specialization adds a single state transition inbound from the client to the owning structure object. The purpose of this object system is the orderly transfer of resources to a client and the return of those resources to the owner, hence this class specializes to hiring, servicing, and repair type applications. The goal state asserts that all loaned key objects must be returned to the container at some specified time in the future.

Level-3 class: Object Hiring. The purpose of these systems is to control the loan, hire, or temporary transfer of ownership of some resource from an organization and a client who requests possession of the resource. This object system is characterized by two structure objects, the owning organization and the client, one key object (the resource) and two transitions representing outbound transfer of the resource to the client and a subsequent return. The goal state is the same as for the OR model, as is the key object.

Level-3 class: Object Placement. The purpose in this object system is to control the exchange of objects between a client and a servicing organization. Further specializations map to service and repair applications in which the key object is in some way improved or has value added to it by the servicing structure object. This subclass has the same structure as in the object hiring model, but is differentiated by the concept of augmentation of the key object. This concept is represented by a relationship between the inbound and outbound state transitions and is specified in the goal state.

Level-1 class: Object Composition. In this class the purpose is to aggregate key objects and thereby synthesize new ones, so the goal state is that all input objects should be composed into one or more new composite key objects. Concrete instantiations are a wide variety of manufacturing systems which take in raw materials and compose them by some process into a product. This object system is characterized by two structure objects (the source of objects to be composed, and the destination of composed objects), two sets of key objects (components and composite objects), and one transition for merging component objects into composites. A controlling agent is also present in this model to describe organization of the process. Robotic manufacturing systems are match to this family aggregated with the object manipulation OSMs.

Level-1 class: Object Decomposition. This is the opposite of composition; it maps to concrete examples of systems which disaggregate objects, e.g., disassembly, unpacking, or applications involving decomposition. In this case the goal state at the end of system activity is that no composite key objects should exist. This system is characterized by source and destination Structure objects, two sets of key objects, composites and components, and one transition representing decomposition. As with object composition a controlling agent is present.

Level-1 class: Financial Object Exchange. This class is a special case when structure has not been used to differentiate classes at the top level. Financial and accounting systems share many structural properties with other classes. Furthermore, in many domains accounting systems are aggregated with other transaction processing. Given that finance is a salient descriptor of business information systems this class is discriminated by possessing a financial key object. The top-level structure is the same as object containment and this class shares many of the same structural specializations. Loans, for instance, have the same structure as the OR abstraction as the loan is augmented by being repaid with interest. The goal states are similar to the OC family, with the addition that a monetary value must have exchanged between agents and Structure objects.

Level-1 class: Object Allocation. This family models applications that create an object state change that is a precursor to some other transaction, e.g., booking type of applications. The purpose of such systems is to establish an association between a key object and a client so a further transaction is enabled. The second transaction is consummatory in nature when the allocated resources are used by the client, however, this will be modeled in a separate OSM family e.g., inventory control in the OC family. This class is modeled by two structures, the source of unallocated objects and destination of allocated objects and an agent which is responsible for the activity which determines the link. The destination structure is conceptually partitioned to model the structure of slots with which key objects may be associated, examples being theater seats after booking, timetable slots allocated to lectures, etc. This OSM is often aggregated with containment and hiring models.

Level-1 class: Agent-Object Control. This OSM family models command and control applications. The class is characterized by a controlling agent and one or more controlled key objects contained in one structure object. In this class key objects are contained within a structure which models a controlled part of the real world environment, e.g., airspace in air traffic control. The transitions represent messages passed from the agent to key objects to change their behavior in some way. Objects respond to commands and change state within the structure, and specializations of this class refine how that change occurs with respect to structure, (e.g., spatial, temporal change, change in property, number, etc.). The purpose of this class is expressed as an agent having some effect on objects' behavior in a structure. The goal state asserts that the controlled key object must respond to the agent in a satisfactory manner. Specializations of this class are command systems which add further agents

for military command and control applications, controllers, and pilots in air traffic control systems, etc. This class is often aggregated with object sensing and messaging abstractions.

Level-1 class: Object Sensing. The purpose of this class is to monitor the physical conditions or movement of objects in the real world and record their relative state with respect to a spatial/physical structure. For this family, the goal state is that changes in key objects must be detected and signaled to the agent. Key objects change with respect to a structure and these changes are reported to the sensing agent as an event. This class includes a sensing agent which reports on this behavior and specializations are spatial sensing (e.g., radar detection of air traffic), temporal monitoring and property sensing applications (e.g., monitoring temperature, pressure in process control applications). In subclasses of this family, Structure objects model significant spatial components of the real world and can be partitioned, e.g., sectors within an air corridor. Object sensing models are often aggregated with the Agent-Object Control class.

Level-1 class: Object Messaging. The purpose of this class is to communicate key objects which have some information value between Structure objects representing the communication network. Specializations of this class add agents which are purposeful originators and receivers of key objects. The class is characterized by one or more key objects (messages), two or more structure objects representing the network topology through which communication takes place and at least one state transition for exchange of messages. Specializations in this class add more structure objects for communication topologies (e.g., star, network, bus, etc.) and more transitions with relationships to represent different communication protocols (e.g., simplex, half duplex, full duplex, reply-acknowledge, etc.). The goal state of this family asserts that key objects must arrive at their destination, however, within subclasses the key, transferred object may be conceptual or physical. Physical key objects create subclasses for logistics, i.e., the key objects to be distributed are physical goods. The goal state for this family simply asserts that a key object must be delivered from the sender to the receiver. CSCW (Computer Supported Cooperative Work) applications are usually aggregations of Object messaging and the next class, Object manipulation.

Level-1 class: Object Manipulation. In this class the key object does not change state with respect to structure; instead it may undergo many incremental changes which change its secondary state. The state transitions model a change emanating from an agent which alters a key object and feedback from observed change to the manipulated key object. The goal state specifies that any change requests from the agent are acted upon by the key object. No key object movement is involved in this abstraction; instead structure object models the device which facilitates or carries out the manipulation, e.g., a machine or a computer. This class represents a large family of applications including computer based tools for word processing, graphics packages, decision support, design support, CAD systems, etc. From the design viewpoint this class also contains all computer programs. However, the focus of the domain theory is on requirements, so the essence of this class is applications which require manipulation of an artifact in its own right. These systems are characterized by an agent, who carries out actions on the key object mediated by a machine or computer support tool.

This is the breadth of the Object System Model space as it currently stands. Further analysis of applications may lead to discovery of new OSM families; however, the set described above has remained stable over 18 months of successive analysis of applications from real life and case study descriptions taken from the software engineering literature.

So far detailed specialization has concentrated on the Object Containment, Object Allocation, Object Sensing, Object Composition, and Agent-Object control families. The current domain space is composed of over 120 Object System Models. A set of 20 generic information system functions has also been modeled; examples include report values, states, count individuals matching criteria, list-duration, and progress tracking.

It is important to emphasize that in any one application, there is usually an aggregation of Object System Models. For instance, a typical library domain will have an Object hiring system to handle book loans, an Object supply system for book stock control, new book stock acquisitions, and possibly Financial object exchange for book purchase with Object repair to return damaged books to a satisfactory condition.

3.3.1 Modeling Information Systems

Information System Models differ from Object System Models. They represent processes which feed off, report on and provide external representations of the Object System Model. OSMs contain information in primary and secondary states, however, they have no means of producing reports or external representations of that information. Producing information for users, who may be human agents modeling within OSMs is the responsibility of Information System Models (ISMs). Thus in an inventory application, modeled by an Object Supply OSM, the key objects in stock maintain information about their state. Accessing that information is carried out by an ISM process which may produce an inventory report as its output. We have this distinction because we propose generic report types and processes as an orthogonal set of abstractions which may be applied to any OSM. This concept also owes its heritage to Jackson System Development [28] which differentiates between system entities and information producing processes. Information system models provide information to agents for their decision making and action, so this definition is narrower than the concept of an information system which also encompasses transaction processing.

An information system activity reports the primary and secondary states of a key object. Information System Models also have activities which produce reports, queries and other information outputs. These can belong to agents, are triggered by events and are controlled using preconditions as in OSMs. To illustrate the roles of OSM and ISMs, a package express delivery business matches the OM family as a subclass of Object logistics. The essential model of the business is getting packages, etc. from one customer to another via series of depots. This would be modeled in an OSM with key objects (packages)

being moved between structure objects (sender, receiver customers, depots). The number of structure objects and their topology depends on the level of specialization. The information system which then tracks the progress of packages from source to destination is handled by a progress reporting ISM which samples each structure and reports where each key object instance is.

3.3.2 Customization Tools

OSM families may be extended by specialization within the constraints laid down by the domain theory ontology. The OSM library is thus extensible and customizable to more detailed domains. Customization tools are provided with the toolset to add new knowledge types to OSM models, although the user interface is, as yet, primitive. The system library could be extended further by addition of completely new families. The matcher is a general process which will search any library of models organized according to the domain theory schema. While we doubt that new families will be required, specialization of current OSM families to model more domain detail is an important requirement, and we are working on improved customization tools.

3.3.3 Validation Studies

The OSM library structure and semantics have been validated using experiments to elicit mental abstractions possessed by experienced software engineers and users. Protocols and knowledge acquisition techniques were employed to elicit experts' mental knowledge structures and domain categories, following established practices [7]. Experimental studies used card sorting exercises to test whether expert software engineers naturally sorted descriptions of specific applications into generic categories predicted by the domain theory. Independent judges then assessed whether the resulting category groups produced by the experts accorded with the domain theory abstractions. Three OSM families and three OC subclasses were tested with 20 experienced software engineers, who were also asked to judge the utility of the intermodel discrimination criteria proposed by the domain theory.

The experts' categorization agreed with theory predictions apart from the FO (financial object exchange) family, where ironically experts categorized according to the underlying structure of the exchange and placed these applications within the OC family. In retrospect we should have conformed with our own assertion that structure and transaction behavior determine generic classes, not properties. However, some experts did create a new category based on safety critical properties, so categorization by attribute rather than structure requires further investigation. The experts agreed with the domain theory discrimination criteria, so overall, we believe there is good evidence that the theory has cognitive validity in predicting memory of problem abstractions. Further details of this study are described by Maiden et al. [42].

4 Discussion

The computation implementation of the domain theory has been applied to component retrieval in specification reuse [38]. The matcher and its knowledge base of generic models have been tested with several different problems [40] demonstrating that the theory is robust, the tool support is effective, and that generic models can help reuse of knowledge in requirements engineering. Furthermore, this approach could be developed further by integrating the matcher with a CASE repository. Through two-step matching, from new requirements to generic models and then from generic models to existing specifications, the matcher could find existing specifications which are appropriate solutions for a new application. In this scenario the computational model of analogy, implemented in the domain matcher, can deliver interdomain reuse. We have experimental evidence that this is feasible when combined with cooperative assistance tools for explanation [66], [67].

The utility of domain theory for supporting requirements validation is demonstrated in critiquing a developing application model. The requirements critiquer briefly described in this paper, and more extensively in [41], employs retrieved generic models to point out missing or inconsistent facts to the requirements engineer. We also place considerable emphasis on explanation, as understanding the problem domain is an important part of requirements analysis. The requirements explainer/critiquer can afford cooperative assistance similar to that provided by the Domain-Oriented Design Environments [17]; however, because the domain matcher uses generic models our tool can function in many different domains, unlike Fischer's domain specific tools. DODEs may be considered to provide generic knowledge in horizontal domains i.e., general purpose applications such as window layout design in user interfaces; however, for most applications specific domain knowledge has to be added to the DODE knowledgebase, e.g., knowledge about kitchen layouts. While the level of critiquing in our tools might not be as detailed as in a domain specific tool, this can be traded off against the development effort of creating new tools for each specific domain. We believe the domain matcher will be more effective in supporting requirements definition in an organization with many different types of applications, while domain-oriented support may be more effective for organizations dedicated to a narrower range of applications. The utility of the theory is now being explored in elaboration and validation of requirements in different application areas, as reported in this paper and in more detail in other papers [41]. However testing with further problem classes will be carried out to further validate our ideas.

We believe that the domain theory is the first systematic formalization of generic domain knowledge for requirements engineering. Several reports have hinted at taxonomies, classes, templates or generic applications [55], [52]; however, few reports have described a significant number of abstractions which can cover a range of software engineering problems. Similar abstract classes have also been proposed in AI, notably the Generalized Task (GT) theory of Chandrasekaran [5] and the abstract models proposed by Gruber [22], [23]. Generalized tasks, however, are abstractions of problems and methods for their solution; in contrast, our Object System Models emphasize the essential model of system problem motivated from a teleological viewpoint. The causal models of GT theory are similar to the

agent and state transitions described in the Object System Models. Knowledge engineering methods, notably KADS [70] have also espoused use of generic models in analysis and design of knowledge based systems, although only a few models have been described and these focus on the solution domain, e.g., reasoning models for diagnosis, scheduling, etc.

In software engineering libraries of abstractions have been developed at the design level of abstract architectures [62] and libraries of reusable functions have had some success in practice [49], [26]; however, the absence of a sound theory hinders definition of the granularity and level of abstraction of reusable components. Without such a basis, standardization and exchange of reusable components will be impaired. It is notable that the knowledge engineering community has devoted considerable effort in this direction in the Ontolingua [23] and KADS projects [1]. Like the domain theory these have proposed an ontology and generic models of domain structures and problem solving methods, although the libraries of models in both these projects are, as yet, modest. Within software engineering, the domain theory provides a systematic guide for constructing reusable components with a mechanism for matching and retrieving such components from large scale reuse libraries. Whether a finite set of generic models exists remains an open question. One danger is incremental erosion as more categories are added ad hoc to deal with exceptions, a criticism levelled at case grammars that also proposed regular semantic structures [15]. However, case grammars have proven robust for describing a wide range of natural language semantics [16]; moreover, the domain theory models are at a higher level of granularity than case frames, so while our schema is less versatile, it does not have to deal with the variety of natural language semantics. However, matching natural language domain descriptions to domain models is part of our program of future research, so we will have to address the problem of matching the variety and ambiguity in natural expression to concise specifications. We will approach this problem by developing constrained dialogues from our current matcher tool with case frame grammars.

Experimental testing has shown that part of the domain model library conforms with predictions from the cognitive view of the theory and were recognized in blind testing card sorting experiments by expert software engineers. The set of generic models has remained stable through iterative testing against case study examples but this offers no proof of a finite set. While we believe the top level abstractions are stable, the subclasses within each family have not been tested. It is unlikely that a complete library of models could be developed without substantial resources. The domain theory offers a framework within which other projects could create more specialized libraries. Extending the theory to give more systematic rules for creating subclasses is part of our ongoing work.

The KAOS project [12] shows many interesting similarities with the domain theory, although the emphasis of this project is supporting the process of requirements engineering via a formal language for goal-related requirements modeling rather than providing a library of reusable models. KAOS provides a modeling language and a process of deriving constraint-based specifications from requirements expressed as goals, similar to our approach of linked goals to object-oriented models for satisfying goal states. Generic models are also advocated by Dardenne [12], although a library of such models is not specified.

Further work is required on connecting the domain theory with the process of requirements engineering. Users often express requirements as goals relating to tasks, so a combination of requirements capture using goal refinements linked to the matcher may be one improvement to the matcher's fact capture dialogue. Although we have started by linking object modeling to goal and policy expression of requirements [68], further research is necessary into the process of policy and goal decomposition. Mylopoulos [46], Yu [72], and Chung [9] have proposed elaborating and refining requirements statements by using agent-organization models to describe the interdependencies between goals, activities, and agents. They describe a linked series of models for argumentation, organizational descriptions, and requirements with some FOL formalization of responsibility and dependency relationships, although, the power of the formality is limited by treating complex social phenomena, e.g., power, authority, responsibility, etc. as unitary variables. Less formal argumentation models [33], [54] take a different approach by recording the rationale behind requirements analysis to help the process of domain understanding. These approaches are complementary to the domain theory. Integrating rationales with generic models within the requirements engineering process is one direction we wish to investigate.

In our future work more example-based domain analyses will be undertaken to extend the coverage of the current set of object and information system models. Further development of prototype tools will progress towards integration of existing tools into a coordinated toolkit capable of assisting many different requirements engineering tasks. Usability testing is in progress with the domain matcher, requirements critic, and requirements capturer to enable more extensive and realistic assessment of the utility and effectiveness of domain knowledge during requirements engineering. Domain abstractions may also be extended towards support for conflict detection and viewpoint resolution (e.g., [36], [13]). How viewpoints are detected and resolved is poorly understood and the domain theory models could provide a shared artifact for viewpoint reconciliation while the domain matcher could support detection of viewpoint clashes and overlaps. e.g., viewpoint integration by using the OSMs to establish goodness of fit between two requirements specifications held by different stakeholders.

Finally to deliver effective industrial strength software engineering the domain theory and its computational process will have to be integrated within commercial CASE tools environments supporting standard structured methods (e.g., SSADM). This challenge is a prime means of establishing the utility of the domain theory which we shall follow in parallel with empirical studies and formalization of domain theory semantics, e.g., order-sorted FOL [69] to further validate the theoretical claims.

APPENDIX A

A.1 Algorithms of the Matcher Process

The algorithm first determines mappings between states, state transitions, and semantic relationships. Local mapping detects the knowledge types shared by both models using rules to test for each knowledge type defined in the metaschema. One hundred and ten rules define the permissible neighboring mappings for each knowledge type.

In procedural outline the algorithm is:

PROCEDURE for retrieval of OSMs
- set search-level = 0
- set matched-OSM = top-node
- CALL PROCEDURE matchcontroller

ENDPROCEDURE

PROCEDURE matchcontroller
- set unable-to-specialize = false
- REPEAT until unable-to-specialize = true
  - search-level = current-match + 1
  - set number-of-retrieved-OSMs = 0
  - REPEAT for each child of matched-OSM at search-level
    - CALL PROCEDURE feasibility-check using local mapper
    - IF feasibility-check = Success
      - IF matched-OSM = top-node
        - CALL PROCEDURE structure-matcher
      - ELSE
        - CALL PROCEDURE rule-based matcher
      - ENDIF
      - IF structure-match = Success
        - number-of-retrieved-OSMs = number-of-retrieved-OSMs + 1
  - ENDIF
- ENDREPEAT
- IF number-of-retrieved-OSMs = 1
  - current-match = current-match + 1
  - set matched-OSM to retrieved OSM
- ELSEIF number-of-retrieved-OSMs > 1
  - cooperate with requirements engineer to select best-fit OSM
  - current-match = current-match + 1
  - set matched-OSM to selected OSM
- ELSEIF number-of-retrieved-OSMs = 0
  - acquire new facts about the target domain
  - set unable-to-specialize = true
  - ENDIF
- ENDREPEAT
- IF current-match _ lowest-level (more matching is possible)
  - cooperate with requirements engineer using requirements critic
  - ENDIF
  - IF current-match = lowest-level (no more matching is possible)
    - CALL PROCEDURE retrieval of ISMs
- ENDIF

ENDPROCEDURE

This procedure represents typical retrieval of OSMs and ISMs. It does not describe the nature of interaction with the requirements engineer through the critic, which is controlled by the dialogue manager. An outline of the structure matching algorithm follows, although, as the system is implemented in Prolog, a procedural algorithm is only an approximation to the fact unification and pattern matching processes:

PROCEDURE structure-matching for one OSM
- REPEAT mapping cycle FOR all state transitions and states discriminating OSMs at the current search level
  - CALL PROCEDURE local-mapping
  - IF number of local mappings = 0
    - reject-knowledge-instance-pair mapping
  - ELSEIF number of local mappings = 1
    - CALL PROCEDURE neighbor-mapping-fit
    - IF neighbor-mapping-score > Threshold
      - record knowledge-instance-pair-mapping
    - ELSE
      - reject knowledge-instance-pair-mapping
    - ENDIF
- ELSEIF number of local mappings > 1
    - CALL PROCEDURE neighbor-mapping-fit for candidate knowledge-instance-pair mapping
    - CALL PROCEDURE neighbor-mapping-fit for other candidate knowledge-instance-pair mappings
    - IF candidate-knowledge-instance-pair-mapping-score > other-knowledge-instance-pair-mapping-score
      - record knowledge-instance-pair-mapping
    - ELSE
      - reject-knowledge-instance-pair-mapping
    - ENDIF
  - ENDIF
- ENDREPEAT

ENDPROCEDURE

Neighbor mapping represents an exhaustive search of pairwise combinations of knowledge types, constrained by connections from the current locus of the search. Structure matching is refined by the use of weights, which denote the importance of mappings between different knowledge types. State transitions with respect to structure objects are central to discriminating between domains, therefore the structure matcher gives greater importance to these mappings.

A.2 Example of Prolog Rule

Prolog rules detect a local mapping, for example, between two primary states describing a library containing a set of many books and a container containing a set of many resources, for the OSM called OC:

```
localmapping(_osm):-
        target_primary_state(_tobject1, _tobject2, _trelation),
        osm_primary_state(_cobject1, _cobject2, _crelation,_osm),
        _trelation==_crelation.
```

where the following predicates describe the primary states of the OSM and input model:

```
target_primary_state(library, book, many).
osm_primary_state(container, resource, many, OC).
```

APPENDIX B

B.1 Rationale and Origins of the Domain Theory

The domain theory is a hybrid theory which bridges cognitive science and software engineering. As a consequence its claims from different perspectives require some explanation. Essentially the theory is motivated by cognitive science and belongs to theories of knowledge representation and memory in the categorial tradition. However, it also functions as a theory of expertise which is applied in computer science to predict and explain concepts of abstraction which should have demonstrable utility in requirements engineering and software reuse. To elaborate these points a little further; we deal with each perspective in turn.

B.1.1 Cognitive Science Origins

The domain theory draws upon three precursor theories:

Natural categories from Rosch [57] in its various forms. This describes human memory for objects in terms of approximate memberships of categories with exemplars (called prototypes, e.g., a robin is a prototypical bird) being better members than atypical examples (a penguin is a rather atypical bird). Natural categories proposes knowledge is organized in three-level hierarchies in which the basic level is the middle (e.g., birds) with more detailed subordinates (species of bird) and superordinates (birds being members of the category animals).

While natural categories seem to work well with physical concrete objects, the theory is less predictive for abstract and functional categories, e.g., classes of events and tasks [25]. The notion of simple categories has evolved (see Lakoff [35] for a review) into radial categories which extend the categorial concept of Rosch towards network like schema. A further problem is whether natural categories claim to be a representation theory of knowledge structures in the mind or a generative theory whereby categories are simply manifestations of processes mediated by language. Neural nets and connectionist research suggests that categorization may well have some neuropsychological basis, however, this debate is ongoing in psychology and need not concern us further.

The connection from natural categories to the domain theory is indirect. The notions of classes and prototype examples has its origins in Rosch's work, but because the domain theory models are functional rather than simply structural we must look elsewhere for a baseline theory.

Gentner's [19] structure matching theory of analogy. This is a process and representational theory. Analogical knowledge follows schema theory origins [60], [53] which propose memory is organized in semantic networks of connected facts, or schema. Recall accesses a connected set of facts rather than isolated items. Structure matching proposes that abstract schema are learned when people recognize and solve problems. The abstract schema is built from the concrete source domain and has mappings from the source to the analogue schema and then to a new target domain. For example the solar system of planets revolving around the sun may give rise to an abstract schema of a central object with orbiting subobjects (planets) governed by conflicting force fields (momentum and gravity). The abstract structure can then be transferred to understanding a new domain via the analogy, e.g., a molecular system may be explained as a nucleus (central object) with orbiting subobjects (electrons), governed by forces (electrical attraction/repulsion, weak forces).

Gentner's theory forms the most important baseline for the domain theory. We are proposing a set of analogical abstract schema as object system models. Rather than describing analogies in everyday life, the domain theory proposes that a set of analogies represented as abstract memory schema may be built up by repeated exposure to similar problems which come from different domains. Hence, the domain theory is indeed a theory of analogy as the concrete examples which form the analogy belong to dissimilar contexts (e.g., library loans and car hire), yet they share a common abstraction (loaning resources).

Theories of ecological memory. Schank's work on dynamic memory schema as scripts and memory organized packets [61] has been another important influence on our thinking. Script-memory theories, of which Schank is the leading exponent, propose that we remember patterns of experience associated with specific constructs and purposes, e.g., satisfying hunger by eating a meal in a restaurant. Scripts encapsulate objects, actions and events in an expected sequence which achieves a desired goal. Memory is developed by generalization from many specific instances of similar episodes, so we apply a generic script to situations we recognize by their entry conditions. The concordance between the script and events in the real world determine how typical we judge a situation is and our prediction of the appropriate responses to make.

The domain theory OSMs are equivalent to scripts for achieving system's goal. They represent a collection of objects, events, and state transitions which achieve a purpose. Unlike everyday scripts, the domain theory OSMs are deeper abstractions of problems, hence they are unlikely to be so easily recognized as script for more mundane problems e.g., going to the supermarket, car journey, etc.

B.2 Theory Perspectives and Validation

The domain theory's claim to be a natural theory of memory is based on its origins in cognitive science. However, because it is also a theory of abstraction, we do not posit that the majority of people will possess OSM models as described. Such abstractions, which are the basis of deep analogies, are only acquired by learning. Hence the domain theory predicts models of expertise from two viewpoints:

Cognitive science. Naturally occurring but not universal phenomena which are acquired by experts from multiple exposure to related problems in software engineering. Novices should have partial models at lower levels of abstraction.

Validation of this view is by experimentation. Expert designers should be able to recognize problem descriptions as belonging to the classes predicted by the domain theory. Furthermore, they should use the same ontology. The validation experiments with experts have proven a subset of the domain theory models, although important differences were uncovered. This exposed a dual mode categorization, schema based models which agreed with our predictions

and attributional labeling of domains (e.g., safety critical, real time) which did not. A weaker validation uses expert judgment to assess whether the theory has sufficient coverage, i.e., it can account for the diversity of naturally occurring software engineering problems. We have also exposed the domain theory to this test which has uncovered further abstractions, however, over time the OSMs we describe have survived the test of independent judgment.

Computer science. Models of problem abstractions derived by introspection and eclectic surveys of software engineering and other design literatures. In this sense the models are a priori assertions about an ontology for abstraction, and a population of generic models covering an extensive, but not exhaustive range of software engineering problems.

Validation in computer science is by engineering. First the theory should be capable of operationalization in a proof of concept demonstrator, i.e., it should be sufficiently detailed and consistent that a computational implementation can be created which demonstrates its operation. This paper reports the proof of concept via the matcher process which retrieves OSM models appropriate for a set of input facts. Secondly the theory's operational form should have demonstrable utility for improving the process of software engineering, either as a method or software tool. This paper reports development of software tools to improve requirements engineering by applying generic reusable models to critiquing, explanation, and guided fact capture. This is illustrated by the case study. Further validation of utility by usability testing and industrial assessment is part of our future work.

Appendix C — Telos Definitions of the Domain Modeling Language

C.1 Objects

Objects have properties and states and are defined using Telos as:

```
Object in Component, MetaClass end Object
```

Objects are specialized into key objects, agents, or structure objects. Key objects are subject to state transitions and therefore undergo state change. The formal Telos definitions are:

```
Keyobject in Component, MetaClass isA Object with
   attribute
        objectproperty:    Objectproperty
end Keyobject
```

Each Object System Model must contain at least one key object.

C.2 Agents

Agents have properties and methods and therefore accord with the more classic object-oriented view. Agents are specializations of objects which carry out activities, the results of which are manifest as events initiating state transitions. Agents can be typed as human or automated agents to describe a required computer system or a manual operator in a paper-based system:

```
Agent in Component, MetaClass isA Object with
   attribute
        agenttype:      Agenttype
end Agent
```

Agents undertake activity and hence cause behavior in the form of state transitions on objects, defined by:

```
Activity in Component, MetaClass
   attribute
      agent:          Agent;
      triggers:     Event;
      undertakes:   Event;
      reads:          State
end Activity
```

C.3 Structure Objects

Structure objects model approximations to the real world entities which have spatial properties and can contain or possess key objects. The relationship between key and structure objects is equivalent to a state. Object state changes occur with reference to a structure (e.g., an object which is sold and physically moves out of the warehouse structure to the customer structure is logically equivalent to a state transition from in-stock to deleted-from-stock).

```
Structureobject in Component, MetaClass isA Object
end Structureobject
```

C.4 States

OSMs have two means of describing state. Primary states record the containing or possession of objects in structure objects. Secondary states belong to objects independent of structure and describe states such as being reserved or scheduled. The formal definition of a basic state is:

```
State in Component, MetaClass
end State
```

Primary states must be persistent and express containment or possession of key objects in the universe of discourse, for example, a library contains many books. Integrity constraints, expressed in Telos's predicative sublanguage constrain state expressions. The object-relation attribute expresses cardinalities of the relationship between objects in a primary state, e.g., <airspace-many-aircraft>. The formal definition of a primary state specializes the definition of states:

```
Primarystate in Component, MetaClass isA State with
    attribute
       firstobject:      Object;
       secondobject:  Object;
       objectrelation:      Objectrelation
end Primarystate
```

Secondary states belong to objects independent of structure. States are either one- or two-object tuples:

```
Secondarystate in Component, MetaClass with
    attribute
       firstobject:     Object;
       secondobject: Object;
      domainstate:       Staterelation
end Secondarystate
```

The distinction between primary and secondary states is a consequence of Structure objects. These provide a more powerful means of discriminating between model classes than secondary states which have no association with state transition, involving containment or possession.

C.5 Primary State Transitions

OSMs are differentiated by state transitions of objects which represent causal behavior linked to the system purpose. State transitions are specialized to either a primary or secondary state transition, depending on the type of state changed by the transition:

```
Statetransition in Component, MetaClass
end Statetransition
```

A state transition changes the state of one object by transferring its membership between structure objects to achieve desired goal states. Primary state transitions are with respect to structure objects and are central to the knowledge metaschema. They provide the structural association necessary for analogical retrieval.

```
Primarystatetransition in Component, MetaClass is
AStatetransition with
   attribute
      inputobject:          Keyobject;
      inputcardinality:     Cardinality;
      outputobject:         Keyobject;
      outputcardinality:    Cardinality;
      from:                 Primarystate;
      to:                   Primarystate;
      frequency:            Transitionfrequency
   end Primarystatetransition
```

Cardinalities attached to each state transition are usually 1:1, apart from the Object Composition/Decomposition OSM families (see Section 3) which change key objects through compose/decompose transitions and are distinguished by the cardinalities N:n and n:N, respectively, where N > n. Transition frequencies represent the expected occurrence, for instance *normal* transitions such a book returns are likely to occur while *exceptional* transitions such as late or missing returns are less likely.

C.6 Secondary State Transitions

Secondary state transitions change the states of key objects which do not involve any relationships with structure objects. This is defined using the Telos formalism:

```
   Secondarystatetransition in Component, MetaClass isA
Statetransition
   attribute
      keyobject:            Keyobject;
      objectstructure:      Primarystate;
      initialstate:         Secondarystate;
      finalstate:           Secondarystate;
      changeattribute:      Changeattribute
   end Secondarystatetransition
```

The change attribute defines the nature of the object's state change, for example from available to reserved.

C.7 Object Properties

Properties define characteristics of key objects which effect their behavior, allowing three object subtypes to be distinguished:

- physical objects which are subject to physical laws such as gravity or momentum.
- financial objects represent currency or some monetary value.
- conceptual objects representing all key objects which are not physical or financial objects

This has the following definition:

```
Objectproperty in Property, MetaClass
end Objectproperty
```

C.8 Events

Events are defined in two subtypes; domain and time events. Events are implicit in state transitions. To avoid redundancy we only model initiating "triggers" as events, i.e., those events which start state transitions of key objects. For instance, the dispatch of goods from the warehouse to customer is triggered by a customer order (a request event). Domain events model the result of some external activity in the real world while time events are similar phenomena emanating from a clock.

```
   Event in Component, MetaClass with
      attribute
         eventname:          Eventname;
         starts:             Statetransition;
         generatedfrom:      Object;
         eventattribute:     Triggercardinality
   end Event

Domainevent in Component, MetaClass isA Event end
Timeevent in Component, MetaClass isA Event end
```

The event attribute links the event with the state transition it initiates, so the effect of the event can be modeled in terms of the scope of change it initiates. For example, one loan request might trigger hiring of several key objects as in video or book loans or only one object may be hired as in car hire. In this case the event attribute is used to distinguish low value from high value loans. Three types of event trigger cardinality are used; the event triggers one instance of a state transition; triggers many state transition instances from and to the same structure; and triggers many state transition instances from and to different object instances.

C.9 Stative Conditions on State Transitions

Stative preconditions and postconditions are defined for state transitions. Several types of precondition tests are used to discriminate between lower-level OSMs:

- tests on primary states, which must be successful for the state transition to be initiated, for example a book must be in the library before it is loaned;
- tests on secondary states which must be successful for the transition to be initiated, for example a book must not be reserved for another borrower;
- tests on the number of key objects contained or possessed in structure objects, for example stock is replenished when the number of stock objects in the warehouse reaches a minimum-level;

- tests which enable secondary state transitions, for example a theatergoer's request and seats must have matching attributes before the seat can be allocated to the theater-goer.

Each condition is specialized into preconditions and postconditions which must exist before and after a state transition. Stative conditions are defined using Telos's notation:

```
Stativecondition in Component, MetaClass with
    attribute
        tests:        Statetransition;
        condition1:   Primarystate;
        condition2:   Secondarystate;
        condition3:   Expression
end Stativecondition

Precondition in Component, MetaClass isA Stativecondition end
Postcondition in Component, MetaClass isA Stativecondition end
```

C.10 Goal States

Goal states are an orthogonal set of state subtypes describing a future, required state which the system should satisfy. Linguistic statement of goals are refined into more formal expressions of future states which the system should (or should not) attain. Such state expressions link linguistic statements to OSMs in the domain. These types are intended as intermediate modeling constructs to mediate transformation of linguistic expressions towards more formal goal state expressions.

C.11 Semantic Relationships

Semantic relationships add further structural information to OSMs which is used to discriminate between lower-level OSMs. States and state transitions alone are insufficient for distinguishing between all OSMs. Semantic relationships occur between primary domain states and between state transitions. Semantic temporal and scaling relations exist between primary states and ordering relations exist between primary state transitions. The two forms of semantic relation between primary states are:

- temporal relations which indicate that one single key object A persists in one structure object A′ for a significantly longer time than key object B does in structure object B′, where this duration represents the time from the first existence of A in A′ and B in B′. It is defined as A longer B. It represents an occurrence which is typical but does not hold for every instance of the Object System Model;
- scale relations which indicate that a greater number of key objects A are held in structure object A′ than key objects B in structure object B′ at any time t. It is defined as A greater B. Again, this represents an occurrence which is typical but does not hold for every instance of the Object System Model.

Temporal relations express simple properties which a relationship is expected to possess rather than temporal logics expressing system behavior. Thus, an OSM with relationship A longer than B, expresses the expected property that the key object, generally would be expected to spend more time in structure A than B. These are defined using the following Telos formalism:

```
Temporalrelation in Component, MetaClass with
    attribute
        firststate:     Primarystate;
        secondstate:    Primarystate
end Temporalrelation

Scalerelation in Component, MetaClass with
    attribute
        firststate:     Primarystate;
        secondstate:    Primarystate
end Scalerelation
```

Semantic ordering relationships link state transitions to define sequential ordering, so that one transition precedes another in changing the state of the same key object instance. For example, in the lending library, loan of a book instance precedes the return of the same book from the same structure object. This is represented in Telos as:

```
Transitionsequence in Component, MetaClass with
    attribute
        firsttransition:     Statetransition;
        secondtransition:    Statetransition
end Transitionsequence
```

Relationships complete the knowledge metaschema for modeling OSMs.

C.12 Definition of Object System Models

The previous sections have defined knowledge types for modeling OSMs, which are defined in Telos as:

```
ObjectSystemModel in Class, MetaClass with
    attribute
        hasobject:        Object;
        hasstructure:     Primarystate;
        hasstate:         Secondarystate;
        hasbehavior:      Primarystatetransition;
        hasbehavior:      Secondarystatetransition;
        hasquantities:    Temporalrelation;
        hasduration:      Scalerelation;
        hassequence:      Transitionsequence;
        hascondition:     Stativecondition;
        hasevent:         Event;
        hasgoal:          Goalstate
end ObjectSystemModel
```

Integrity constraints control the cardinalities of instances of each knowledge type in OSMs.

ACKNOWLEDGMENTS

This work was partially supported by the European Commission's Esprit program in basic Research Action 6353 NATURE (Novel Approaches to Theory Underlying Requirements Engineering). We thank two anonymous reviewers, whose comments stimulated many improvements to this paper.

REFERENCES

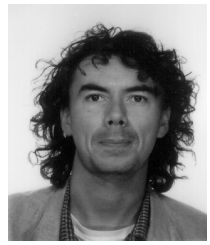
[1] A. Aamodt et al., "The Common KADS Library," Vrieje Universiteit, Brussels, Deliverable KADS-II Project, KADS-II/T1.3/VUB/TR/005, 1992.

[2] J. Allen, "A Common Sense Theory of Time," *Proc. Int'l Joint Conf. Artificial Intelligence*, 1985.

[3] J.R. Anderson, *The Adaptive Character of Thought*. Hillsdale, N.J.: Lawrence Erlbaum Assoc., 1990.
[4] G. Arango, E. Schoen, and R. Pettengill, "Design as Evolution and Reuse," *Proc. Second Int'l Workshop Software Reusability* (Advances in Software Reuse), pp. 9-18, 1993.
[5] B. Chandrasekaran, A. Keuneke, and M. Tanner, "Explanation in Knowledge Systems: The Roles of the Task Structures and Domain Functional Models," *Proc. Workshop Task Based Explanation*, Univ. of the Aegean, Samos, Greece, 1992.
[6] P.W. Cheng and K.J. Holyoak, "Pragmatic Reasoning Schemas," *Cognitive Psychology*, vol. 17, pp. 391-416, 1985.
[7] M.T.H. Chi, R. Glaser, and E. Rees, "Expertise in Problem Solving," *Advances in the Psychology of Human Intelligence*, R. Sternberg, ed., pp. 7-75, 1982.
[8] M.T.H. Chi, M. Bassock, M.W. Lewis, P. Reimann, and R. Glaser, "Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems," *Cognitive Science*, vol. 13, pp. 145-182, 1989.
[9] L. Chung, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach," Research in Data and Knowledge Base Systems, DKBS-TR-93-1, Dept. of Computer Science, Univ. of Toronto, 1993.
[10] P. Coad and E.E. Yourdon, *Object-Oriented Analysis*. New York: Yourdon Press, 1990.
[11] P. Constantopoulos, M. Jarke, J. Mylopoulos, and Y. Vassilou, "Software Information Base: A Server for Reuse," Technical Report, FORTH Research Inst., Univ. of Heraklion, Crete, 1991.
[12] A. Dardenne, A. v. Lamsweerde, and S. Fickas, "Goal Directed Requirements Acquisition," *Science of Computer Programming*, vol. 20, pp. 3-50, 1993.
[13] S. Easterbrook, "Handling Conflict Between Domain Descriptions with Computer-Supported Negotiation," *Knowledge Acquisition*, vol. 3, pp. 255-289, 1991.
[14] B. Falkenhainer, K.D. Forbus, and D. Gentner, "The Structure-Mapping Engine: Algorithm and Examples," *Artificial Intelligence*, no. 41, pp. 1-63, 1989.
[15] C.J. Fillmore, "The Case for Case Reopened," *Syntax and Semantics VIII*, P. Cole and J.M. Sadock, eds. New York: Academic Press, 1977.
[16] C.J. Filmore, P. Kay, and M.C. O'Connor, "Regularity and Idiomaticity in Grammmatic Constructions: The Case of Let Alone Language," vol. 64, pp. 501-538.
[17] G. Fischer, A. Girensohn, K. Nakakoji, and D. Redmiles, "Supporting Software Designers with Integrated Domain-Oriented Design Environments," *IEEE Trans. Software Eng.*, vol. 18, no. 6, pp. 511-522, 1992.
[18] G. Fischer, K. Nakakoji, J. Otswald, G. Stahl, and T. Sumner, "Embedding Computer-Based Critics in the Contexts of Design," *Proc. INTERCHI'93*, S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel, and T. White, eds., pp. 157-163, 1993.
[19] D. Gentner, "Structure-Mapping: A Theoretical Framework for Analogy," *Cognitive Science*, vol. 5, pp. 121-152, 1983.
[20] T.R.G. Green and R. Navarro, "Programming Plans, Imagery and Visual Programming," *Proc. Human Computer Interaction—INTERACT'95*, K. Nordby, P.H. Helersen, D. Gilmore, S.A. Arnesen, eds., London: Chapman and Hall, 1995.
[21] R. Greiner, "Learning by Understanding Analogies," *Artificial Intelligence*, vol. 5, pp. 81-125, 1988.
[22] T.R. Gruber, "Towards Principles for the Design of Ontologies Used for Knowledge Sharing," Knowledge Systems Laboratory Report, KSL 93-04, Dept. of Computer Science, Stanford Univ., 1993.
[23] T.R. Gruber, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition*, vol. 5, pp. 199-220, 1993.
[24] R. Guindon, "Designing the Design Process: Exploiting Opportunistic Thoughts," *Human-Computer Interaction*, no. 5, pp. 305-344, 1990.
[25] J.A. Hampton, "Disjunction in Natural Categories," *Memory and Cognition*, vol. 16, pp. 579-591, 1988.
[26] M.T. Harandi and M.Y. Lee, "Acquiring Software Design Schemas: A Machine Learning Perspective," *Proc. Sixth Conf. Knowledge Based Software Engineering*, Syracuse, N.Y., vol. 1, pp. 239-250, Sept. 1991.
[27] K.J. Holyoak and P. Thagard, "Analogical Mapping by Constraint Satisfaction," *Cognitive Science*, vol. 13, pp. 295-355, 1989.
[28] M.J. Jackson, *Systems Development*. Prentice Hall, 1983.
[29] L. Jacobson, *Object-Oriented Software Engineering: A User Case-Driven Approach*. ACM Press, 1992.
[30] M. Jarke, J. Mylopoulos, J. Schmidt, and Y. Vassilou, "DIADA—An Environment for Evolving Information Systems," *ACM Trans. on Information Systems*, vol. 10, pp. 1-50, 1992.
[31] M. Jarke, "ConceptBase V3.1 User Manual," Aachen, Germany, RWTH-Aachen, 1992.
[32] M. Jarke, Y. Bubenko, C. Rolland, A.G. Sutcliffe, and Y. Vassilou, "Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis," *Proc. IEEE Symp. Requirements Engineering*, pp. 19-31, 1993.
[33] M. Jarke et al., "Requirements Engineering: An Integrated View of Representation," *Proc. Fourth European Software Engineering Conf.*, Garmish-Partenkirchen, Sept. 1993.
[34] P. Johnson, H. Johnson, R. Waddington, and R. Shouls, "Task-Related Knowledge Structures: Analysis, Modeling and Application," D.M. Jones and R. Winder, eds., *HCI'88*, pp. 35-61. Cambridge: Cambridge Univ. Press, 1988.
[35] G. Lakoff, *Women, Fire and Dangerous Things: What Categories Reveal about the Mind*. Chicago: Univ. of Chicago Press, 1987.
[36] J.C.S.P. Leite and P.A. Freeman, "Requirements Validation Through Viewpoint Resolution," *IEEE Trans. Software Eng.*, vol. 17, no. 12, pp. 1,253-1,269, 1991.
[37] P. Loucopoulos, G. Papamastatiou, D. Pantazis, and G. Diakonolaou, "Design and Execution of Event/Action DB Applications," *Proc. Second Int'l Workshop Deductive Approach to Information Systems and Databases*, Aiguablava, Spain, 1991.
[38] N.A.M. Maiden and A.G. Sutcliffe, "Analogical Matching for Specification Retrieval," *Proc. Sixth Knowledge-Based Software Engineering Conf.*, pp. 108-116, 1991.
[39] N.A.M. Maiden and A.G. Sutcliffe, "Exploiting Reusable Specifications Through Analogy," *Comm. ACM*, vol. 35, no. 4, pp. 55-64, 1992.
[40] N.A.M. Maiden and A.G. Sutcliffe, "Analogical Retrieval in Reuse-Oriented Requirements Engineering," *Software Engineering J.*, vol. 11, no. 5, pp. 281-292, 1996.
[41] N.A.M. Maiden and A.G. Sutcliffe, "Requirements Engineering by Example: An Empirical Study," *Proc. IEEE Symp. Requirements Engineering RE-93*, E. Finkelstein A.C. IEEE CS Press, 1993.
[42] N.A.M. Maiden, P. Mistry, and A.G. Sutcliffe, "How People Categorise Requirements for Reuse: A Natural Approach," *Proc. Second Int'l Symp. Requirements Engineering RE'95*, P. Zave and M.D. Harrison, eds., pp. 148-157. IEEE CS Press, 1995.
[43] S.M. McMenami and J.F. Palmer, *Essential Systems Analysis*. Yourdon Press, 1984.
[44] B. Meyer, "On Formalism in Specifications," *IEEE Software*, pp. 6-26, Jan. 1985.
[45] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis, "Telos: Representing Knowledge about Information Systems," *ACM Trans. Office Information Systems*, vol. 8, no. 4, 1990.
[46] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," *IEEE Trans. Software Eng.*, vol. 18, no. 6, pp. 483-497, 1992.
[47] J.M. Neighbors, "Software Construction Using Components," PhD thesis, Dept. of Information and Computer Science, Univ. of California, Irvine, 1980.
[48] N. Pennington, "Stimulus Structures and Mental Representation in Expert Comprehension of Computer Programs," *Cognitive Psychology*, vol. 19, pp. 295-341, 1987.
[49] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, pp. 6-16, Jan. 1987.
[50] R. Prieto-Diaz, "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, vol. 15, no. 2, pp. 47-54, 1990.
[51] R. Prieto-Diaz, "Implementing Faceted Classification for Software Reuse," *Comm. ACM*, vol. 34, no. 5, pp. 88-97, 1991.
[52] P.P. Puncello, "ASPIS: A Knowledge-Based CASE Environment," *IEEE Software*, pp. 58-65, Mar. 1988.
[53] M.R. Quillian, *Semantic Memory*. Cambridge, Mass.: Bolt, Beranak and Newman, 1966.
[54] B. Ramesh and V. Dhar, "Supporting Systems Development by Capturing Deliberations During Requirements Engineering," *IEEE Trans. Software Eng.*, vol. 18, no. 6, pp. 498-510, 1992.
[55] H.B. Reubenstein and R.C. Waters, "The Requirements Apprentice: An Initial Scenario," *Proc. Fifth Int'l Workshop Software Specification and Design*, Pittsburgh, pp. 211-218, May 1989.
[56] C.K. Riesbeck and R.C. Schank, *Inside Case-Based Reasoning*. Hillsdale, N.J.: Lawrence Erlbaum Assoc., 1989.

[57] G. Roman, "A Taxonomy of Current Issues in Requirements Engineering," *Computer*, pp. 14-22, Apr. 1985.
[58] E. Rosch, C.B. Mervis, W.D. Grey, D.M. Johnson, and P. Boyes-Braem, *Basic Objects in Natural Categories*. Academic Press, 1976.
[59] E. Rosch, "Prototype Classification and Logical Classification: The Two Systems," *New Trends in Conceptual Representation: Challenges to Piaget's Theory?* E.K. Scholnick, ed., 1985.
[60] D.E. Rumelhart, "Notes on a Schema for Stories," *Representation and Understanding*, D.G. Bobrow and A.M. Collins, eds., 1976.
[61] R.C. Schank, *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge Univ. Press, 1982.
[62] M. Shaw, "Heterogeneous Design Idioms for Software Architecture," *Proc. Sixth Int'l Workshop Software Specification and Design*, pp. 158-165, 1991.
[63] D.R. Smith, "KIDS: A Semi-Automated Program Development System," *IEEE Trans. Software Eng.*, vol. 16, no. 9, pp. 1,024-1,043, 1990.
[64] D.R. Smith, "Track Assignment in an Air Traffic Control System: A Rational Reconstruction of System Design," *Proc. KBSE'92, Knowledge Based Software Engineering*, pp. 60-68, 1992.
[65] J.F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, 1984.
[66] A.G. Sutcliffe and N.A.M. Maiden, "Software Reusability: Delivering Productivity Gains or Short Cuts," *Proc. INTERACT'90,* D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, eds., pp. 895-901, 1990.
[67] A.G. Sutcliffe and N.A.M. Maiden, "Supporting Component Matching for Software Reuse," *Proc. Fourth Conf. Advanced Information Software Engineering CAiSE'92*, M.P. Loucopoulos, ed., pp. 290-303, 1992.
[68] A.G. Sutcliffe and N.A.M. Maiden, "Bridging the Requirements Gap: Policies, Goals and Domains," *Proc. Seventh Int'l Workshop System Specification and Design*, pp. 52-55, 1993.
[69] C.N. Taylor, A.G. Sutcliffe, N.A.M. Maiden, and D. Till, "Formal Representations for Domain Knowledge," Technical Report, 95/5, Centre for HCI Design, School of Informatics, City Univ., London, 1995.
[70] B.J. Wielinga, A.T. Schreiber, and J.A. Breuker, "KADS: A Modeling Approach to Knowledge Engineering," Technical Report ESPRIT Project P5248 KADS-II, 1991.
[71] R. Wirfs-Brook, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*. Prentice Hall, 1990.
[72] E.S.K. Yu, "Modeling Organizations for Information Systems Requirements Engineering," *Proc. IEEE Symp. Requirements Engineering* RE-93, San Diego, Calif., A.C.W. Finkelstein, ed., pp. 34-41, 1993.

Alistair Sutcliffe is professor of systems engineering and director of the Centre for Human-Computer Interface Design at City University, London. He has research interests in software engineering and human computer Interaction. His software engineering research focuses on cognitive modeling within the software engineering process and investigation of intelligent assistance for requirements capture. He has extensive publications in human-computer interaction and software engineering and has written two textbooks. He is a member of the British Computer Society, a member of the IEEE, the ACM, and chair of IFIP working group 13.2 on methodology for user-centered design.



Neil Maiden received his doctorate for research into analogical reuse of specifications during requirements engineering in 1992. Maiden is a lecturer in the Centre for Human-Computer Interface Design at City University, London. His research interests include requirements and software reuse, requirements acquisition, IT procurement, scenario-based systems development, and computational models of analogical reasoning. He is a member of the IEEE, and the IEEE Computer Society; is an associate member of the British Computer Society (BCS); and is treasurer and co-founder of the BCS Requirements Engineering Specialist Group.