Proceedings of IFIP WG8.1 Conference on Information System Development Process, Como Italy, 1-3 September 1993, Elsevier North-Holland.

Use of Domain Knowledge for Requirements Validation

A.G. Sutcliffe and N.A.M. Maiden

Department of Business Computing, City University, Northampton Square,
London EC1V 0HB,United Kingdom.

This paper reports reuse of generic domain knowledge in the form of templates, patterns or clichés to aid requirements capture and validation. These templates are retrieved through machine-based analogical matching. A cooperative paradigm for domain knowledge reuse is proposed. Human interpretation of templates is needed to maximise benefits from knowledge reuse, necessitating explanation and exploration of domain templates.

Keyword Codes: D.2.1
Keywords: Requirements/Specifications

1. INTRODUCTION

Domain knowledge is acknowledged to be important for the acquisition and validation of requirements specification. Unfortunately this is one of the most error prone and costly activities in system development which has led a number of researchers to investigate how the process of requirements capture and validation may be improved.

Previous approaches have varied from development of powerful specification languages [14], to investigation of declarative, rule based paradigms [10] and embedding domain knowledge in CASE tools to allow intelligent requirements validation [15]. While more powerful languages can help expression of requirements and verification of specifications, they do not in themselves help validation. Embedded domain knowledge can give better leverage on validation but at a considerable cost in acquiring and representing such knowledge in a machine processable form. It is doubtful whether tools with embedded domain knowledge could ever keep pace with the development of new systems in changing domains.

A possible solution to the domain knowledge acquisition dilemma is to use more generic domain knowledge in the form of templates, patterns, or cliches. This approach has been primarily explored in software reuse [6, 16], and could be applied to requirements validation, although matching of appropriate generic templates to new applications is problematic. Concepts of analogy based matching, also employed in specification reuse [11, 20] could be used to solve this problem.

This paper extends our work in analogical matching for specification reuse, by extending models of domains into a wider theory for requirements capture and validation. A preliminary theory of domain knowledge is proposed which describes abstract domain

classes, rules which discriminate between domain classes and a matching process which takes descriptions of new applications and finds their generic domain class. Requirements of a new application are then validated against a model of the domain class. The paper is organised into four sections. First the domain theory is presented then the matching process is described. This is followed by use of the domain models for a case study in requirements validation and the paper concludes with a brief discussion.

2. A PRELIMINARY THEORY OF DOMAIN KNOWLEDGE

The assumptions that the theory rests upon are drawn from cognitive models of memory, in particular the hierarchical models of natural categories [17], hierarchical memory schemata [1] and categories of dynamic memory [21]. These assert in slightly different forms that human memory is organised in an informal hierarchy of classes. Generally there is good evidence that memory for several different types of knowledge (e.g. objects, procedures and plans) is organised hierarchically and applied theories of knowledge representation embody classification structures (e.g. Task Knowledge Structures [9]). We propose that domain abstractions are modelled as classes which are specialised by addition of further knowledge to achieve appropriate targeting in terms of generalisation/specialisation. This leads to the basic propositions:

- domain knowledge is organised into a hierarchy of models, each domain model is defined by a finite set of shared features. Each class level adds more specialised features;
- domain knowledge is aggregated by association with a problem that exists in the universe of discourse, that is some state in a system which should be changed;
- differences between domain classes can be distinguished by a small number of key determining features;
- most software engineering problems can be ascribed to one of a tractably small set of domain models.

The first distinction the theory makes is between object models and information system models:

- object models are the basic processing of the system composed of objects, actions, and constraints. Objects models describe what we more readily call transaction processing;
- information system models describe processing which is itself dependent upon the existence of the object model. Information systems feed off the object model and are more familiar as report-generating sub systems or online queries.

Both models describe problem spaces, that is collections of knowledge which relate to a system requiring change. Problems are conceived as a future state of affairs that the user wants the system to achieve. Initially these may be linguistically expressed functional requirements. As analysis progresses ambiguous expression in language can be more formally expressed as description of system behaviour and the desired (solution) state of system objects. For instance ‘the library should control book loans’ is expressed as an invariant that each book which is loaned must be returned. This approach is similar to ideas employed on

the KAOS project [3]. The problem is therefore the domain's rasion d'etre. From this we hypothesise that the fundamental aspects of domain are objects and transitions to achieve change from initial (problem) to final (solution) states. To this we add the components which are fundamental to modelling object transitions. These we term system structures which represent the physical or abstract sets containing objects. These predictions accord with our empirical knowledge of systems analysts [20], cognitive studies of problems solving in other domains and the fact gathering strategies employed in structured system development methods. The basic semantics of the domain are therefore structure, objects, transitions with respect to the structure and states to be achieved.

The basic models are augmented with further knowledge about the environment, e.g. organisational knowledge and causal reasons for behaviour in the domain. This leads to two further models:

- an organisation framework describes the superstructure within which domain models exist;
- models describing the casual relationships between objects and structures in the domain model in order to explain their behaviour. Causal relationships can also explain how the system relates to its purpose, i.e. how behaviour is linked to the system's goal state.

The organisational framework gives a means of structuring knowledge for validation, although such knowledge will probably require human interpretation. A payoff will have to be established between the effort of recording such knowledge and possible benefit in validation, and this is one focus of our ongoing work. A further justification may, however, be in explanation. We believe that an important part of requirements engineering is understanding the application by reference to its immediate context and by reference to similar contexts. The organisation framework could prove invaluable as a explanatory setting linking a problem-oriented domain to an organisation, its place within the organisation structure, and relevance to organisation policy.

Explanation prompts the final component of the theory, capture of causal knowledge about the problem domain. Experience with abstract domain models has shown that while people can understand them [12], there was a need for explanation about the object structure and for linking goals with the problem abstraction. This suggests a validation role for causal knowledge about 'why things happen as they do'. Inspiration for this work comes from the concept of causal stories in knowledge engineering [2]. The need is for explanation of the observed transitions of objects and laws governing them. For example in air traffic control causal explanation would provide knowledge about structural components (air corridors, manoeuvring areas, airports, flight paths), objects and their properties (types of aircraft, flight speeds, manoeuvring capabilities) rules governing transitions (direction, height, speed, instructions) and system invariants (separation distance). The corresponding abstract causal model describes spatial object management with objects following paths according to rules governing movement and separation. A generic model of spatial object management can be applied several applications in its class (e.g. air traffic control, ship navigation control, railway signalling, etc.). Causal knowledge is represented in informal semantic networks to help tracing of causal links. This part of the theory is the subject of further investigation to look at the stopping problem, i.e. at which point is the causal story sufficient for explanation. For instance, does the system need basic physical laws about aero dynamics ? Again there

will be a trade off between the effort of capture and benefit of understanding for reuse and explanation for validation.

To summarise the theory as it currently stands has three main components as shown in Figure 1:

- a set of abstract domain problem classes, a meta-schema which defines the modelling language and rules for determining class membership. These form the object and information system models of the application;
- the organisation framework;
- causal models for explanation of abstract problem classes.

Constraints of space in this paper preclude further description of the whole theory. Instead we shall concentrate on the domain theory and its matching process, and attempt to demonstrate how validation fits into this component. We shall return briefly to the wider picture in the discussion by suggesting how further validation and explanation can be delivered by using other components by virtue of their links to the abstract domain models. The components of the theory are a meta-schema which defines the modelling language for abstract domain models, an initial set of domain models and rules for matching facts describing application problems to one or more domain models. The rules have been developed as a computational matching process. Each component is elaborated in the following sections.

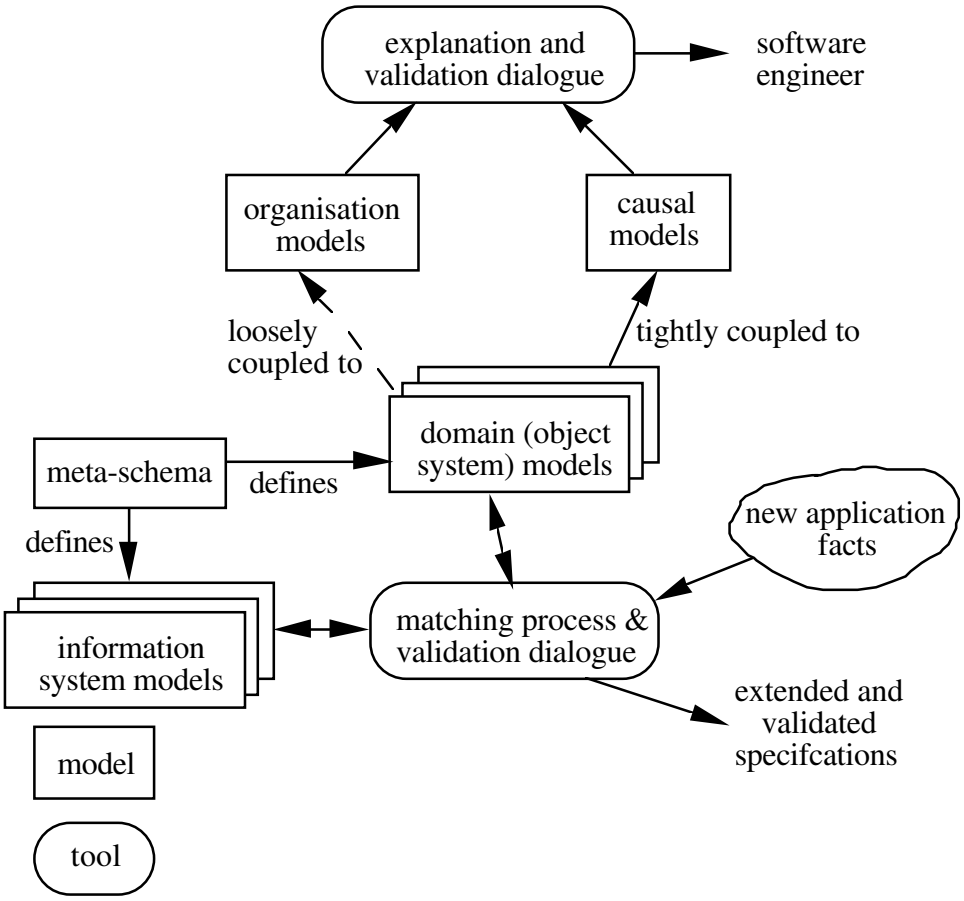

Figure 1. Components of the domain theory and their interaction.

2.1. Meta-Schema of Domain Knowledge

The meta schema describes our modelling language. We hypothesise that seven key knowledge determine software engineering domains:

- objects. These are the prime modelling primitive which represent constructs of interest in the universe of discourse;
- the structure of the system. This is knowledge of the objects within the systems and the sets they naturally fall into. The prediction is that people will perceive sets which pertain to physical structures in the real world. Hence the term ‘containment’ may be a familiar cue for system structure, e.g. stock objects-contained-in warehouse; students are-

contained-in a school. The system structure describes the object sets implicated in input and output (i.e. changes in object status across the system boundary) and any major physical entity sets within the system itself. Most of the system structure should be apparent from an entity relationship diagram;

- state transitions which cause objects to change set membership with respect to the system structure. Inter-domain class differences are critically determined by these state transitions more than any other feature;
- events which trigger state transitions. Internal and external events (with an origin outside the system) are recognised. This enables description of the scope or boundaries of the system;
- object properties. High level properties of an object, focusing on roles within the system (e.g. resource, mechanism), movement, and other high level abstract or concrete qualities. Properties are essentially a type definition of objects, which although currently not formalised, are amenable to such treatment;
- requirements/purpose. This is the goal or solution state that the system exists to achieve goals and statements of purpose often differ between users and may be expressed with different degrees of precision, more formal definition is in terms of future states which the system must attain;
- conditions. Stative information or values held in attributes of an object and other non-object states (e.g. time, duration) which are evaluated before a state transition can occur.

Users are predicted to use requirements and purpose as the most 'natural' descriptors of domains, hence these features will be more easy to elicit. However, given the ambiguity inherent in many teleological descriptions, system structure and state transitions are predicted to be the most reliable determinants of domains. A composite of system structure and related transitions could be taken as the domain/class 'key' in database terms.

Another modelling component, although not part of the meta-schema, is the transformation. These are set of actions leading to a state change of system objects within a single procedure. In other words transformations can not be halted midway and all the operations must execute to change the object's state. Transformations may result in state transitions in set membership in the system structure. To illustrate the concept, common transformations in information systems are scheduling, allocation by constraint satisfaction, searching for objects, reservation, etc.. These result in conceptual-stative changes to objects such as 'reserved, found, sorted, scheduled.' Transformations are similar to 'methods' in object oriented specifications or procedures in structured methods.

The relationship between meta-schema components is shown as an informal entity relationship diagram in Figure 2. Space limitations preclude discussion of the many of the schema primitives which elaborate the above principal modelling components. Most of the schema primitives are familiar from the modelling languages of structured developments methods (e.g. JSD, NIAM, SADT, etc.). The additions made by the domain theory is to emphasise system structure as a modelling concept, to link system structure to system purpose and differentiate between object and information system models.

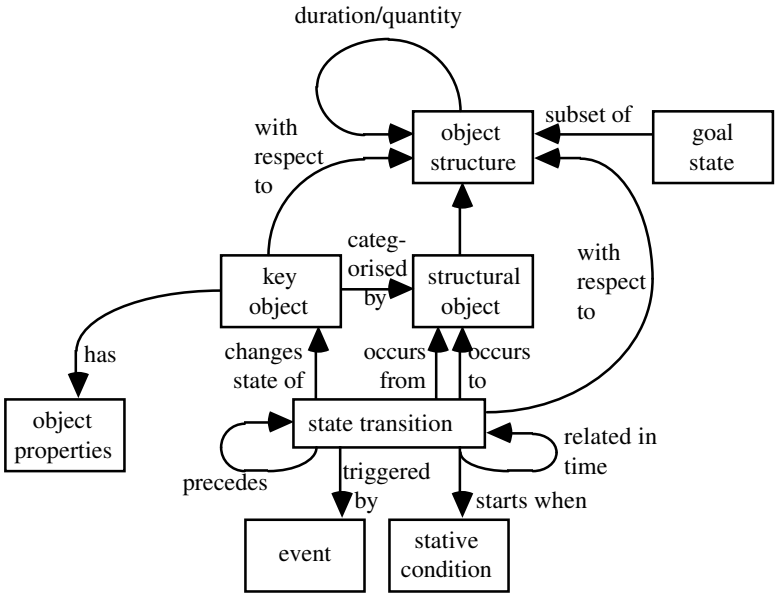

Figure 2. Domain theory meta-schema.

2.2. Matching applications to domain models

The abstract domain classes for object hiring and object containment share similarities in their basic system structure however they are differentiated by the key state transition of a return from the client to the system.

The resource hiring abstraction may appear to fit a standard stock control problem in which stock items are matched to customers orders, and stock is replenished; however two key transitions differentiate resource hiring from stock control. First, in stock control objects are dispatched and the majority never return and new-resources arrive. A return in the library changes an object's state from on-loan to in library, whereas this transition is not common in stock control, return of damaged stock non withstanding. In contrast, some actions do not discriminate between abstract classes; for instance, reservation of objects can lead to a status change in either abstraction.

The architecture and algorithms of the domain matcher are described in three parts. First the components of the matcher are described, then the matching algorithms are followed by the role of the domain matcher in an example. The aim of the domain matcher is to determine a degree of similarity between a target domain description and one or more candidate domain abstractions, then use inferred fact mappings to identify equivalences between objects in the domain and its abstraction.

2.3. Architecture of the domain matcher

The matcher's architecture shown in Figure 3 has five components, namely a match controller, structure matcher, rule-based matcher, domain discriminator and computational analogical matcher. These components access three knowledge sources, the target domain description, domain abstractions and domain discrimination rules. The target domain description represents facts about the new application while domain abstractions and discrimination rules are defined by the domain theory.

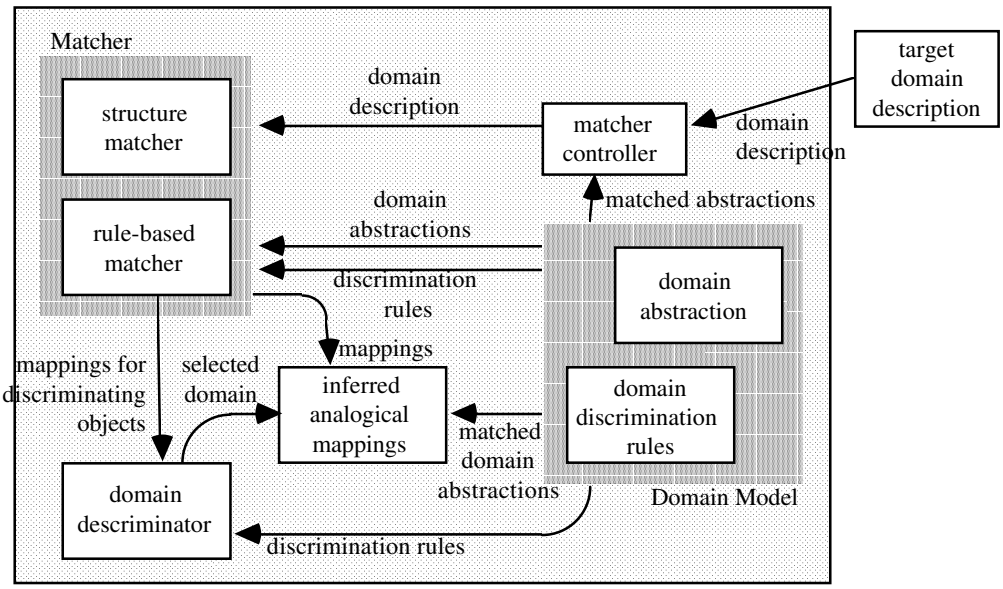

Figure 3. Architecture of the domain matcher.

The match controller decides how a target description is mapped to domain abstractions. It sends this description to the structure or rule-based matcher depending on the current state of the match. Structure matching identifies mappings between state transitions and object structures while other knowledge types are mapped by the rule-based matcher. This distinction is dictated by the knowledge structure defined in the domain theory. State transitions in respect to object structures are central to the theory and are matched structurally because of the inter-relatedness of such facts.

The structure matcher uses an analogical matching paradigm [22] to match object structures and state transitions and outputs a quantified match between input facts and each abstract domain model. Rule based matching calculates the degree of mapping between facts not related to the system structure and extend an existing match inferred by the structure matcher. First, events and stative conditions for state transitions are matched, then actions, then domain requirements. The domain discriminator reasons heuristically about key differences between two or more candidate domain models using rules defined by the domain theory. The quantified result of this reasoning discriminates between borderline matches.

Matching between a domain description and candidate abstractions at one level in the domain specialisation hierarchy is achieved in five phases:

phase 1: examine candidate type mappings for the likelihood of a structure or rule-based match. The total of type mappings defines the potential of a match and acts as a filter to avoid unnecessary and time-consuming computational matching;

phase 2: rule-based or structure matching for each object and fact needed to infer key differences between two or more candidate abstractions;

phase 3: reasoning about key differences between candidate abstractions using the object and fact mappings inferred in the previous stage;

phase 4: further rule-based matching between the domain description and the selected abstraction to calculate the full extent of the match with the abstraction;

phase 5: comprehensive structure matching between the domain description and each candidate domain abstraction.

Operation of the matching process is best described in the following example scenario which demonstrates both structure and rule-based matching. Structure matching is used to infer an initial degree of match with a candidate abstraction while rule-based matching refines the match to specialisations of the abstract domain class. The algorithm maps pairs of facts which belong to a coherent knowledge structure in preference to those which do not. The matching process is repeated for each application fact until a degree of match between it and each candidate abstraction has been calculated. Resulting mappings between objects are calculated from the total scores of fact mappings which include these object pairs in equivalent positions.

In the first phase of matching, the structure matcher attempts to map a target application description to the object containment domain shown in Figure 4. The structure matcher infers an interrelated knowledge structure shared by the application and top-level abstractions, in this case the OC abstraction. Details of the matching mechanism are given in [22]. As the software engineer enters further facts, the domain matcher matches these facts against the lower level OM and OR abstractions. The domain theory dictates that the OM and OR abstractions are differentiated by state transitions with respect to object structures, and defines two discriminating rules between the OM and OR abstractions. Each rule defines one difference between two abstractions, a difference being constrained to the inclusion of one or many instances of a fact type in domain abstraction but not in the other.

Rule-based matching in next stage validates the agreed match with the OM abstraction. It takes new application facts (events, conditions and domain requirements) and matches them to the abstract models in a predefined order dictated by the domain theory, e.g. events, conditions, etc. Rule-based matching reveals several omissions from the stock control description, for instance, there are no facts describing domain requirements or situations. Previous structure and local mappings are used to infer the omitted facts and to produce validation guidance for the software engineer.

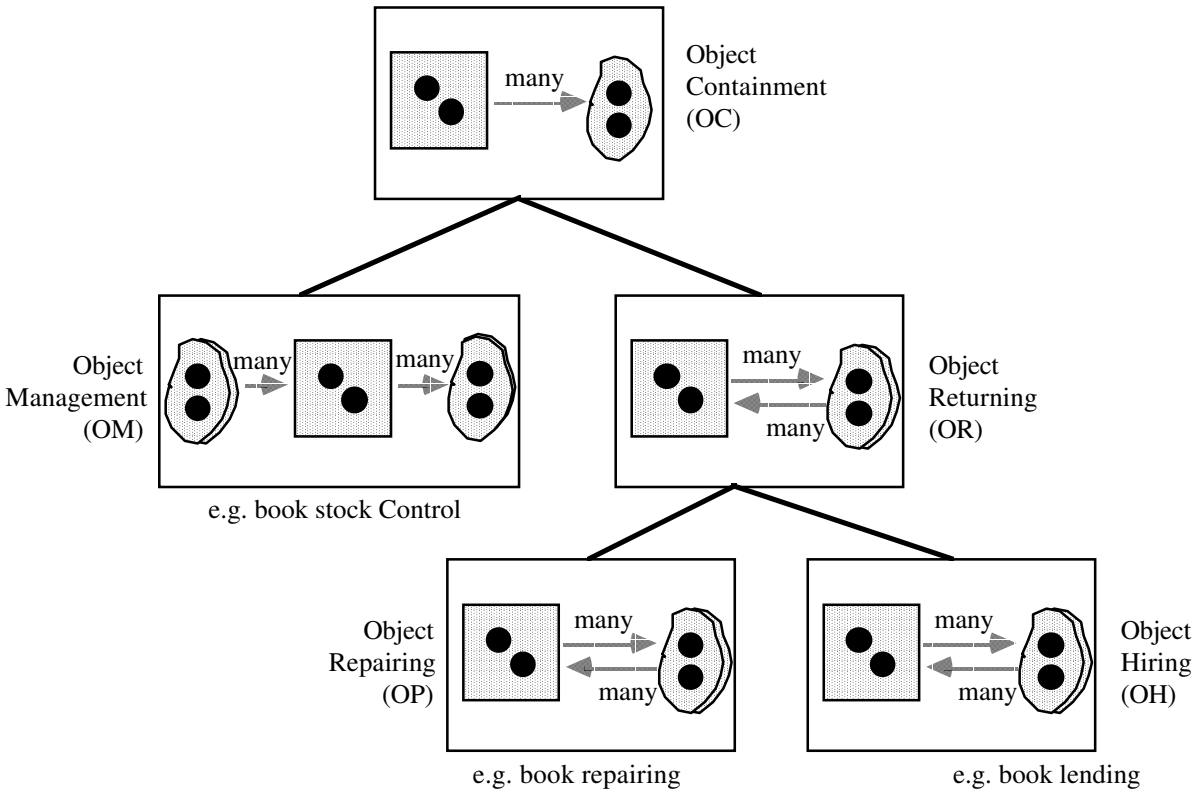

Figure 4. Example specialisation hierarchy of domain abstractions, showing specialisation of basic object containment abstraction through addition of state transitions, object structures, and events to distinguish the object repairing and object hiring abstractions.

3. VALIDATION AND THE DOMAIN THEORY

The intention of the following scenario is to demonstrate how the domain theory assists derivation of a more complete and consistent requirements specification. Validation is envisaged as a cooperative task in which the machine matches facts describing a new application to existing generic models of domains and then provides assistance by pointing to likely omissions, inconsistencies and contradictions to the software developer. The scenario also reveals how the theory may help move from an informal to formal representation of requirements. Informal requirements are usually represented as narrative descriptions. The system permits requirements to be entered as a formal representation using frames describing knowledge types, similar to those used in a previous prototype developed for reusable component matching (Maiden & Sutcliffe 1992).

The scenario examines the need for stock control capabilities within a library and illustrates the transition from an informal and incomplete statement of stock control requirements to a more formal and complete domain description. This is achieved by:

- using the meta-schema of domain knowledge types to capture key facts as formal domain predicates;

- matching this domain description to known domain classes represented as abstract conceptual models;
- validating and guiding the domain description in light of agreed matches with these abstractions.

The scenario illustrates the domain matcher component which is implemented in Prolog. However, some of the fact capture dialogue is simulated in the current version of the prototype because we intend carrying further usability testing on the user interface before fully implementing the fact acquisition and validator component. A revised implementation is in progress using TELOS [14], an object oriented deductive database language, and its implemented form the ConceptBase [8]. The classification structure of Telos allows meta-classes hence the domain theory modelling language is implemented using the class constructs provided. The TELOS implementation allows inferences to made about facts entered as queries and domain models held in the database.

The domain matcher can be used in two different modes of cooperation:

(i) the user enters application facts and the system matches these against its domain models and selects an appropriate class. This is presented to the user who can then validate their understanding of the domain against that of the system. Further validation by extension of the specification may be possible by the system bringing further related facts to the the user attention;

(ii) the user first selects an abstract domain model and then enters application facts. In this case the system acts as a validator on fact capture. If the entered facts do not conform to the abstraction selected by the user then the system rejects them.

These two modes can be used interchangeably so the software engineers can make assumptions about the nature of the problem and ask the system to confirm them or alternatively ask the system to select the appropriate domain class. In both modes the system can import further knowledge from linked domain models to help validation and augmentation of the specification.

3.1. Initial requirements capture: the incomplete problem statement

The problem statement defines the required system functionality and describes what happens in during stock control in the library domain. This problem statement is both incomplete and overspecified (c.f. Meyer 1985) to demonstrate typical problems encountered when refining the requirements specification:

> *The computerised library lending system is to be extended by a stock control facility with the following needs:*
>
> *• to identify missing or damaged books which are no longer in the library;*
> *• to permit a stock take of books within parts of the library.*
>
> *Limitations on this functionality include:*
>
> *• the system must support both automated and staff identification of missing books;*
> *• books cannot be purchased without staff authorisation entered to the system.*

In the current library stock control books can be on the library shelves to be lent, in the archive store or with borrowers. Damaged or out-of-date books are taken from the library to the archives store. Missing books are assumed to be stolen by borrowers. There is only library and one archive store. It is known that some borrowers steal books.

3.2. Transformation of informal to formal requirements statement

The aim of this first stage is to capture informal requirements as semi-formalised domain descriptions. This is achieved using requirement definition frames. Each frame provides a restricted set of terms for defining the application. Terms are defined by the meta-schema of domain knowledge types and individual tokens in the system lexicon.

Figure 5. Dialogue screen showing fact capture for state transitions.

Figure 5 illustrates the first stage when the user enters key state transitions with respect to object structures. Fact capture is aided by description of state transitions and explanation by analogical examples. The initial problem statement is incomplete because it does not mention stock replenishment actions. Fact entry at each stage is validated by matching against candidate domain models and then suggesting possible omissions and inconsistencies to be repaired by the user. Key fact types are captured early in the dialogue to identify the most important requirements and assist problem scoping, two tasks known to be difficult for less experienced software engineers [23]. The dialogue actively encourages better domain understanding by presenting examples, partial views of abstract domain model and new fact types linked to the requirement engineer's current mental model of the domain.

The interface has four parts. A definition of state transitions uses both narrative and spatial diagrams. Such definitions are supported by presentation of concrete examples of state transitions. A frame permits entry of the state transition from options. Finally a TELOS definition of existing library objects is presented to avoid duplication domain description**.**

3.3. Retrieve and present appropriate domain abstraction

The partial domain description is sufficient to match the Object Containment (OC) abstraction since it describes depletion of books from the library but not their renewal. This inferred match is explained to the requirements engineer for checking and confirmation, see Figure 6. This explanation can be in four parts. First, a formal Telos definition of the abstraction is supported by narrative explanation and spatial diagrams representing the domain. Familiar concrete examples of the OC abstraction can also assist comprehension. A fourth window instructs the requirements engineer to select or reject the OC abstraction as a partial match for the stock control domain.

3.4. Further fact acquisition

The tool explains that the current application should be specialised to either an Object Management (OM) or Object Resource (OR) domain. The two abstractions are represented using spatial diagrams which highlight key state transitions with respect to object structures. Selection of the OM abstraction permits the tool to provide further hints for extending the stock control domain description. These hints are controlled to ensure that the domain description is extended according to the domain theory. The requirements engineer is encouraged to elaborate the stock control description in a specific order:

- state transition linked to the replenish action;
- object structures describing the supplier concept;
- events and stative conditions controlling the state transition;
- elaboration of action definitions linked to the key transition;
- situations and domain requirements.

The dialogue is sufficiently robust to identify omission of fact in the entry order and return to the requirements engineer for such facts. As in earlier stages, fact acquisition is supported by explanation of the matched domain abstraction, intelligent hints based on a match with the abstraction.

Figure 6. Explanation of the retrieved OC abstraction

3.5 Validating and extending the domain description

The domain matcher uses rule-based matching with the OM abstraction to validate the application description, revealing several omissions from the description. These omissions provide the basis for guiding the requirements engineer to extend the stock control description in the next stage. Figure 7 demonstrates inference of a possible omission of a domain requirement from the stock control description and the subsequent dialogue encouraging entry of the required fact(s). The result of this final stage is a more complete and consistent domain description for the library's stock control domain according to key facts in the domain theory.

Validation is currently driven from the structure of abstract domain models contained in the domain theory. Hence if the tool expects a certain transition to be present in an abstract domain classes and this fact has not been entered, it suggests there may be an omission to the user. Further validation is driven from the type structure and meta-schema, hence the tool prompts for conditions to be added to state transitions. So far validation rules perform structure and type checking which enables at least limited semantic validation of requirements and specification. However more can be achieved and in the final section we return to the causal and organisation models.

4. FUTURE WORK ON VALIDATION

Once a domain class has been recognised by the matcher further stored knowledge can be employed for validation purposes. First, the causal model, which is tightly coupled to the domain model, can be used to explain certain causal effects on system behaviour and thereby helping development of requirements. For instance in the library domain the link between fines and the effect of causing users to return books on time can be explained or the causal relationship between authorisation for purchase of new books and the necessary discipline in accounting procedures. Causal models can be used in two ways, as a repository of wisdom about the reasons of domain behaviour, and secondly as descriptions about the behaviour of systems given their relationship to physical, biological, or social laws. Causal models therefore represent knowledge which may be useful for determining requirements in a system and for understanding why systems behave as they do.

Figure 7. Validation of the domain description, revealing a possible omission which the requirement engineer is requested to reason about and enter if correct.

Organisation models are less closely coupled. The same domain model may be found in several different organisation environments, so organisation models can only be used as general advice. Some tighter coupling may be possible when organisation types can be identified as reliably co-occuring with specific domain types, as may be found in command and control systems in military organisations. However, we generally consider organisation models to be a 'framework of thought' within which we intend to develop further work extending the concept of goal backwards from the domain models towards business goals and organisational policy.

Finally, information systems model are amenable to the same matching and validation process as the abstract domain models for the object system. We are developing a set of abstractions of typical information systems needs in line with the philosophy of our theory, hence the definition of the model is linked to its purpose. In information systems models the purposes at the abstract level are progress tracking, status checking, and valuation; in more concrete terms these are reporting functions such as age of debt reports, stock status, and financial cost control.

5. CONCLUSIONS

The theory and description of domain classes is still in its preliminary stages. So far 40 abstract problem domain classes have been described ranging from object hiring (e.g. library, car hire, video hire) to spatial object management (e.g. air traffic control). Each class has approximately three levels of specialisation. However several problems which remain to be solved:

(i) the granularity between levels in one class;

(ii) the naturalness and usability of key features in determining the identity of domains by users;
(iii) goodness of fit of natural problems into domain classes;
(iv) formal properties of domain abstractions and how they related to design abstractions.

So far we have some preliminary, encouraging results on the second problem [12]. The granularity issue does have a partial answer in terms of system purpose and the organisation framework. Our studies continue on extending the number of problem abstractions and examining the fit between applications and abstract problem classes. The theory as it stands describes the problem domain. It therefore models requirements specification and current system description. One future direction is to examine how solutions (designs) may be linked to specifications (problem description). Other workers have studied abstract classes of design problems [6, 18] and proposed classes of solutions. Extension of our current work in analogical matching envisages matching abstract domain models to software solutions belonging to the same problem class.

The domain theory represents an alternative view of the Subject, Usage and Development worlds espoused in [17]. The domain abstractions give a richer view of the subject world than captured in conceptual models of structured development methods (e.g. JSD, SSADM). Furthermore it adds the dimension of abstraction to models of problems which, with a machine assisted matching process [11] can support validation. The need for explanation in requirements engineering has been partially recognised by interest in argumentation models and design rationales. A further extension of our work is to look at modelling how decisions are made when requirements specifications are transformed into software designs. We believe explanation of requirements documents has an important function in decision making and validation.

In conclusion, the domain theory provides a systematic means for defining models of domain knowledge. It also gives a mechanism for empowering validation at several levels. We believe the future for validation in requirements analysis lies in creating more formal, knowledge intensive models which grow with corporate CASE repositories and our work in the NATURE project will progress in that direction.

ACKNOWLEDEGEMENTS

This work was partially funded by the European Commission's Esprit programme in Basic Research Project P6353 NATURE (Novel Approaches to Theory Underlying Requirements Engineering). It is a pleasure to acknowledge the stimulus and help of partners Matthias Jarke, Klaus Pohl, Stephan Jacobs, University of Aachen, Germany (project coordinator); Collete Rolland, Georges Grosz, Jean-Roch Schmidt, University of Paris 1, the Sorbonne; Yannis Vassiliou, Panos Constantopoulos, Giorgios Spanoudakis, FORTH research institute, Crete, Greece; Janis Bubenko, Benkt Wangler, Peter Holm, SISU, Sweden and David Till, Sara Jones, City University.

REFERENCES

1. Anderson J.R., (1985) Cognitive psychology and its implications. W.H. Freeman, New York.
2. Chandrasekaran B., Keuneke A. & Tanner M., (1992), Explanation in Knowledge Systems, The roles of the Task structures and Domain Functional Models. In Proceedings of Workshop on task based Explanation, Samos, Greece, Univ of the Aegean.
3. Dardanne A., Fickas S. & van Lamsweerde A., (1991) Goal directed acquisition in requirements elicitation. Proceeding 6th International Workshop on Software Specification and Design, Como Italy, IEEE Computer Society Press, 14.
4. Gentner D., (1983) Structure-Mapping: A Theoretical Framework for Analogy, Cognitive Science 7, 155.
5. Gick M.L. & Holyoak K.J., (1983), Schema Induction and Analogical Transfer, Cognitive Psychology 15, 1.
6. Harandi M.T. & Lee M.Y., (1991), Acquiring Software Design Schema: A machine learning perspective. Proceedings 6th Conference on Knowledge Based Software Engineering, Syracuse, NY, IEEE Computer Society Press, 188.
7. Jarke M., Rolland C., Sutcliffe A.G. & Vassiliou Y., (1993), Theories underlying requirements engineering: An overview of NATURE at genesis. Proceedings 1st International Symposium on Requirements Engineering, San Diego CA, IEEE Computer Society press, 19.
8. Jarke M., (1992), ConceptBase V3.1 User Manual, Aachener Informatik Berichte 92-17.
9. Johnson P., Johnson H., Waddington R. & Shouls A., (1988), Task Related Knowledge Structures: Analysis, Modelling and Application. In People and Computers IV (HCI-88); (eds) Jones D.M. and Winder R., Cambridge University Press, 35.
10. Loucopoulos P., Papamastatiou G., Pantazis D. & Diakonolaou G., (1991), Design and execution of event/action DB applications. Proceedings 2nd International Workshop on Deductive Approach to Information Systems and Databases., Aiguablava, Spain.
11. Maiden N.A.M & Sutcliffe A.G. (1992); Exploiting reusable specification through analogy. Communications of the ACM 35(4), 55.
12. Maiden N.A.M & Sutcliffe A.G., (1993), Requirements Engineering by Example: an Empirical Study. Proceedings of 1st International Symposium on Requirements Engineering, San Diego, CA, IEEE Computer Society Press, 104.
13. Meyer B., (1988), Object oriented software construction. Prentice Hall.
14. Mylopopoulos J., Borgida A., Jarke M. & Koubarakis M,, (1990), Telos: Representing Knowledge about Information Systems. ACM Transactions on Office Information Systems 8(4), 325.
15. Punchello P.P., Torrigiani P., Pietri F., Burion R., Cardile B., & Conti M., (1988) ASPIS, a knowledge based CASE environment. IEEE Software March 1988, 58.
16. Reubenstein H.B., 1990, 'Automated Acquisition of Evolving Informal Descriptions', Ph.D. Dissertation (A.I.T.R. No. 1205), Artificial Intelligence Laboratory, Massachusetts Institute of Technology.
17. Rosch, E., (1985), Prototype classification and logical classification: The two systems. In E.K. Scholnick (ed), New trends in conceptual representation: Challenges to Piaget's theory? New Jersey: Lawrence Erlbaum.

18. Shaw M., (1991), Heterogeneous design idioms for software architecture. Proceedings 6th International Workshop on Software Specification and Design, Como Italy, IEEE Computer Society Press, 158.
19. Sutcliffe A.G. & Maiden N.A.M., (1990), Software reusability: Delivering productivity gains or short cuts. Proceedings Interact'90, (eds) Diaper, D., Gilmore D., Cockton G. & Shackel B., North Holland, 895.
20. Sutcliffe A.G. & Maiden N.A.M., (1991), Analogical software reuse: Empirical investigations of analogy based reuse and software engineering practices. Acta Psychologica 78(1-3), 173.
21. Schank R.C. & Leake D.B., (1989), Creativity and learning in a case-based explainer, Artificial Intelligence 40, 353.
22. Maiden N.A.M & Sutcliffe A.G., (1991), Analogical Matching for Specification Retrieval, Proceedings 6th Knowledge-Based Software Engineering Conference, IEEE Computer Society Press, 108.
23. Sutcliffe A.G. & Maiden N.A.M., (1992), Analysing the novice analyst: cognitive models in software engineering, International Journal of Man-Machine Studies 36, 719.