In Proceedings 7th Knowledge-Based Software Engineering Conference, September 1992, IEEE Computer Society Press, 112-121.

Domain Abstractions in Requirements Engineering: an Exemplar Approach

N.A.M. Maiden & A.G. Sutcliffe

Department of Business Computing
School of Informatics
City University
London

Abstract

This paper reports an intelligent advisor which assists software engineers to reuse domain abstractions to improve the consistency, completeness and clarity of requirement specifications. Understanding unfamiliar domain abstractions can be difficult, so partial exposure and visualisation of concrete examples and metaphors are proposed to aid comprehension prior to reuse. These strategies are incorporated into an iterative fact acquisition and domain retrieval dialogue with important implications for fact capture and modelling during requirements engineering. The effectiveness of this paradigm is shown during user studies with a prototype of the intelligent advisor, during which software engineering retrieved and understood correct domain abstractions whilst analysing a new domain.

1: Introduction

Requirements engineering is complex, error-prone and in need of intelligent tool support (e.g. [1]) to assist the capture, modelling and validation of requirements [2]. One solution is to populate tools with the domain knowledge necessary to provide this support [3]. This paper reports an implemented and evaluated intelligent assistant for software engineers during requirements engineering. The intelligence of this assistant is derived from a set of domain abstractions which it uses to validate requirements specifications and guide requirements engineering.

Reuse of domain abstractions to assist the requirements engineering phase was used in the Requirements Apprentice project [2] to retrieve generic knowledge structures, known as cliches, for disambiguating and completing requirements specifications. Reuse of cliches

is indicative of the growing awareness of the need to model integrated knowledge structures, in contrast to structured analytic notations (e.g. [4]) which only capture knowledge subsets such as entity relationships or entity life histories. However, few clues were provided about the granularity or semantic content of these cliches, or about strategies for their retrieval and explanation. Indeed, examples from their paper [2] indicate a rather ad hoc approach to deriving these cliches. To overcome these limitations a theoretically-based model of domain abstraction is proposed. This model, in the form of a meta-schema for representing knowledge of software engineering domains, identifies key knowledge types for abstraction, their granularity for effective reuse and distinctions between different abstractions [5, 6]. This paper investigates how reuse of these theoretically-defined domain abstractions can guide and enhance the requirements engineering process. In particular, interactive dialogues for intelligent fact acquisition and explanation of retrieved domain abstractions are proposed and evaluated.

This paper has four sections. First, a user-centred approach to reusing domain abstractions is proposed, then a theoretically-based, logical model of domain abstraction is described. This is followed by specification of a tool for reusing domain abstractions during requirements engineering, and a detailed requirements engineering example of how the tool interacts with and assists a software engineer. Finally user studies with a prototype of this tool are reported to indicate the effectiveness of the proposed approach to reusing domain abstractions during requirements engineering. The paper is concluded with some future research directions.

2: How to reuse domain abstractions

The intelligent assistant described in this paper differs from other tools such as the Requirements Apprentice (Reubenstein & Waters 1991), ARIES [7], KATE [8] and

Leite's work [9] in the way that it uses domain knowledge and abstractions. For instance, the Requirements Apprentice implements a dependency-directed reasoning mechanism to retrieve and exploit cliches. As such, the cliches remain hidden from the software engineer. This paper proposes that domain abstractions can be exploited more effectively by software engineers than by an automated reasoning mechanism. People tend to have a greater capacity for understanding and reasoning than current artificial intelligence mechanisms although they often lack the key mental abstractions representative of software engineering expertise [10]. In this light, providing software engineers with domain abstractions is intuitively appealing. Domain abstractions must be well-understood before they can be reused effectively. Unfortunately, mental abstraction can be difficult, especially for inexperienced software engineers who need the most assistance during requirements engineering. One approach for encouraging software engineers to induce relevant abstractions is to provide them with concrete examples of these abstractions [11, 12], although the dilemma for intelligent tool support is how to retrieve domain abstractions to assist domain understanding without first understanding the domain.

To overcome this dilemma a paradigm for iterative retrieval has been adopted, during which the software engineer incrementally refines his or her definition of the domain by searching the space of all domain abstractions. Understanding domain abstractions is assisted by presenting concrete examples of these abstractions (e.g. [13]). Dialogue with the software engineer is directed by the underlying logical model of domain abstraction to focus the requirements engineering effort on critical domain facts. As such this paradigm blurs the distinction between gathering facts about a domain, retrieving domain abstractions and improving the domain description in light of these retrieved abstractions, see Figure 1. This iterative retrieval paradigm has been implemented in a prototype research tool known as AIR. AIR presents domain abstractions and concrete examples throughout a requirements session to assist software engineers to ensure the completeness and consistency of specifications and guide the processes of fact gathering and domain understanding during requirements engineering.

The benefits to software engineers from successfully learning relevant mental abstractions are obvious. Software engineers can effectively exploit domain abstractions to ensure completeness, consistency and accuracy of domain descriptions and requirements specifications. In the longer term they learn the key

mental abstractions upon which software engineering expertise is founded [10]. As such the use of examples during requirements engineering may be regarded as an effective paradigm for both problem solving and training. AIR's use of domain abstractions and concrete examples during requirements engineering is discussed in the remainder of this paper. First however, a theoretically-based, logical model of domain abstraction upon which this tool is built is described.

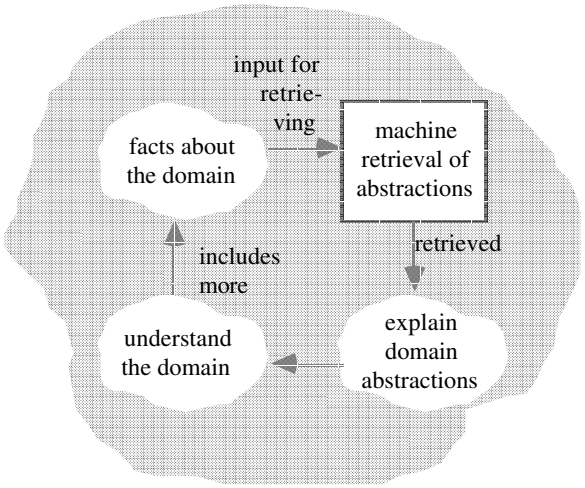


Figure 1 - overview of the iterative retrieval paradigm

3: A model of domain abstraction

A logical, theoretically-based model of domain abstraction is needed to determine the semantic content, boundaries and granularity of software engineering abstractions. It is founded on a logical model of abstraction in software engineering domains which proposes that many software engineering domains are instantiations of identifiable abstractions. Domain abstractions were defined by example-based analyses of software engineering analogies to determine the abstractions shared by many domains of the same class. These definitions were strengthened by imposition of constraints on these domain abstractions borrowed from existing computational models of analogical reasoning (e.g. [14]). This model is defined fully in [6]. Each abstraction can be differentiated from other abstractions by seven different types of fact or knowledge:

* actions leading to key state transitions with respect to an object structure;
* object structures describing domain states in the form of object-relations;
* object categories describing object roles in the context of state transitions;
* preconditions which trigger key state transitions;

* domain requirements describing domain states, functional needs or constraints to be supported by the information system;
* system functionality and domain events linked causally or otherwise to key state transitions;
* events which cause state transitions are either initiated by the information system or events external to it.

This model defines that domain abstractions are differentiated by key state transitions, hence a resource hiring abstraction, of which library loans is an example, can be distinguished from a resource containment abstraction such as stock control by the key transition of return. However, state transitions and object structures alone do not critically determine all software engineering abstractions, for instance the resource containment and object allocation abstractions can also be distinguished by several additional knowledge types. Stock replenishment in a stock control domain occurs when stock levels reach a minimum, while allocating theatre reservations to seats only occurs if the reservation and the seat have similar constraints such as seat price, so transitions between domain states can be triggered by different conditions. Furthermore, stock items and theatre seats both act as resources in their respective domains while each theatre reservation represents a need to be met by resources, so object categories can also be defined in the context of key state transitions. Finally system functionality related to key state transitions can also distinguish domains, for instance stock control systems involve restock, goods-in and stock-out functions while theatre reservation functions include allocate, assign and reserve. So far this model has been evaluated by the relatively-weak test of example. However, this attempt to formalise abstraction in software engineering domains represents an advance of Reubenstein & Waters' cliches [2]. An example of a theatre reservation domain and its underlying abstraction is shown textually in Figure 2, and graphically in Figure 3.

All domain abstractions are partitioned in an inheritance hierarchy which incrementally specialises each domain abstraction to identify larger groupings of domain classes. This hierarchical partitioning simplifies the retrieval of relevant abstractions, however it also draws on assumptions from cognitive models of memory, and in particular hierarchical models of natural categories [15] and memory schema which assert that human memory is organised in an informal hierarchy of classes. The current set of 35 identified domain abstractions is given in [6]. A subset of 10 such abstractions partitioned into four hierarchies implemented in a prototype version of AIR is shown in Figure 4. This abstraction hierarchy provides a

basis for procedural guidance during fact acquisition by combining incremental problem reformulation and hierarchical retrieval of abstractions. Retrieval of domain abstractions occurs iteratively, thus retrieval of more detailed abstractions provides feedback for incremental reformulation and improvement of the domain description, which in turn leads to improvement of the requirements specification and better retrieval of further domain abstractions. The semantic definition of these abstractions permits the validation of domain descriptions and their focus on omissions and inconsistencies. The mechanisms by which AIR exploits these domain abstractions is examined in the remainder of this paper.

---

transition (allocate, booking, reservation, seat, many)
precondition (allocate, sameproperties)
object structure (world, reservation, has-many)
object structure (world, theatre, has-one)
object structure (theatre, seat, has-many)
object structure (seat, booking, contains-one)
object structure (seat, booking, contains-no)
reqt (seat, booking, contains-one, sameproperties)
object category (booking, different-object-types)
object category (allocation, different-object-types)
triggering events of transitions (none)
general description (object-allocation)

transition (allocate, requirement, reqts, resource, many)
precondition (allocate, sameproperties)
object structure (space, requirement, has-many)
object structure (space, resources, has-one)
object structure (resources, resource, has-many)
object structure (resource, requirement, contains-one)
object structure (resource, requirement, contains-no)
reqt (resource, requirement, contains_one, sameproperties)
object category (requirement, different-object-types)
object category (resource, different-object-types)
triggering events of transitions (none)
general description (object_allocation)

Figure 2 - description of the theatre reservation domain and its abstraction

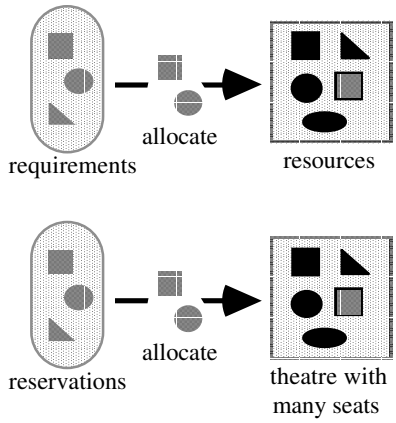


Figure 3 - spatial representation
of the theatre reservation domain
and its abstraction

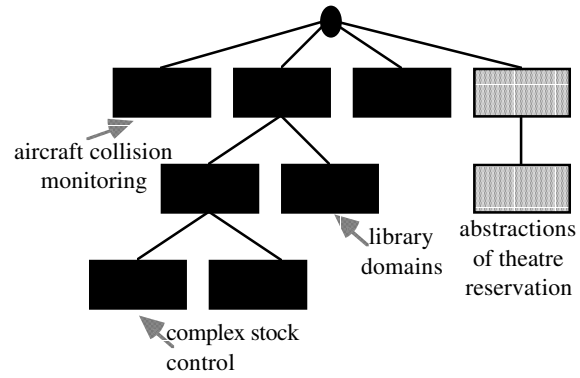


Figure 4 - sample portion of the model of
domain abstractions

4: A specification of AIR

AIR (Advisor for Intelligent Reuse) is a research tool developed originally for analogical specification reuse during requirements engineering. It is described more fully in [16]. One of AIR's major components is the problem identifier. This component uses the proposed model of domain abstraction to control dialogue with a single software engineer to acquire a complete, correct and unambiguous description of a domain's key facts. The resulting domain description is then used as input to the analogical matching process to retrieve reusable specifications described in [5]. However, AIR can also play an important role during requirements engineering, by assisting the software engineer to scope the problem domain, identify its boundaries, structure the problem domain and highlight its major features. This section

provides an overview specification of AIR for requirements engineering as well as some theoretical and empirical justification for this specification.

AIR plays two main roles during requirements engineering as suggested by the architecture shown in Figure 5. First it obtains facts about a new domain. Second it explains retrieved domain abstractions to the software engineer to aid the verification and validation of the specification for incompleteness, inconsistency and ambiguities. AIR adopts an iterative retrieval paradigm, obtaining facts about a domain then retrieving and explaining domain abstractions so that the requirements specification is refined incrementally. The aim of this process is to encourage the software engineer to induce relevant mental abstractions of key facts about the domain by presenting concrete examples of AIR's domain abstractions.

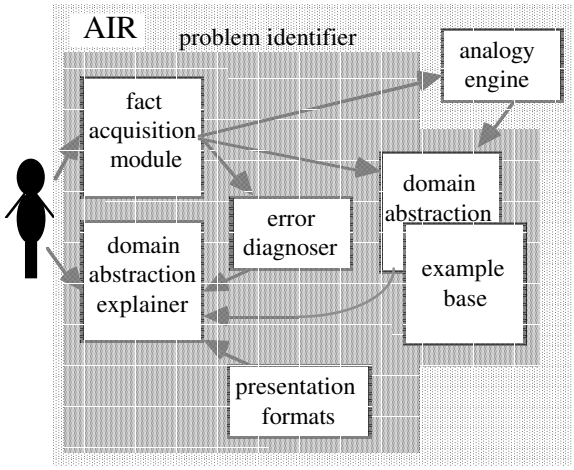


Figure 5 - overview architecture of AIR

Tool-based explanation of retrieved domain abstractions is necessary since psychological studies suggest that inducing or understanding abstractions is difficult without extra help. Empirical studies [17] revealed that inexperienced software engineers were unable to understand generic specification templates during reuse. Furthermore, studies of problem solving by example (e.g. [11], [18]) revealed that understanding abstract concepts is difficult. Subjects required considered help to understand key abstract concepts, so the need for explanation of even simple domain abstractions should not be underestimated. AIR's two main roles are examined in more detail.

4.1: Acquiring facts about a domain

Systematic fact capture of domain knowledge is likely to require considerable interaction with the software

engineer. Studies have suggested that it is difficult to get an agreed set of terms to describe computer artifacts which are simpler and better understood than information system requirements [19]. This paper hypothesises that examples can assist fact acquisition as well as explain domain abstractions by demonstrating semantic descriptions of analogical domains. Example-based categorisation has already been shown to be an effective retrieval mechanism [13] during studies of information access mechanisms for poorly understood concepts. These concepts were found to be continuously elaborated upon and evaluated until appropriate cues are constructed. People thought about categories of things in terms of prototypical examples as opposed to formal or abstract attributes, a result also identified by [15]. AIR encourages early understanding of domain abstractions using a battery of strategies for tool-based explanation including the use of concrete examples (e.g. [11]), which in turn necessitates the need for a retrieval mechanism based on the underlying model of domain abstraction. AIR also provides a closed set of domain semantics for describing concrete domains and their abstractions, thus the software engineer selects key domains semantics best describing the new domain throughout fact acquisition.

4.2: Explanation of domain abstractions

Strategies for explaining domain abstractions are drawn from current theoretical work on explanation and empirical investigation of induction and abstraction during problem solving by example. Empirical studies of example-based problem solving reported in [11, 18] revealed that induction of relevant mental abstractions only occurred when subjects were presented with the abstraction, at least two concrete examples of that abstraction and an overt prompt to similarities between the examples. Furthermore the representation of the abstraction and its examples affected induction, and spatial diagrams representing the critical abstraction proved more effective than textual descriptions. These findings provide an empirical basis for how AIR explains retrieved domain abstractions.

Theoretical research of explanation (e.g. [20, 21]) has suggested taxonomies of explanation borrowed from didactic strategies in intelligent tutoring systems which could be incorporated into AIR. These include concrete examples, analogy, abstraction and justification from first principles, indicating that abstraction and example are well-founded explanation strategies in their own right, although additional strategies will be needed to support explanation of complex abstractions by example.

4.3: AIR's fact acquisition strategies

Example-based retrieval and explanation of domain abstractions is central to AIR's design, however two additional strategies support the paradigm, namely visualisation and partial exposure to concrete examples. All three strategies are examined in more detail.

Retrieval and explanation by example: AIR presents concrete examples to encourage induction of mental domain abstractions and to assist prior retrieval of these abstractions. AIR's retrieval mechanism selects correct examples from simple domain descriptions by matching on key domain events and system functionality as defined in the meta-schema in Figure 2. This occurs for both empirical and theoretical reasons. Empirical studies of requirements engineering with inexperienced software engineers [22] revealed that system functions were identified more easily than other key facts about a new domain. System functions are linked to key state transitions are central to the proposed model of domain abstraction, as well as being the simplest form of fact to be defined using the knowledge meta-schema.

Visualisation of concrete examples: diagrammatic visualisation of domain abstractions and their examples can assist software engineers. Diagrams emphasise key state transitions and object structures which can be instantiated as familiar metaphors easily understood by software engineers. For instance, state transitions in respect to object structures may be instantiated as simple blockworlds [23] which involve the movement of objects or blocks from one state to another, see Figure 3 for an example. Visualising the target domain can also be promoted by permitting the software engineer to input diagrammatic representations of the domain.

Partial exposure of examples: mental laziness manifest as copying of analogical examples was revealed as a major problem in specification reuse [17]. Example-based explanation may also lead to undesirable copying, so AIR must discourage laziness while still encouraging understanding. Partial exposure and controlled access to domain abstractions and examples was proposed to inhibit such copying.

Additional strategies: explanation by visualisation and partial exposure to examples are insufficient to explain detailed constraints on the use of domain semantics, so description and justification [21] of semantics is required.

These explanations are also needed to understand detailed features of retrieved domain abstractions and examples. AIR also presents summaries of domain facts to encourage evaluation of domain descriptions.

5: An example session with AIR

This example session demonstrates AIR's dialogue for the acquisition of key domain facts prior to the retrieval of candidate domain abstractions. The semi-automated nature of this dialogue has important implications for fact acquisition strategies during requirements engineering. The dialogue consists of the six phases shown in Figure 6, although real world sessions may be more complex and involve iterative retrieval of abstractions. It demonstrates the overall fact acquisition process rather than the final, implemented version of AIR. Each of the dialogue's six phases is examined more closely during specification of the theatre reservation domain defined in Figures 2 and 3.

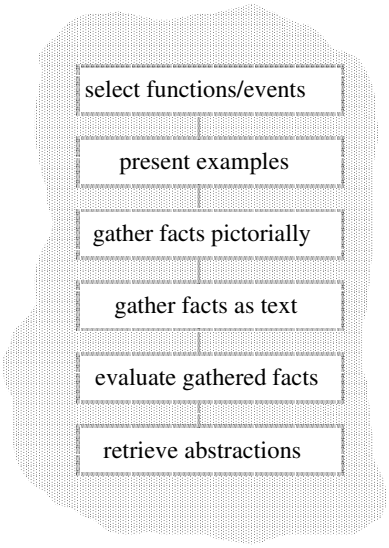


Figure 6 - major phases
of the example session with AIR

Select system functions or domain events: AIR asks the software engineer to choose key system functions from a selection provided, see Figure 7. Selection is supported by more detailed definitions and examples of each function/event which can be browsed by the software engineer. For the theatre reservation example the software engineer might be expected to select the allocate function. These functions/events are matched using AIR's simple retrieval mechanism to retrieve candidate abstractions.

Present concrete examples: the software engineer is presented with simple concrete examples of the same abstraction as the current domain. These examples are

presented in pictorial form to emphasise key state transitions and object structures, supported by explanation of important similarities with the current domain. Examples some in two forms. Well-understood analogical software engineering domains demonstrate the domain scope, its required level of abstraction, and the semantics needed to describe the remainder of the domain. In addition, familiar metaphors can be presented to emphasise key abstract concepts of certain classes of domain. These metaphors include water flow in and out of a beaker (stock control), and fitting different-shaped pegs to holes with the same shape (theatre reservation), as shown in Figure 8.

Gathering facts pictorially: AIR provides diagramming facilities to allow the software engineer to sketch key state transitions and object structures. Retrieved domain examples can provide a skeletal model for developing this sketch, supported by textual summaries of the semantic structure of the domain.

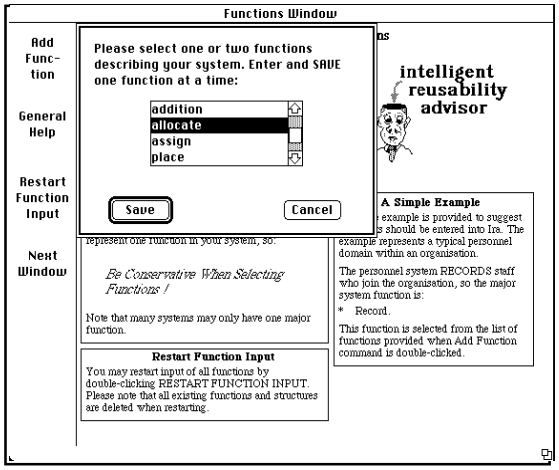
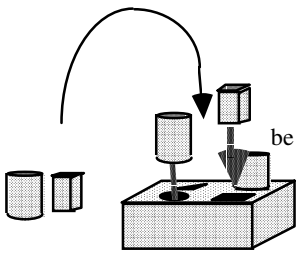
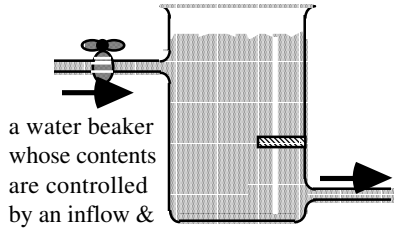


Figure 7 - function/event selection during description of the theatre reservation domain


A children's game with different-shaped pegs to be fitted to blocks, representing constraint matching in the theatre reservation domain



a water beaker whose contents are controlled by an inflow & outflow, similar to inventory control domains

Figure 8 - familiar metaphors representing key facts of several known domain abstractions

Gathering facts textually: several fact types defined in the meta-schema are best defined textually, including object categories, conditions on state transitions and triggering events on system functions. AIR provides domain semantics to be chosen, as shown in Figure 9. Descriptions, justifications and examples of each selection can be browsed by the software engineer as required.

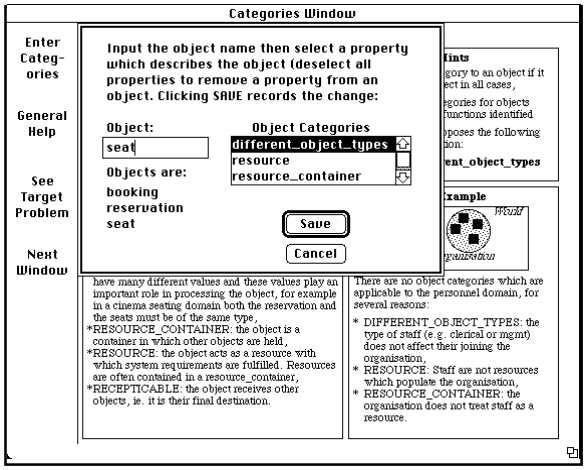


Figure 9 defining object types for the domain object 'theatre seat'

Evaluating gathered facts: the resulting description of the domain can be evaluated by both AIR and the software engineer. AIR matches the domain description against descriptions of tentatively-matched abstractions to identify possible omissions and inconsistencies, which are presented as a report [2] to the software engineer for improvement. Furthermore, the software engineer is encouraged to evaluate their own domain description to identify possible improvements. Once agreed, the domain description is passed to AIR's analogy engine to be matched to all known domain abstractions to confirm or reject the match.

Retrieval of matched domain abstractions: the acquired domain description is matched to retrieve complete domain abstractions using AIR's analogy engine [5]. An example of a complete domain abstraction for the theatre reservation domain is shown in Figure 10. Retrieved domain abstractions are explained using concrete examples, visualisation and partial exposure, although explanations of well-matched abstractions are often more complete and complex than explanations than candidate examples matched earlier in the dialogue.

To conclude, this example session demonstrates the major phases of an interactive dialogue with AIR, although more realistic scenarios may involve more iterative retrieval and agreement of equivalent examples. The dialogue represents semi-automated fact acquisition which differs from existing approaches to requirements capture and modelling. The remainder of this paper examines the implementation and evaluation of a prototype AIR founded on the design outlined in this paper.

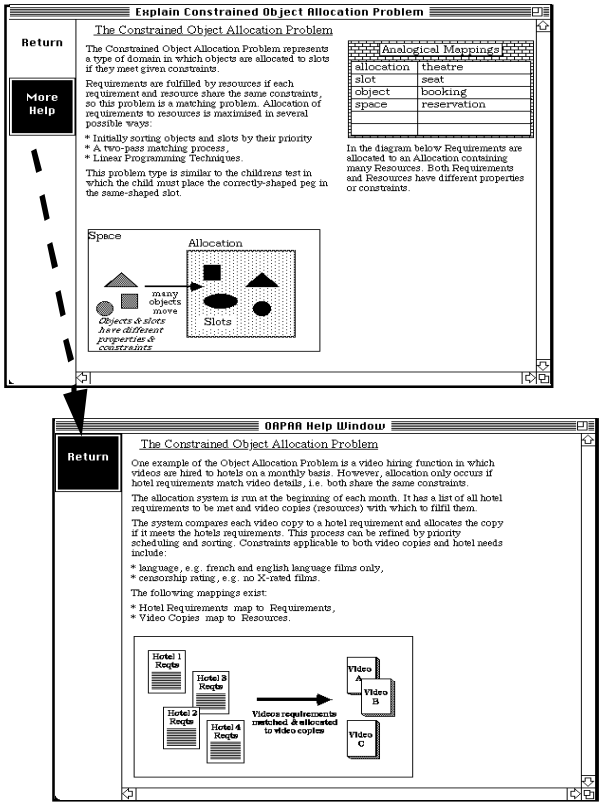


Figure 10 - retrieved domain abstractions for the theatre reservation domain

6: AIR's prototype

A prototype of the specified AIR was developed on an Apple Macintosh IIcx using LPA MacProlog 2.5 to exploit its extensive graphics facilities during

development of AIR's explanation dialogues. This prototype is described more fully in [6]. It differs from the description given in this paper in several ways. The most important omission is the lack of diagramming facilities for sketching key state transitions and object structures. Instead, software engineers were encouraged to sketch key transitions on paper prior to inputting their description in textual form. The other main difference from the specified AIR was the tool's limited ability to provide immediate feedback on the domain description. Otherwise the prototype implemented the explanation strategies and procedures described in this paper.

7: Evaluation of the prototype

Sofware engineering researchers often fail to evaluate the effectiveness of tools which they develop. The strategies incorporated into AIR's design were evaluated in a study which examined its effectiveness for acquiring and matching key facts about a domain during a requirements engineering task. The study permitted the evaluation of semi-automated fact gathering and domain abstraction reuse strategies during requirements engineering, and in particular example-based explanation and visualisation. AIR's effectiveness was measured by the prototype's ability to retrieve the correct domain abstractions for the target domain based on the key facts acquired by AIR. Evaluation was achieved through observation of inexperienced software engineers using the prototype. Inexperienced software engineers represented a worse case scenario for the tool since they were unlikely to have any previous exposure to domains and abstractions presented by AIR during the study. To add further realism to this evaluation, the software engineers were given no prior training or exposure to the prototype.

7.1: Method

Four software engineers with moderate or little systems analytic experience used the prototype to describe a production planning domain analogous to the theatre reservation domain shown in Figure 2. Each subject was given a textual description of the domain and asked to identify key facts about it, enter these facts into AIR in a sequence imposed by the dialogue, then understand and select between retrieved domain abstractions. Subjects were doctoral students and junior lecturers in the Department of Business Computing at City University with experience of structured analytic techniques including SSA [4] and JSD [24]. Subjects had 50 minutes to analyse and describe the production planning domain

followed by five minutes to understand and select the correct retrieved domain abstractions. Upon completion of this task a verbal and written questionnaire elicited further details of their behaviour. The effectiveness of the prototype was measured by the "goodness of fit" of the domain abstractions retrieved for the production planning domain. The prototype possessed two domain abstractions of the production planning domain, see Figure 4.

7.2: Results

All four subjects (S1-S4) input facts about the domain, although only three of these domain descriptions matched the correct abstraction while S3's description could not be matched to any known abstraction. The three successful subjects were also able to understand and agree that the retrieved domain abstractions could be instantiated to the production planning domain, representing a noticeable improvement over the results from earlier fact acquisition strategies reported in Maiden (1992). This limited evaluation of AIR's prototype indicates the effectiveness of an incremental, retrieve-then-refine paradigm to requirements engineering. Reasons for subjects' success and failure in the task were examined more closely.

Subjects who used diagrams to represent and understand their domains also developed more accurate and complete descriptions of that domain, although several discrepancies between subjects' diagrams and domain definitions did occur. In addition, S1 retrospectively admitted that promotion of diagramming techniques led to considerable improvements in AIR's usability while S4 claimed to have sketched the production planning domain directly from an example retrieved by AIR. The need for further diagramming facilities was also underlined by S4, who made extensive notes and sketches of the domain prior to fact input into AIR, to ensure that she understood the target domain effectively. Indeed, S4 reused her existing sketch of a retrieved example to model the target domain, suggesting the importance of visualisation through diagrams.

AIR promotes mental abstraction by partial exposure to pictorially-represented concrete examples of domain abstractions, so the effectiveness of these key strategies was examined more closely. Successful subjects claimed retrospectively to understand the relevance of retrieved examples to production planning while the unsuccessful subject failed to understand the example or map it to the production planning domain. Indeed, S1 claimed that the example was well-explained. However, a note of caution should be sounded, since S4 retrospectively claimed to rely too much on the example and extended similarities

with the example beyond the validity of the 'analogical' match [25, 26]. Finally, successful domain description also depended upon effective use of visualisation and immediate feedback. Retrospective questioning revealed that these strategies resulted in more effective acquisition of key state transitions and object structures, thus greater domain visualisation should be encouraged.

Despite the qualified success of the study, all subjects encountered difficulties while using AIR. The failed subject (S3) succeeded initially in describing key system functions, state transitions and object structures, all of which could have been matched to the correct abstraction. However, he extended this description to include incorrect and unnecessary facts. Retrospective questioning revealed that the subject found the description too simple in comparison with structured analysis models such as data flow diagrams [4], so he overspecified his requirements and added further object structures to describe entity-relationships between domain objects and a second state transition and more objects added to the domain definition. The result of these modifications was to obscure key domain facts so that the analogy engine was unable to retrieve any abstraction. Interference from structured analytic techniques presented an unforeseen problem for AIR. Indeed, both S3 and S4 expressed doubts about the simplicity of the domain description in comparison with complex models developed using structured analytic techniques. Further explanatory dialogues will be necessary to justify the scale of domain descriptions captured by AIR. Another problem encountered in this study was mental laziness: S3 admitted to being lazy and copied the retrieved examples during the latter stages of domain description, despite the strategies implemented in the prototype. These findings may suggest that mental laziness manifest as example copying was difficult to discourage in AIR.

7.3: Summary of the evaluation

Evaluation of AIR's prototype proved successful in that most software engineers were able to understand and select the correct domain abstraction for a new domain. Presentation of concrete examples, visualisation of domains through sketching, guided fact acquisition and concretisation of domain definitions all proved effective for three of the four subjects. Although the scale of this study is small, results do indicate that retrieval of domain abstractions and concrete examples, supported by visualisation and partial exposure of these abstractions during requirements engineering was effective. In particular, these strategies assist software engineers to

model new domains correctly and accurately as a basis for more effective and productive requirements engineering.

8: Discussion

This paper presents a prototype intelligent assistant for fact capture and reuse of domain abstractions during requirements engineering. Tool support for fact acquisition and explanation of retrieved domain abstractions was investigated as a basis for improving completeness and accuracy of requirements specifications. An intelligent assistant known as AIR was developed and evaluated during studies with real software engineers. AIR helped software engineers by iteratively acquiring facts about a domain then retrieving and explaining matched domain abstractions to improve the requirements specification. Tool-initiated dialogue was informed by an underlying model of domain abstraction which identified key facts for acquisition then retrieved relevant domain abstractions. Strategies for explaining retrieved abstractions founded on visualisation and partial exposure of concrete examples proved effective during user studies with the prototype.

This paper has implications for intelligent tool support to assist both fact acquisition and reuse of domain abstractions. In particular three research directions emerge, namely the need for a validated and effective model of domain abstraction, the impact of this model of knowledge representations and the extension of the user-oriented paradigm for domain abstraction reuse. Effective fact acquisition must be based on a model of domain abstraction to identify key facts about domains and provide procedural guidance during the requirements engineering process. This effectiveness may be determined by the naturalness of the fact acquisition strategies and domain abstractions during further user studies with AIR's prototype, especially when analysing more complex domains which involve iterative retrieval of abstractions. The proposed model stresses the need to acquire integrated domain knowledge, otherwise key domain features cannot be identified and relevant domain abstractions cannot be retrieved. This contrasts with the isolated facts captured using existing structured analytic notations [4]. However, the meta-schema of knowledge types will need to be extended to include problem, social and organisational facts important during requirements engineering.

Previous research of knowledge representation languages during requirements and system specification [27] has failed to suggest an effective set of domain semantics. The model of domain abstraction defined in this paper

provides a theoretical basis for such semantics by representing key facts and differences between domains without the need for domain-specific thesauri. However, more research is needed to validate and extend the coverage of domain abstractions and the proposed set of domain semantics. This may be achieved by evaluating domain abstractions either against key mental abstractions possessed by expert software engineers or by contrasting them with concrete domain models derived through domain analyses.

Finally a user-centred approach to reusing domain abstractions proved effective in studies reported in this paper, although more theoretical and empirical work is needed. A stronger theoretical basis is needed to support effective visualisation [28] and example-based explanation of domain abstractions. A taxonomy of example types for explaining domain abstractions is needed, including well-known analogical software engineering domains and familiar metaphors. This taxonomy must be supported empirical evaluation of how software engineers understand and learn abstractions from different types of example. In particular, choosing the correct example to meet software engineers' needs is strategically important if more flexible and iterative fact acquisition dialogues are to be implemented. The effectiveness of examples on mental abstraction is also dependent on their representation [11], so experimental investigation of examples presented textually, diagrammatically and using animation is needed. The role of animation for example-based explanation of abstractions is particularly appealing since movement can emphasise the key state transitions with respect to object structures central to the proposed model of domain abstraction. Findings from example-based explanation in cognitive psychology and human-computer interaction can also inform improvements to AIR's design. Indeed, the direction of current research in these disciplines may indeed suggest that reusing domain abstractions is an exemplar approach to requirements engineering.

References

[1] Meyer B., 1985, 'On Formalism in Specifications', IEEE Software, January 1985, 6-26.

[2] Reubenstein H.B. & Waters R.C., 1991, 'The Requirements Apprentice: Automated Assistance for Requirements Acquisition', IEEE Transactions on Software Engineering 17(3), 226-240.

[3] Arango G. & Freeman P., 1985, 'Modelling Knowledge for Software Development', Proceedings of 3rd Intl Workshop on Software Specification and Design, London, IEEE

Computer Society Press, 63-66.
[4] De Marco T., 1978, 'Structured Systems Analysis and Specification', Prentice-Hall International.
[5] Maiden N.A.M & Sutcliffe A.G., 1991, 'Analogical Matching for Specification Retrieval', Proceedings of 6th Knowledge-Based Software Engineering Conference, IEEE Computer Society Press, 108-116.
[6] Maiden N.A.M., 1992, 'Analogical Specification Reuse During Requirements Analysis', PhD Thesis, Department of Business Computing, City University, London, UK.
[7] Johnson W.L., Feather M.S. & Harris D.R., 1991, 'The KBSA Requirements/ Specification Facet: ARIES', Proceedings of 6th Knowledge-Based Software Engineering Conference, Syracuse, IEEE Computer Society Press, 48-56.
[8] Fickas S. & Nagarajan P., 1988, 'Critiquing Software Specifications', IEEE Software, November 1988, 37-47.
[9] Leite J.C.S.P., 1989, 'Viewpoint Analysis: A Case Study', Proceedings of 5th International Workshop on Software Specification and Design, IEEE Computer Society Press, 111-119.
[10] Guindon R., 1990, 'Designing the Design Process: Exploiting Opportunistic Thoughts', Human-Computer Interaction 5, 305-344.
[11] Gick M.L. & Holyoak K.J., 1983, 'Schema Induction and Analogical Transfer', Cognitive Psychology 15, 1-38.
[12] Riesbeck C.K. & Schank R.C., 1989, 'Inside Case-based Reasoning', Lawrence Erlbaum Associates, Hillsdale NJ.
[13] Fischer G., Henninger S. & Redmiles D., 1991, 'Intertwining Query Construction and Relevance Evaluation, Proceedings of CHI'91, ed. S.P. Robertson, G.M. Olson & J.S. Olson, ACM Press, 55-62.
[14] Holyoak K.J. & Thagard P., 1989, 'Analogical Mapping by Constraint Satisfaction', Cognitive Science 13, 295-355.
[15] Rosch E., Mervis C.B., Grey W.D., Johnson D.M. & Boyes-Braem P., 1976, 'Basic Objects in Natural Categories', Academic Press.
[16] Maiden N.A.M. & Sutcliffe A.G., 1992, 'Exploiting Reusable Specifications through Analogy', Communications of the ACM 35(4), 55-64.
[17] Sutcliffe A.G. & Maiden N.A.M., 1990, 'How Specification Reuse can Support Requirements Analysis', Proceedings of Software Engineering'90, ed. P. Hall, Brighton UK, July 24-27 1990, Cambridge University Press, 489-509.
[18] Gick M.L., 1989, 'Two Functions of Diagrams in Problem Solving by Analogy', Knowledge Acquisition from Text and Pictures, ed. H Mandi & J.R. Levin, Elsevier Science Publishers B.V. (North-Holland) 1989, 215-231.
[19] Furnas G.W., Landauer T.K., Gomez L.M. & Dumais S.T., 1987, 'The Vocabulary Problem in Human-System

Communication', Communications of the ACM 30(11), 964-971.
[20] Wenger E., 1987, 'Artificial Intelligence and Tutoring Systems: Computational Approaches to the Communication of Knowledge', Morgan-Kauffman, Los Altos.
[21] Breuker J., 1988, 'Coaching in Help Systems', Artificial Intelligence and Human Learning (Intelligent Computer-Aided Instruction), ed. J.A. Self, Chapman & Hall Computing, 310-337.
[22] Sutcliffe A.G. & Maiden N.A.M., 1992, 'Analysing the Novice Analyst: Cognitive Models in Software Engineering', International Journal of Man-Machine Studies 36, 719-740.
[23] Gupta N. & Nau D.S., 1991, 'Complexity Results for Blocks-World Planning', Proceedings of AAAI'91/MIT Press, 629-633.
[24] Jackson M., 1983, 'Systems Development', Prentice-Hall International.
[25] Halasz F. & Moran T.P., 1982, 'Analogy Considered Harmful', Proceedings of Human Factors in Computing Systems Conference, 15-17 March 1982, Gaithersburg, Maryland, ACM Press.
[26] du Boulay B., 1989, 'Some Difficulties of Learning to Program', Studying the Novice Programmer, ed. Soloway E. & Spohrer J.C., LEA Hillsdale, N.J., 283-299.
[27] Dardenne A., Fickas S. & van Lamsweerde A., 1991, 'Goal-directed Concept Acquisition in Requirements Elicitation', Proceedings of 6th Intl Workshop on Software Specification and Design, Como It, 25-26th October, IEEE Computer Society Press, 14-21.
[28] Lewis M., 1992, 'Situating Visualisation: Building Interfaces from the Mind Up', Multimedia Review Winter 1991/Spring 1992, 23-39.