In Proceedings of First International Symposium on Requirements Engineering, January 1993, IEEE Computer Society Press, 104-112.

Requirements Engineering by Example: an Empirical Study

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

Department of Business Computing
City University, London, UK

Abstract
Domain abstraction reuse during requirements engineering was evaluated as a paradigm during use of a research tool. Iterative fact acquisition and abstraction retrieval, supported by example-based explanation of abstractions, was implemented in a prototype known as AIR. Observation of AIR's use by inexperienced software engineers revealed this paradigm's effectiveness despite several problems arising. These problems were examined to identify improvements to future versions of intelligent requirements engineering environments.

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. One solution is to populate tools with generic domain abstractions to assist requirements specification [2] [3] [4]. This paper proposes a user-centred paradigm for reusing such abstractions and a tool called AIR (Advisor for Intelligent Reuse) to implement the paradigm.

This paper proposes that domain abstractions can be exploited more effectively by software engineers than by automated reasoning mechanisms such as [2]. A software engineer who understands key abstractions can improve the completeness, consistency and accuracy of specifications as well as gain software engineering expertise [5]. Unfortunately, retrieving and understanding domain abstractions may be difficult, so iterative fact acquisition, retrieval and explanation of abstractions is proposed [6], see Figure 1. Throughout these iterations, explanation of domain abstractions draws on empirical findings [7] which revealed that people understood abstractions when presented with concrete examples. AIR explains abstractions using simple and familiar examples alongside descriptions of these abstractions.

Unfortunately software engineering researchers often fail to evaluate the effectiveness of such tools, so this paper

reports an empirical study of how software engineers reuse domain abstractions during requirements engineering. It reports user studies with AIR to determine the effectiveness of the paradigm. First, key facts to be acquired by AIR for abstraction retrieval are outlined in a theoretical model of domain abstraction. Second, a prototype version of AIR is described, then an empirical study of inexperienced software engineers using AIR is reported. Findings from this study are then used to inform and improve the design of AIR.

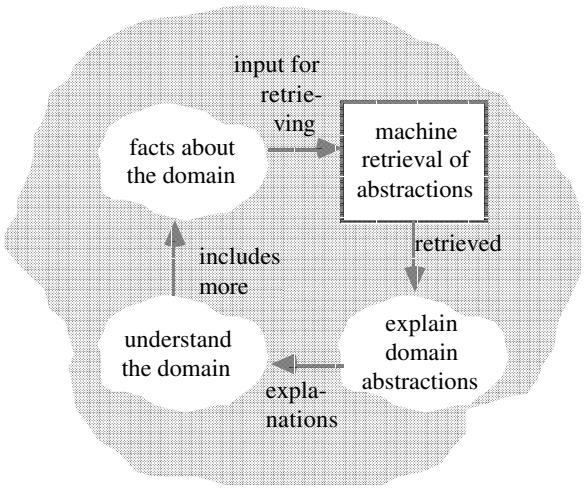

Figure 1 - overview of the proposed retrieval paradigm

2: A model of domain abstraction

A theoretical model of domain abstraction has been proposed to determine the semantic content, boundaries and granularity of software engineering abstractions [4]. This model claims that each abstraction is differentiated by seven fact types acquired by AIR. These fact types are:

* state transitions with respect to an object structure;
* object structures defined as relations between objects;
* domain requirements expressed as language statements which elaborate object structures;

* object categories describing their roles in the context of state transitions;
* conditions which must exist for state transitions to occur;
* key system functions and domain events associated with state transitions;
* the role of functions differentiating state transitions, similar to the distinction between internal and external actions in [8].

The model predicts that domain abstractions are defined first by key state transitions and object structures then by other fact types. These abstractions represent key facts about classes of domain, therefore they can act as kernel domain models for defining viewpoints and stakeholders in requirements engineering (e.g. [9] [10]). The model has been tested by example [4] and is being extended and evaluated in work reported by [11] to develop a comprehensive and valid model of domain abstraction in software engineering.

Retrieval is assisted by partitioning abstractions in an inheritance hierarchy which specialises their definition. The set of 35 identified domain abstractions is given in [4] and a subset of 10 abstractions for evaluating AIR is shown in Figure 2.

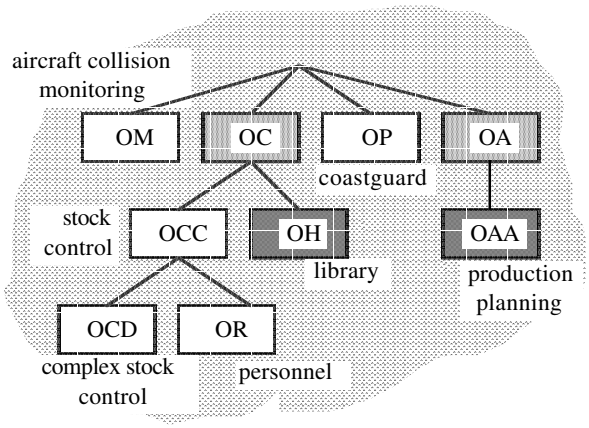

Figure 2 - AIR's problem space of 10 domain abstractions

The object hiring (OH) abstraction, of which university library loans is an example, can be distinguished from an object containment abstraction (OCC) such as stock control by the key transition of return. At lower levels of the hierarchy these abstractions are differentiated by other fact types such as object categories. The four domain class groupings and their candidate abstractions shown in Figure 2 are distinguished by the following key facts:

* the OM abstraction monitors the movement of objects between spaces to avoid object collisions;

* the OP abstraction examines object movement to ensure that they occupy the required positions;
* the OA abstraction assigns objects from requirements to resources assuming that they meet prespecified constraints. It can be specialised to represent a more complex object allocation abstraction such as production planning (OAA) by adding object structures;
* the OC abstraction describes the movement of objects out of a container. This abstraction can be specialised to object hiring (OH) or object containment (OCC), the latter of which can be further specialised by adding object structures so that it can be instantiated to either a complex stock control (OCD) or personnel (OR) domain, see Figure 2.

Ninety-three domain terms represent these abstractions. AIR encourages software engineers to describe new problems using these terms. An example of the university library description is shown textually in Figure 3 and graphically in Figure 4. It demonstrates facts which AIR must acquire and match to domain abstractions.

```
transition (lend, book, library, student, many)
transition (return, book, student, library, many)
condition (return, date-limit)
object structure (world, student, has-many)
object structure (world, library, has-one)
object structure (student, book, contains-many)
object structure (library, book, contains-many)
domain requirement (student, book, contains-many, date-limit)
object category (book, resource)
object category (library, resource-container)
```

Figure 3 - description of the university library domain

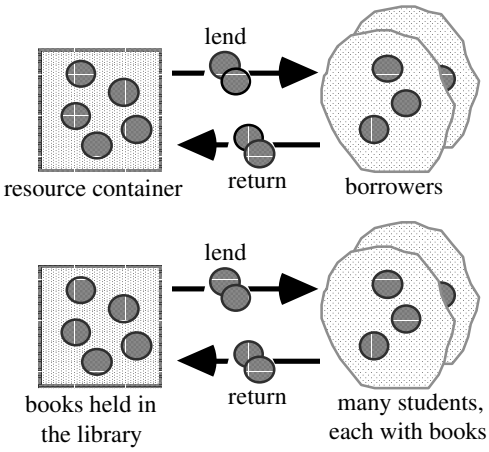

Figure 4 - spatial representation of the university library domain

and its abstraction

3: AIR's prototype

AIR is a research tool developed originally for analogical specification reuse during requirements engineering [12]. One of AIR's major components is the problem identifier. This component uses the model of domain abstraction to control dialogue with a single software engineer and acquire a domain's key facts. The resulting description is then matched to retrieve reusable specifications as described in [3]. However, AIR can also assist during requirements engineering by capturing key facts about a domain and explaining retrieved domain abstractions to improve a software engineer's domain understanding.

AIR's prototype was developed on an Apple Macintosh IIcx using LPA MacProlog 2.5 to exploit its extensive graphics facilities for explanation dialogues, see [4]. AIR uses example-based retrieval, explanation and visualisation of abstractions alongside description and justification of key fact types to assist fact acquisition. These strategies are implemented during a five-phase sequential dialogue with a software engineer which captures key state transitions and object structures before other fact types (see Figure 5). Each window elicits a different type of fact, although the software engineer can use pull-down menus to return to previously-explained facts. Most importantly, domain abstractions are retrieved early in this dialogue to explain and guide subsequent fact retrieval. This dialogue is demonstrated in the following sections during description of the university library domain shown in Figure 3.

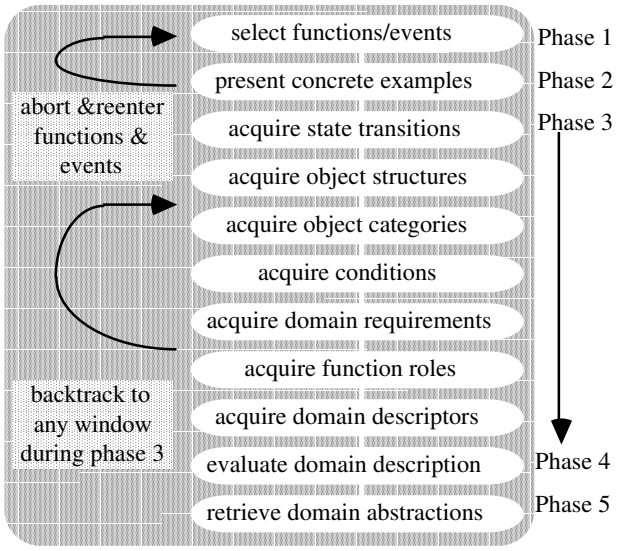

Figure 5 - Major phases and fact acquisition windows in AIR

3.1: Select system functions/domain events

AIR asks the software engineer to choose key system functions or domain events from a scroll menu to permit retrieval of domain abstractions at the first opportunity. Empirical studies [13] revealed that functions and events are identified more easily by software engineers than other domain facts. AIR then presents examples of two retrieved top-level abstractions matching the new domain. Entering the lend and return events for the library problem retrieves the top-level OC and OA abstractions (see Figure 2) and concrete examples for example-based explanation of the new domain.

3.2: Present concrete examples

The two retrieved abstractions are explained using concrete examples, see Figure 6. Gick & Holyoak [7] reported that presenting abstractions as spatial diagrams proved more effective than textual descriptions, so AIR presents pictorial representations which emphasise state transitions and object structures in the domain. Software engineers are also encouraged to sketch these examples to assist understanding. If the software engineer does not agree with the examples as a match then the software engineer can enter new functions and events.

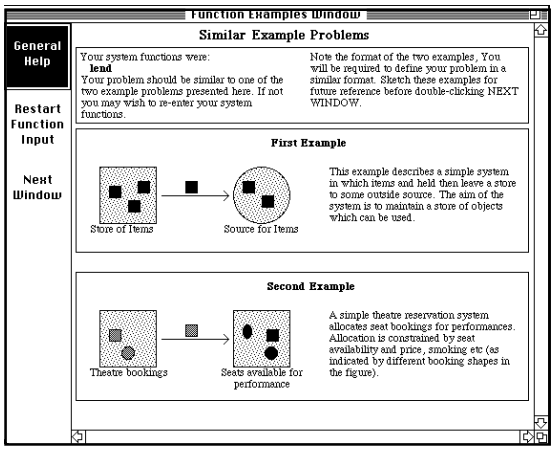

Figure 6 - explanation of domains abstractions by AIR

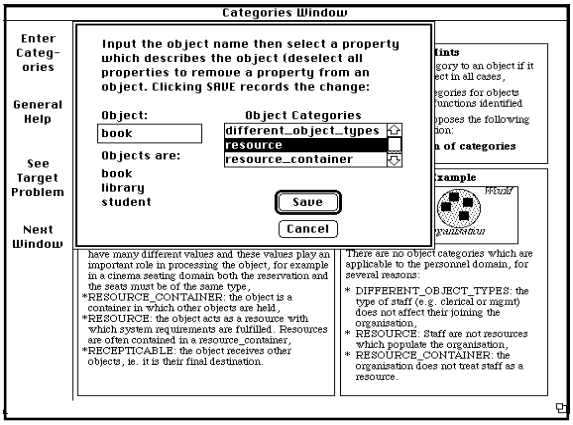

Figure 7 - typical fact acquisition window in AIR

3.3: Acquire facts

AIR acquires domain facts from the software engineer in text form. Candidate terms for each fact type are explained by description, justification and browsable examples [14]. Hints inferred by AIR from its original match are also presented. A single, non-analogical example illustrates each fact type. An example of a fact acquisition window is shown in Figure 7.

3.4: Evaluate acquired facts

Domain descriptions are evaluated by both AIR and the software engineer. AIR matches the domain description against the abstraction retrieved earlier to identify omissions and inconsistencies in state transitions and object structures. The software engineer is also prompted to evaluate the description and identify possible improvements.

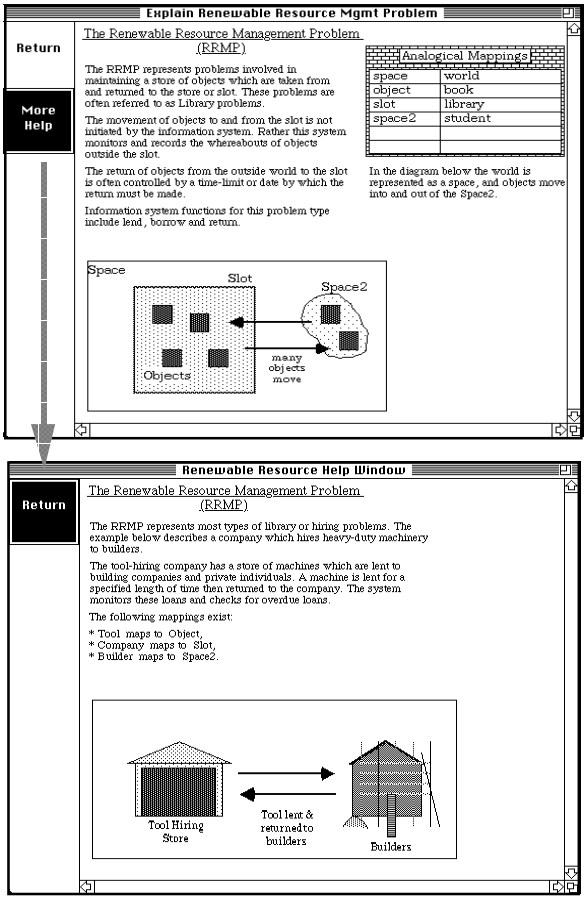

Figure 8 - domain abstraction and concrete example retrieved for the university library problem

3.5: Match domain abstractions

The evaluated domain facts are matched to one or many domain abstractions using AIR's retrieval mechanism [3]. Its effectiveness has already been shown by its ability to match incomplete domains to correct abstractions [4]. Explanations of the retrieved abstraction for the university library problem are shown in Figure 8. Once abstractions have been retrieved successfully AIR can provide more comprehensive support for requirements specification, including retrieval of specifications from repositories belonging to the same domain type [12]. However, this is beyond the scope of the paper and reported elsewhere.

4: Evaluation of AIR

AIR's fact acquisition strategies were evaluated in a study of their use by inexperienced software engineers. The principle aim was to evaluate the strategies' effectiveness in acquiring domain descriptions to retrieve abstractions. Hypotheses in this study were:

* fact acquisition strategies would capture descriptions complete and consistent enough for retrieving abstractions;
* concrete examples, spatial diagrams and description and explanation of detailed fact types would promote mental induction of these retrieved abstractions;
* structured fact acquisition is a natural form of dialogue for software engineers.

These hypotheses were examined by the extent of strategy use during the study and retrospective claims about their effectiveness, following Ericsson & Simon's practice [15]. The prototype had a standard Macintosh interface including windows, buttons and menus familiar to the software engineers, so little tool learning was necessary. Rather this study investigated the effectiveness of iterative retrieval and visual explanation of abstractions during use of the tool.

4.1: Method

Fourteen software engineers with moderate systems analytic experience used AIR while analysing a software engineering problem. Subjects were 10 second-year undergraduate business computing students, 3 post-graduate students and one lecturer in the Department of Business Computing at City University. All had some experience of structured analytic techniques such as SSA [16] but none had experience of object oriented techniques. Nine subjects (S6-S14) were requested to model the university library problem shown in Figures 3 and 4 while five subjects (S1-S5) analysed a production planning problem described in [17]. Each subject was given a textual description of the relevant problem then asked to:

* identify key facts about the domain and enter them into AIR in the sequence requested;
* understand one or many domain abstractions retrieved by AIR's matching mechanism;
* select or reject each retrieved abstraction as a match for their own problem.

Subjects had 1 hour to analyse and describe the problem and 5 minutes to understand and select machine-retrieved abstractions. A verbal and written questionnaire then elicited further details of their behaviour and understanding of the abstractions. The 'goodness' of subjects' domain descriptions was measured by the retrieved domain abstraction. Entered facts were also compared against the problem's correct abstraction to determine whether they correctly described that domain. It was hypothesised before the study that the university library (OH) abstraction was more difficult for AIR to

retrieve than the OAA abstraction due to its greater similarity to several object containment abstractions.

4.2: Results

Surprisingly AIR failed to retrieve the best-fitting domain abstraction for any subject, see Table 1. However, it did succeed in retrieving a correct, higher-level abstraction for 50% subjects, although it also retrieved the wrong abstraction for 2 subjects and no abstraction for 5 subjects. Failure appeared to be due to problems with the fact acquisition strategies rather than AIR's already-evaluated analogical retrieval mechanism [3]. Inter-problem differences also occurred. AIR retrieved a correct match for 4/5 subjects analysing the production planning problem but for only 3/9 library problem subjects. Both problems were equally familiar and complex, and retrospective rating revealed that AIR's interface easy to use (average 5.36 & 5.71 respectively on scale of 1-7), so results appeared to reflect greater difficulty in acquiring key facts to retrieve the OH abstraction.

Successful subjects						
S1	S2	S4	S5	S7	S9	S12
oa	oa	oa	oa	oc/ om	oc	oc
Unsuccessful subjects						
S3	S6	S8	S10	S11	S13	S14
		ocd/ or				oa

Table 1 - retrieved abstractions by subject

Subjects who were presented with the correct abstraction were categorised as successful while others were unsuccessful subjects. All 7 successful subjects understood and selected the correct retrieved abstraction. Retrospective questioning revealed that abstractions were easy to understand and map to the original problem, see Table 2. Of the others, S7 selected a correct abstraction from a choice of two retrieved abstractions, S14 correctly rejected the wrong OA abstraction but S8 incorrectly selected the OR abstraction. Overall, subjects understood and selected the machine-retrieved abstractions, so reuse of domain abstractions may be effective. However, acquiring key facts for retrieving these abstractions was problematic, so subjects' domain descriptions were examined for errors and omissions.

Domain descriptions: subjects' domain descriptions were examined for the number of correct and incorrect domain facts, see Table 2. Unsuccessful retrieval of abstractions appeared linked to:

* total numbers of incorrect facts entered (unpaired t-test: $t=3.365$, $p<0.006$);
* total numbers of incorrect object structures (unpaired t-test: $t=2.846$, $p<0.015$);
* total numbers of incorrect conditions on state transitions (unpaired t-test: $t=2.216$, $p<0.047$).

In addition, S14 received the wrong abstraction because she entered incorrect object categories defining books in the library. Indeed, omitting incorrect facts from the domain descriptions of five unsuccessful subjects would have improved abstraction retrieval.

The key state transition linked to returning books to the library distinguished the object hiring abstraction from other object containment abstractions, however only 3/9 subjects selected return as a key event and only S14 defined the return state transition correctly. Indeed, this lack of return combined with inclusion of some stock control concepts may explain why S8 was presented with the abstraction describing complex stock control. Therefore, omitted state transitions appeared to be another reason for failed abstraction retrieval. Unsuccessful subjects rated the entry of state transitions as significantly more difficult (unpaired t-test, $t=2.294$, $p<0.041$) than other fact types, so improved explanation of state transitions is needed.

The effect of AIR's key fact acquisition strategies on entry of correct domain facts and inclusion of key state transitions was examined more closely.

	Subjects	
	Succe-ssful	Unsucc-essful
Average retrospective scores for ease of understanding of:		
Domain abstraction: 1(poor)-7	5	0.57
Similarity mappings (1-7)	4.86	-
Average incorrect domain facts		
Total facts	3.29	6.14
Total of state transitions	0.43	0.85
Total of object structures	0.57	1.85
Object categories	0.43	0.71
Conditions on state transitions	0.29	1.14
Domain requirements	1	0.57
Roles of functions	0.14	0.14

Table 2 - subjects' identification of incorrect key domain facts

The role of examples in fact acquisition: AIR presented subjects with analogical and non-analogical examples during fact acquisition. The non-analogical personnel example explained terms permitted by AIR for describing domains. Two unsuccessful subjects claimed that the example was unhelpful. Subject S3 found it irrelevant and unhelpful while S10 believed it to be too basic and verbose.

	Subjects	
	Succe-ssful	Unsucc-essful
Averge retro understanding of personnel example (scale 1-7)	5.71	4.4
Total subjects claiming to under-stand analogical example	5	2
Total subjects who sketched the retrieved analogical example	6	3
Total subjects who sketched the problem domain	7	1

Table 3 - subjects' success in visualising the target domain and understanding AIR example (for retrospective understanding score: 1=poor, 7=good)

Subjects were also categorised by their retrospective understanding of analogical examples. Successful subjects

understood these examples (see Table 3). Unsuccessful subjects made several claims about problems with these examples which impeded fact acquisition:

* the examples were not related to libraries (2 subjects);
* graphic and text descriptions did not agree (3 subjects);
* the wrong analogical example was chosen (3 subjects);
* the examples did not help at all (1 subject).

Therefore, understanding analogical examples appeared to improve fact acquisition. Furthermore, two library problem subjects (S11, S14) criticised the retrieved OC example (see Figure 2) for omission of the return state transition. Therefore, incomplete analogical examples may have mislead subjects.

Spatial representation of the domain: subjects were encouraged to sketch their domain and 5/14 subjects developed full, representative diagrams. Four of these subjects were successful while 3 also sketched the correct analogical example (see Table 3). This indicated that development of spatial diagrams may have led to better domain understanding and entry of fewer incorrect domain facts.

AIR also encouraged subjects to sketch abstractions and their examples. Four out of the five subjects who did not sketch the examples were unsuccessful retrievers. Four of these five also failed to understand the examples, so sketching may be linked to better example and domain understanding. Reasons for not sketching the examples included a preference for mental rather than diagrammatic model formation (S11) and not finding the examples useful (S14).

Text-based description of fact types: subjects encountered difficulties when entering some facts in spite of descriptions, explanations and examples of terms for describing these facts:

* only 2/12 conditions on state transitions entered about the library domain were correct. Retrospective questioning revealed that fact types were easily understood and selected. Therefore, errors indicated subjects' inability to choose between date-limit and maximum/minimum-quantity conditions and suggested that they may have confused the library's lending and stock control tasks;
* only 5/14 domain requirements were modelled correctly. Subjects were unable to choose between date-limit and maximum-quantity as the main requirement, again suggesting confusion between lending and stock control. Indeed, 5/9 library problem subjects entered at least one fact about stock control;
* three subjects (S6, S10, S14) misused object structures.

Confusion with entity-relationship diagrams by S3 and S12 suggested an unexpected problem of interference from more familiar structured analytic notations.

5: Discussion

The reported study of domain abstraction reuse found that less experienced software engineers understood retrieved abstractions, however capturing key facts prior to retrieval was more difficult. Graphical representations of abstractions, concrete examples and sketching domains all appeared to improve the acquisition of key facts. However, problems arose when the scoping the domain and describing it using abstract terms. Furthermore, unsuccessful software engineers reused examples incorrectly. Inter-individual differences also occurred between domain descriptions. Close similarities between domain abstractions for the library problem reflected the need for a more accurate fact acquisition mechanism. These problems were examined more closely:

* poor domain scoping is a trait of inexperienced software engineers [17]. Therefore, software engineers may need to understand the whole problem before its parts;
* dependence on analogical examples (e.g. Sutcliffe & Maiden 1990) may have led to the omission of the return state transition from university library descriptions. Therefore, more accurate analogical examples may be needed, although a note of caution is necessary since software engineers may rely too much on the example and extend the match so that it becomes invalid (e.g. [18], [19]);
* mechanised entry of facts may be an unnatural way of thinking about problems. It contrasts with opportunistic reasoning exhibited by software developers elsewhere (e.g. [5], [13]). Software engineers in this study also ignored some prompts to enter data or sketch domains, hence allowances for some omissions and iterative domain description is necessary. Tolerance for information omission must be incorporated into fact acquisition strategies. An earlier study of unguided fact acquisition proved unsuccessful [4], so some form of mechanised fact capture is necessary;
* software engineers were sometimes unable to select the best fact to describe the domain. For instance, several subjects did not categorise books as library resources. Other definitions were also correct, however they were critical for retrieving the object hiring abstraction. Therefore, better explanation of fact types and key differences between candidate abstractions is needed.

Improvements to AIR's design which overcome these problems are six-fold:

* more iterative, example-based fact acquisition is needed to explain key differences and infer and acquire new domain facts as hints to the software engineer. More iteration may improve the naturalness of the fact acquisition dialogue by allowing software engineers to reason in terms of prototypical examples [20];
* graphic entry of object structures and state transitions is needed. One solution may be to allow software engineers to browse and select from combinations of transition and object structure templates. An alternative is to provide direct manipulation interfaces for constructing pictorial representations;
* multi-layered explanation of fact types is needed to overcome individual differences in software engineers' fact understanding. For instance, understanding can be gained from browsing more detailed explanations;
* fact acquisition strategies must tolerate some omissions during fact capture. Flexible strategies which allow for individual differences and errors are needed, although more empirical investigation is necessary to determine these differences. Dialogues which allow free fact entry but constrain the software engineer to entered key facts at predetermined markers in this sequence can still permit retrieval of domain abstractions;
* more domain exploration must be permitted prior to fact acquisition, to encourage effective domain scoping;
* clear separation between existing methods and other fact acquisition dialogues must be emphasised to avoid interference from structured notations such as entity-relationship diagrams.

The potential of a user-centred approach to domain abstraction reuse was shown by the empirical findings reported in this paper, however more development and evaluation is needed to refine the paradigm and scale it up to real world problems. Given these improvements this user-centred reuse of domain abstractions may represent an exemplar solution for supporting other requirements engineering activities.

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] Maiden N.A.M & Sutcliffe A.G., 1991, 'Analogical Matching for Specification Retrieval', Proceedings of 6th Knowledge-Based Software Engineering Conference, Syracuse NY, 22-25th September 1991, 101-112.

[4] Maiden N.A.M., 1992, 'Analogical Specification Reuse During Requirements Analysis', PhD Thesis, Department of Business Computing, City University, London, UK.
[5] Guindon R., 1990, 'Designing the Design Process: Exploiting Opportunistic Thoughts', Human-Computer Interaction 5, 305-344.
[6] 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.
[7] Gick M.L. & Holyoak K.J., 1983, 'Schema Induction and Analogical Transfer', Cognitive Psychology 15, 1-38.
[8] Dardenne A., Fickas S., van Lamsweerde A., 1991, 'Goal-directed Concept Acquisition in Requirements Elicitation', Proceedings of 6th Intl Workshop System Specification & Design, Como (It) 25-26 October 1991, IEEE Computer Society Press, 14-21.
[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, Pittsburgh, May 1989.
[10] Easterbrook S., 1991, 'Handling Conflict between Domain Descriptions with Computer-Supported Negotiation, Knowledge Acquisition 3, 255-289.
[11] Jarke M., Rolland C., Sutcliffe A.G. & Vassilou Y., 1993, 'Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis', Proceedings of 1st International Symposium on Requirements Engineering, IEEE Computer Society Press.
[12] Maiden N.A.M. & Sutcliffe A.G., 1992, 'Exploiting Reusable Specifications through Analogy', Communications of the ACM 35(4), 55-64.
[13] 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.

[14] 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.
[15] Ericsson K.A. & Simon H.A., 1984, 'Protocol Analysis', MIT Press.
[16] De Marco T., 1978, 'Structured Systems Analysis and Specification', Prentice-Hall International.
[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] 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.
[19] 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.
[20] Rosch E., 1991, 'Prototype Classification and Logical Classification: The Two Systems', in New Trends in Conceptual Classification: Challenges to Piaget's Theory ?, ed. E.K. Scholnick, Lawrence Erlbaum Associates.