# Requirements Critiquing Using Domain Abstractions

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

Centre for Human-Computer Interface Design
City University, London UK.

**Abstract**

*Reusing domain abstractions representing key domain features has been shown to aid requirement specification, however their role in requirements engineering has not been investigated thoroughly. This paper proposes domain abstractions to aid requirements critiquing as well as specification, thus maximising the payoff from retrieving domain abstractions. The requirements critic is part of a prototype intelligent requirements engineering toolkit being developed as part of the Nature project, ESPRIT basic research action 6353. The critic retrieves domain abstractions to validate requirement specifications for problems including incompleteness, inconsistencies and ambiguities. Intelligent, mixed initiative dialogue between the critic and requirements engineer permits requirements critiquing at the right time and level of abstraction.*

## 1: Introduction

Requirements engineering is complex, error prone [1] and in need of intelligent support for capturing complete, consistent requirement specifications [2] and guiding the requirements engineering process. One solution advocated by the ESPRIT Nature project [3] is to populate tools with domain abstractions to assist requirement specification [4,5]. Domain abstractions represent the fundamental behaviour, structure and functions of a domain class. This paper proposes that domain abstractions can aid critiquing of complex system requirements and describes an intelligent, cooperative critic for that purpose.

Reuse of domain abstractions during requirements engineering has been investigated by several authors [e.g. 4,5,6]. The Requirements Apprentice [4] retrieved clichés to validate the completeness and consistency of requirement specifications. The Advisor for Intelligent Reuse [7] also used domain abstractions, which are a form of complex clichés, as a starting point for specifying new requirements. However, retrieving and understanding domain abstractions are complex tasks [5,8] so the leverage from reusing domain abstractions must be maximised. This paper proposes that retrieving domain abstractions can assist requirements critiquing as well as guide fact capture and aid requirement specification. Indeed, a critic is essential for intelligent assistance during requirements engineering.

This paper also argues for a cooperative critic for requirements engineering, in contrast to existing, automated software design critics [9]. Cooperation between critic and requirements engineer is necessary because existence of complete and application-specific domain models cannot be assumed. Rather, validation is achieved by explaining domain abstractions so that the requirements engineer can detect problematic situations such as incompleteness, inconsistencies and ambiguities in requirement specifications [1,2]. Domain abstractions represent the fundamental behaviour, structure and functions which must be explained to encourage domain understanding prior to requirements critiquing, otherwise cooperative critiquing will fail. Subsequent explanation of more detailed domain abstractions then permits the requirements engineer to detect problem situations in the requirement specification. This paper describes such a cooperative critic to aid detection of problem situations to be resolved and requirements to be defined, elaborated or modified. It claims that this critic is the linchpin for intelligent dialogue between tool and requirements engineer.

The paper is divided into four parts. First, the role of generic domain knowledge for critiquing requirement specifications is examined. Requirement specifications define goal states to achieved, system functions to be implemented and the domain's behaviour and structure. Second, a theoretical model of software engineering domain abstractions is presented. The functionality of the requirements critic is then defined and demonstrated during a scenario. Finally implications and evaluation of the requirements critic are discussed.

## 2: Critiquing requirement specifications

The need for intelligent software engineering toolkits to critique requirement specifications and software designs [9] has been recognised, however few successful critics have been built. The challenge in building critiquing systems is not simply to provide requirements or design feedback. Rather the challenge is 'to say the "right" thing at the "right" time' [10, p157]. Fischer and his colleagues claim that the solution is to embed critics in domain-oriented design environments with domain knowledge for effective critiquing of designs. This paper argues that intelligent critics need domain models, however requirements engineering cannot assume existence of formal, application-specific models because these are expensive to acquire [41,42]. One solution which overcomes this knowledge acquisition bottleneck is to use generic domain knowledge. This solution distinguishes the requirements critic from existing design critics which need complete domain knowledge [10].

Cooperation between critic and requirements engineer is necessary because lack of complete domain models means that requirements cannot be critiqued automatically. However, domain abstractions are difficult to understand [8] and require explanation. This need for explanation is a shortcoming identified in existing critiquing systems [10]. Successful strategies for explaining domain abstractions have included spatial representations of abstractions, presentation of prototypical examples and guided exposure to aid mental model formation [7,8]. These strategies can assist requirements engineers to detect incompleteness, inconsistencies and ambiguities in requirement specifications. Explanation of domain abstractions can also aid detection of redundancy, overspecification and wishful thinking [2] by defining the structure, scope and level of abstraction of the requirement specification. Therefore, explanation of domain knowledge and validation of requirement specifications are facets of the same problem. Requirements are validated by explaining domain abstractions which permits detection of problem situations. This view differs from that of Fischer [11] who argues that explanation and critiquing are distinct activities.

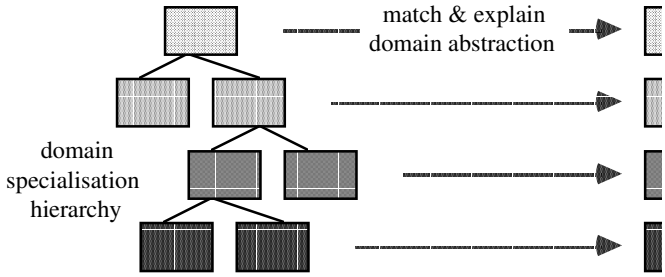


**Figure 1 - role of the domain specialisation hierarch specification**

Furthermore, this paper proposes an active critic which can intervene at the right time and level of abstraction. Fischer et al. [10] reported that existing design critics were not activated early enough in the design process while active critics can disrupt designers' concentration on the task and interfere with creative processes. One solution for effective intervention is frequent matching of requirement specifications to different levels of domain abstraction. Furthermore, retrieved abstractions set the context for the requirements engineering task, thus ensuring the timeliness of intervention and avoiding disruption of the process. Human cognition during software design [12] and requirements engineering [13] is situated [14], and requirements engineers reason in opportunistic fashion and respond to available information rather than a planned analysis worked out ahead of time. Therefore, frequent explanation of domain abstractions is unlikely to disrupt planned strategies for the task.

This paper views requirements engineering as an iterative process of: (i) fact and requirements capture; (ii) retrieval of domain abstractions; (iii) critiquing of requirement specifications. Most software engineering domains can be ascribed to one of a tractably small set of domain classes distinguished by a small number of key features [15]. The domain space is structured as an inheritance hierarchy which specialises high-level domain abstractions to more detailed domain classes. Domain features and functional requirements are matched to domain abstractions at different levels in the specialisation hierarchy, see Figure 1. Retrieval of high-level domain abstractions encourages domain understanding and draws attention to key domain features. This domain understanding is exploited later in the process as the specification is specialised and matched against more specialised domain abstractions. This frequent matching enables more effective critiquing at the right time and level of abstraction. Therefore, the

critic aids requirements validation, however it also guides requirements capture in light of retrieved domain abstractions and detected problems in the specification. Therefore, the requirements critic complements existing requirements engineering approaches which provide reasoning facilities such as propositional deduction and dependency maintenance [4] and expressive knowledge representation languages [16]. The advantages gained from the requirements critic are demonstrated later in this paper. First however, Nature's domain theory is outlined as a basis for describing the critic.

## 3: A theory of domain knowledge

A theoretical model of domain knowledge has been proposed to determine the semantic content, boundaries and granularity of software engineering abstractions [15]. Experienced software engineers' tend to recall and reuse mental knowledge structures or abstractions when specifying new requirements [12]. Therefore reuse of domain abstractions mimics expert software engineering behaviour [4]. The theory uses cognitive models of memory [17] and hypothesises that domain knowledge is organised in a class hierarchy and that classes share general features. Domain abstractions are composed of object and information system models. Object system models define the fundamental structure and behaviour which discriminate domains and underpin the requirement specification. Discrimination is by knowledge type at descending levels of the specialisation hierarchy:

- state transitions of objects with respect to object structures;
- positive or invariant goal states to be achieved or avoided;
- stative pre/postconditions on state transitions;
- events which trigger or stop state transitions.

Information systems define functions, report structures and constraints which are instantiated to functional and non-functional requirements [15]. Each information system accesses many object systems, therefore information systems represent important but non-discriminating requirements in the specification. The knowledge meta-schema and 5 domain abstractions have been implemented using the knowledge representation language Telos [18] and ConceptBase [19], an object-oriented deductive data base. This implementation builds on a first version of the theory which demonstrated analogical matching between many software engineering domains [20]. The object system for object hiring in Figure 2 shows state transitions with respect to object structures which are central to the theory.

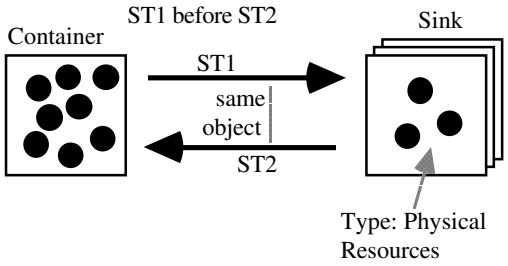


**Figure 2 - graphical illustration of the object hiring object system**

The domain theory is composed of over 120 object and 20 information system models representing key determinants of business and real-time domains. It is the domain specialisation hierarchy which permits active guidance and critiquing during requirements engineering. Furthermore, object system models can be aggregated to represent common patterns of application, thus enabling the use of domain abstractions for requirements critiquing during specification of real-world systems. The next section describes the requirements critic of AIR, a tool which exploits this theory of domain knowledge to aid the requirements engineer during requirements critiquing.

## 4: AIR: Advisor for Intelligent Reuse

The AIR toolkit (Advisor for Intelligent Reuse) has three major components which capture, match and critique requirements.

### 4.1: The requirements capturer

The capturer has two approaches for acquiring new facts and requirements from the requirements engineer. First, guided requirements capture requests fact types which are powerful discriminators between domain classes, thereby enabling matching and selection of domain abstractions. An intelligent dialogue which guided fact capture was implemented in AIR's first version, and evaluation of its use revealed moderate success of the strategy [7]. Second, unguided requirements capture is controlled by the requirements engineer. Requirements modelled using structured notations such as data flow diagrams [21] are translated to domain structure, functions and behaviour using a computational mechanism. In both approaches however, the capturer

supports the critic by enabling requirements validation and guiding the requirements engineer after critiquing.

### 4.2: The domain matcher

The requirements critic explains domain abstractions retrieved by a computational mechanism for analogical reasoning known as the domain matcher [22]. The matcher also infers mappings between abstractions and the new domain to detect problem situations. Retrieval is achieved by first matching domain features to a high-level object system then refining this match to specialisations of the object system until no further specialisation is possible. Matching also retrieves information systems linked to the retrieved object systems. Another tool, the problem classifier, exploits inferred object-pair and fact-pair mappings to aid detection of problem situations in the requirement specification [23]. These computational mechanisms have been implemented on a SparcStation IPX using BIMprolog as part of Nature's concept demonstrator [24].

## 5: The requirements critic

The requirements critic aids domain understanding and critiquing by explaining retrieved domain abstractions and detected problem situations to the requirements engineer. The critic has five major features: (i) different guidance strategies which enable a mixed initiative dialogue with the requirements engineer; (ii) critiquing strategies to enable intelligent validation of requirements; (iii) strategies for explaining domain abstractions; (iv) a controller for intelligent selection of strategies; (v) a problem notepad which reasons about mappings inferred from the domain matcher to detect and type problem situations (Meyer 1985) for critiquing.
Guiding and critiquing strategies enable the critic to alert requirements engineers to problematic situations, avoid unnecessary interruptions of the requirements engineering process and allow user control over the critic's intervention strategy. Guidance strategies encourage specification of complete requirements at the right level of abstraction to avoid overspecification. Critiquing strategies assist detection of inconsistencies, ambiguities and wishful thinking in the specification. The requirements critic combines these guidance and critiquing strategies in a mixed initiative dialogue with the requirements engineer. This dialogue distinguishes the critic from other requirements engineering tools [16,25].
The critic is being implemented in two phases. An existing implementation emphasises the advantages of the domain theory for requirements critiquing [24]. This demonstrator will then be extended towards the full requirements critic as part of the Nature project.

### 5.1: Overview of the critiquing process

Requirements critiquing occurs in two parts. First, the critic retrieves domain abstractions to encourage domain understanding and guide fact capture by drawing attention to key domain features, although some critiquing can also occur at this stage. Domain abstractions must be explained because key features must be understood before more detailed domain comprehension and hence critiquing is possible. Second, explanation of lower-level domain abstractions permits improvement of the requirements model and detection of problem situations in the requirement specification. Problem resolution is aided by explaining problem situations detected by the problem notepad. During both phases, iterative retrieval and explanation of domain abstractions promotes domain understanding and requirements critiquing. Furthermore, the critic provides 'rollback' buttons which enable the requirements engineer to undo matching if mistakes are perceived. Therefore, it is responsive to users' needs and can support situated reasoning during requirements engineering.
The requirements critic controls interaction with the requirements engineer after iterative retrieval of domain abstractions at each level in the specialisation hierarchy. Intervention is controlled by guidance and critiquing strategies:

1: the critic first guides requirement capture and specification in conjunction with the requirements capturer by explaining domain abstractions which are retrieved or which specialise the matched abstraction;
2: the critic then enables requirements critiquing by explaining domain abstractions. Critiquing is assisted by fact-pair and object-pair mappings inferred by the domain matcher;
3: further critiquing is encouraged by explaining problems detected by the domain matcher and problem notepad, akin the critiquing proposed in [10].

The remainder of this section describes how guiding, critiquing and explanation strategies are exploited by the requirements critic.

### 5.2: Active versus passive guidance

Guidance strategies reflect the degree of initiative taken during requirements capture before and after

requirements critiquing. Requirements engineering is composed of many complex tasks, some of which benefit from active guidance while others are supported more effectively by a hands-off approach:

- active guidance during requirement specification is achieved by explaining domain abstractions at the correct level of abstraction. This guidance informs fact capture using the theoretical task model [7];
- active browsing of the domain specialisation hierarchy using graphic illustrations and detailed explanations of domain abstractions aids selection of the best fit;
- unguided requirements specification when the requirements engineer has initiative to enter new requirements and domain features. Unguided requirement specification is needed for tasks such as domain exploration and idea brainstorming where tool intervention is likely to disrupt reasoning processes.

### 5.3: Active versus passive critiquing

Critiquing can be passive or active according to definitions in [10]:

- active critiquing. The requirements critic intervenes at the right time to explain retrieved domain abstractions and detected problems at the right level of abstraction;
- passive critiquing. The problem notepad detects problem situations but the critic does not intervene. Rather, detected problem situations are recorded on a problem notepad containing issues to be resolved. Problems can be browsed and acted upon by the requirements engineer when appropriate.

Therefore guiding strategies aid fact and requirement capture described in [7], while critiquing aids validation by explaining domain abstractions and detected problems.

### 5.4: Strategies for explaining domain abstractions

Domain understanding for effective requirements critiquing is aided by explanation using four explanation strategies.

**Visualisation of domain abstractions:** visualisation [26] of domain abstractions can aid understanding and hence critiquing. Gick & Holyoak [27] reported better learning of abstractions from spatial diagrams representing critical determinants during analogical problem solving. Diagrams are annotated and supported by text descriptions to aid recognition and understanding of domain abstractions [8].

**Presentation of prototypical examples:** prototypical examples can also aid understanding of domain abstractions. Empirical studies revealed that abstraction induction required presentation of two or more analogical instances [27]. Indeed, people often understand new concepts using prototypical examples [17,28].

**Animation of domain behaviour:** domain abstractions can be animated using dynamic illustrations of domain behaviour, similar to animations of computer algorithms proposed in [29]. Text and diagrammatic explanations are also needed to consolidate learning. Domain animations are interactive and permit playback and pause facilities to aid exploration.

**Guidance for formation of mappings:** tool-based guidance aids formation of mappings between domain abstractions, prototypical examples and the requirement specification. Notations which guide analogical mappings include tabular definition of object mappings.

These explanation strategies aid understanding of formal definitions of domain abstractions and operationalise strategies for guidance and critiquing.

### 5.5: Selection of guidance, critiquing and explanation strategies

Intelligent, mixed initiative dialogue needs a controller to select strategies, otherwise active critiquing will fail. User modelling and dialogue planning to support tasks has proved difficult [30], so the controller selects strategies based on the state of the requirement specification and simple states of user-system dialogue. Heuristics for strategy selection are derived from the theoretical model of domain abstractions, empirical studies of their reuse and evidence from cognitive psychology. Guidance is controlled by the following heuristics derived from theoretical models of domain abstractions and the task model for their reuse:

AG1: IF a domain abstraction is retrieved THEN it must be explained to encourage domain understanding;

AG2: IF the requirement specification has no facts remaining to be mapped THEN active guidance occurs. This indicates that the requirement specification cannot be matched to lower-level domain abstractions;

AG3: IF the domain matcher cannot specialise the current domain match THEN active guidance occurs. This indicates that the specification has been extended in the wrong direction;

AG4: IF a domain abstraction is selected THEN the requirements engineer is requested to enter key facts or requirements which discriminate the selected domain from other domains;

AG5: IF the requirements engineer requests active guidance THEN active guidance occurs;

AG6: IF the requirements engineer requests active browsing THEN browsing of the domain specialisation hierarchy is permitted.

Critiquing is also controlled by heuristics in the dialogue controller. Intervention during critiquing is driven by problem types [2] detected by the problem notepad and the number and duration of these problems [23]. For example, incompleteness is detected from a fact describing the domain abstraction which is not instantiated in the requirement specification. The dialogue controller uses the following heuristics:

AC1: IF the requirement specification has omissions which inhibit its specialisation using the domain specialisation hierarchy THEN active critiquing occurs;

AC2: IF the requirement specification has inconsistencies or ambiguities which inhibit its specialisation THEN active critiquing occurs;

AC3: IF the requirement specification is matched to more than one object system model THEN guide the requirements engineer to decompose the requirement specification. This indicates complex requirements which are matched to several object system models. Decomposition of complex requirements is guided by the requirements refiner [31];

AC4: IF the requirement specification includes a network of additional, unmapped state transitions and object structures THEN active critiquing occurs. This indicates overspecification of requirements;

AC5: IF problem situations have not been rectified or acted upon over time THEN active critiquing occurs. This heuristic increases the importance of a problem over steps during a session. A problem is explained if it attains an importance threshold. Importance is a factor of the quantitative degree of mismatch with the domain abstraction, the fact type causing the problem and the number of session steps that the problem has existed;

AC6: IF the explained domain abstraction is a high-level abstraction THEN only passive critiquing occurs to promote domain understanding rather than requirements validation;

AC7: IF the total of detected problems attains a threshold value THEN active explanation of the entire domain abstraction occurs. This indicates possible poor domain understanding to be resolved by further explanation;

AC8: IF the requirements engineer requests active critiquing THEN active critiquing occurs.

Explanation of formal domain abstractions is also controlled by heuristics derived from empirical studies of explanation during analogical problem solving and animation of algorithms:

ES1: IF problem situations are detected THEN explain domain abstractions before explaining detected problems;

ES2: IF explanation using domain visualisation is selected THEN supplement visualisation with text-based descriptions of domain structure, behaviour and functions [8];

ES3: IF explanation using domain animation is selected THEN explain domain abstractions using visualisation before animation [29];

ES4: IF explanation using prototypical examples is selected THEN explain domain abstractions using two prototypical examples. This encourages induction of key abstractions [27]. High-level domain abstractions are not explained using prototypical examples because they cannot be instantiated to real-world domains. Rather, they exist to structure the domain space and guide matching;

ES5: IF domain abstractions or prototypical examples are explained THEN provide tool-based guidance for formation of mappings with these abstractions and examples [8];

ES6: IF more than one domain abstraction must be explained THEN also explain key differences between candidate domain abstractions to aid their selection;

ES7: IF more than one object or information system model must be explained THEN gradual exposure and explanation is needed to aid mental model formation [8];

ES8: IF prototypical examples are explained THEN present simple software engineering examples before more complex software engineering examples [32];

ES9: IF domain abstractions are explained THEN present formal definitions of Telos objects and more formal domain semantics to aid precise understanding.

Therefore, requirements critiquing at the right time and level of understanding is achieved by explaining domain abstractions during a mixed initiative dialogue. This dialogue is controlled by guidance, critiquing and explanation heuristics after iterative retrieval of each abstraction. Heuristics for selecting strategies for guiding

and requirement specifications are demonstrated in the following requirements critiquing scenario.

## 6: Scenario for critiquing by explanation

This scenario demonstrates active guidance and critiquing during requirements engineering to validate the fundamental behaviour, structure and functional requirements of a simple domain. The critic explains retrieved domain abstractions to a requirements engineer with moderate experience, called Nathalie, to assist requirement specification for a computerised stock control system in a library. Due to space limitations, the scenario focuses on explanations provided by the critic rather than the state of the requirement specification. It shows how heuristics triggered by the dialogue controller manage mixed initiative dialogue with Nathalie and how the critic can provide effective intervention at the right time and level of abstraction. Nathalie's behaviour throughout the scenario mimics the behaviour of real requirements engineering observed during empirical studies [7,13].

**The requirements problem:** Nathalie has determined the need for a stock control system for a lending library. The requirements critic exploits the domain specialisation hierarchy shown in Figure 3. The high-level object system represents the concept of resource containment and is specialised by the OM object system to include resource repletion. This specialised abstraction can be instantiated to all stock control domains, and is linked to three information system models representing functions for stocktaking, purchase authorisation and reporting the status of resources. The scenario has three capture, retrieval and critique cycles which retrieve and explain the OC object system, OM object system and information system models, see Figure 3.

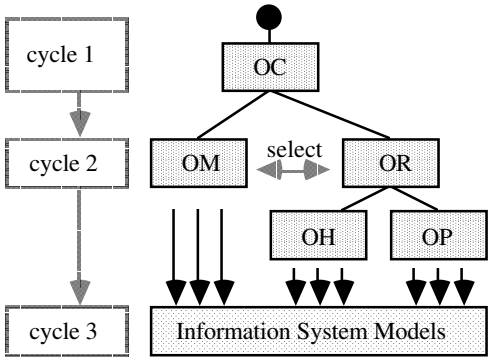


**Figure 3 - example domain specialisation hierarchy. Cycle-1 supports fact capture and domain understanding, cycle-2 fact capture, domain understanding and requirements critiquing, and cycle-3 requirements critiquing**

**Fact capture & matching in cycle-1**: first the requirements capturer acquires facts from Nathalie to enable retrieval of a high-level domain abstraction. Active guidance (heuristic AG2) ensures that these facts discriminate between object system classes. Not surprisingly, Nathalie enters facts which are incomplete and inconsistent [13], although the power of the domain matcher ensures that it retrieves the OC object system as a good fit for stock control [20].

**Domain explanation in cycle-1**: the requirements critic explains the retrieved domain abstraction (AG1) to improve Nathalie's domain understanding, see Figure 4:

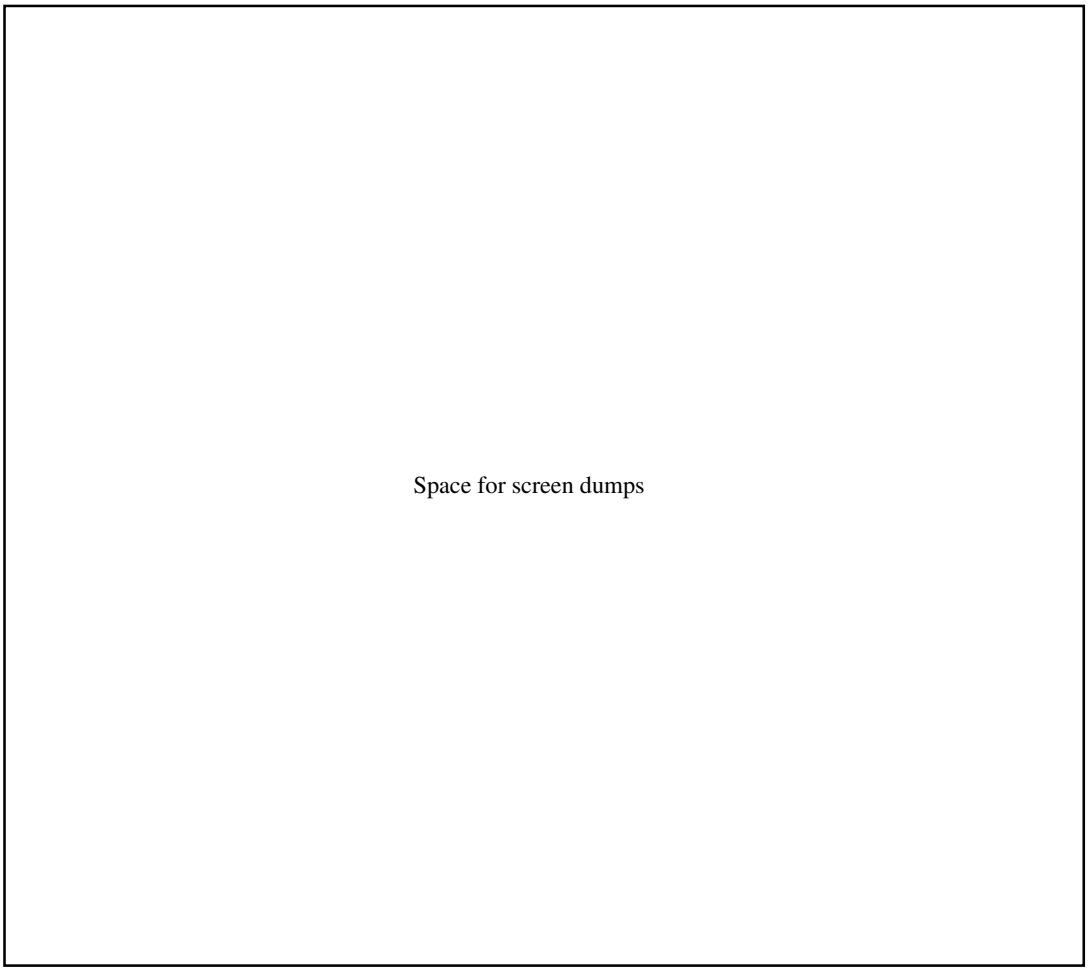


**Figure 4 - explanation of the OC domain abstraction in Cycle-1, inferred object-pair mappings, Telos definitions of objects in the domain abstraction and ConceptBase's Graphbrowser which enables browsing of formal definitions of the domain abstraction**

- the domain abstraction is explained using visualisation and text-based descriptions of domain structure and behaviour (ES1, ES2) formal definitions (ES9) and explanation of object-pair mappings (ES4). Further domain explanation and exploration is available by requesting animation of domain behaviour (ES3);
- the controller selects passive critiquing to emphasise domain understanding (AC6);
- detected problems in the requirement specification which are stored on the problem notepad (AC6).

Empirical studies have revealed that these explanation strategies can lead to effective recognition of domain abstractions [7]. Nathalie agrees the match with the domain abstraction and uses it to aid her domain understanding.

**Domain selection in cycle-2**: the dialogue controller selects active guidance because Nathalie did not extend the requirement specification (AG2). Therefore, the critic requests Nathalie to select the OM or OR abstraction as a specialisation, see Figure 3:

- both domain abstractions are explained using visualisation supported by text-based descriptions (ES1, ES2) and object-pair mappings (ES5);
- explanation of fundamental differences between both abstractions is aided by visualisation of domain structure and behaviour (ES6);
- gradual exposure to candidate abstractions and prototypical examples occurs (ES7).

These explanation strategies have been shown to lead to effective selection between domain abstractions [7]. Nathalie selects the OM domain abstraction. She can then define state transitions and object structures

representing new book supplies which were omitted from the requirement specification and which instantiate facts which discriminate between the domain abstractions (AG4). In this instance, the critic guides requirement specification.

**Domain explanation in cycle-2**: next the critic explains the selected domain abstraction (AG1) in greater detail to improve Nathalie's domain understanding prior to further requirement specification:

- the controller still maintains active guidance (AG2);
- the domain abstraction is explained using visualisation and text-based descriptions (ES1, ES2) and formal definitions (ES9). Nathalie can request prototypical examples of the object system (ES4, ES8).

Nathalie uses these explanations as a starting point for a second bout of requirement specification. She can instantiate features of the explained domain abstraction, thus enabling specification of fundamental domain features in the early stages of the dialogue. Effective domain knowledge transfer has been shown [8,33], however omissions and inconsistencies are still likely. Therefore, the result of this bout is an extended, but incomplete requirement specification.

**Passive critiquing in cycle-2**: the specification is critiqued in light of the selected domain abstraction which is explained in greater detail. Nathalie can browse detected problems including incompleteness, inconsistencies, ambiguities and overspecification using the problem notepad:

- the controller selects passive critiquing because detected problems are not long-term (AC5) and do not block requirement specification in cycle-3 (AC1, AC2);
- Nathalie can browse more prototypical examples representing both software engineering and non-software engineering artifacts (ES8).

The timeliness and level of abstraction of this critiquing is ensured because the explained OM abstraction represents an incremental specialisation of the matched OC abstraction. Nathalie uses her improved domain understanding to resolve detected problems such as incorrect definition of preconditions on state transitions [7]. Problem resolution can also be aided by explanation of detected problems as shown in Figure 5. Nathalie then has the choice to extend the requirement specification as she wishes or follow the requirement capturer's advice and enter requested facts during the next requirement specification bout. She chooses the free-format entry of requirements, defines required system functions including stocktake and purchase authorisation then invokes the critic to validate these requirements.

**Critiquing in cycle-3**: full requirements critiquing is achieved by explaining the OM object system and linked information systems defining likely functional requirements. Matching the requirement specification to specialised domain abstractions enables retrieval of greater domain knowledge at lower levels of abstraction. Narrowing the domain search space in this way enables more effective requirements critiquing. Nathalie can detect problems in her object and information system models:

- the dialogue controller selects active critiquing because the problem notepad detects a large number of problems (AC7);
- guided explanation of information systems and prototypical examples takes place (ES7);
- information systems are explained using formal definitions, visualisation, animation and prototypical examples (ES9, ES2, ES3, ES8).

Examples of these explanations are shown in Figure 5. Nathalie can request tool-based critiquing at any time by rematching the requirement specification to the domain abstractions, therefore iterative explanation and critiquing occurs until Nathalie is satisfied.

## 7: Discussion

Nature's requirements critic exploits a strong, theoretical task and domain models to direct assistance, unlike existing design critics [4,9,34]. This assistance is tempered by control of mixed initiative dialogue driven by the context of the requirements engineering process which enables cooperative requirements critiquing, in contrast to the automated software engineering paradigm [35].

The requirements critic is being extended to validate larger requirement specifications. Object system models are aggregated to represent the behaviour, structure and functions of common applications, for example a typical library application has object hiring, managing, repairing and reordering system models which instantiate to lending, controlling the stock, repairing and reordering library books. Furthermore, Nature's goal theory [31] guides decomposition of complex requirements towards simpler functional requirements, goal states and requirements models for validation by the requirements critic against single object system models. Extended computational mechanisms for critiquing larger requirement specifications will be tested for the next version of the critic.

Requirements engineering involves more complex activities including communication [36] and negotiation

[37]. Indeed, Scacchi [38] argues that requirement specifications are social objects in which people find meaning. Explained domain abstractions can provide common understanding and communication between requirements engineer and end-user, similar to clichés in program understanding [39]. Nature's process model of requirements engineering [40] maximises the benefits of domain abstraction retrieval for requirements engineering.

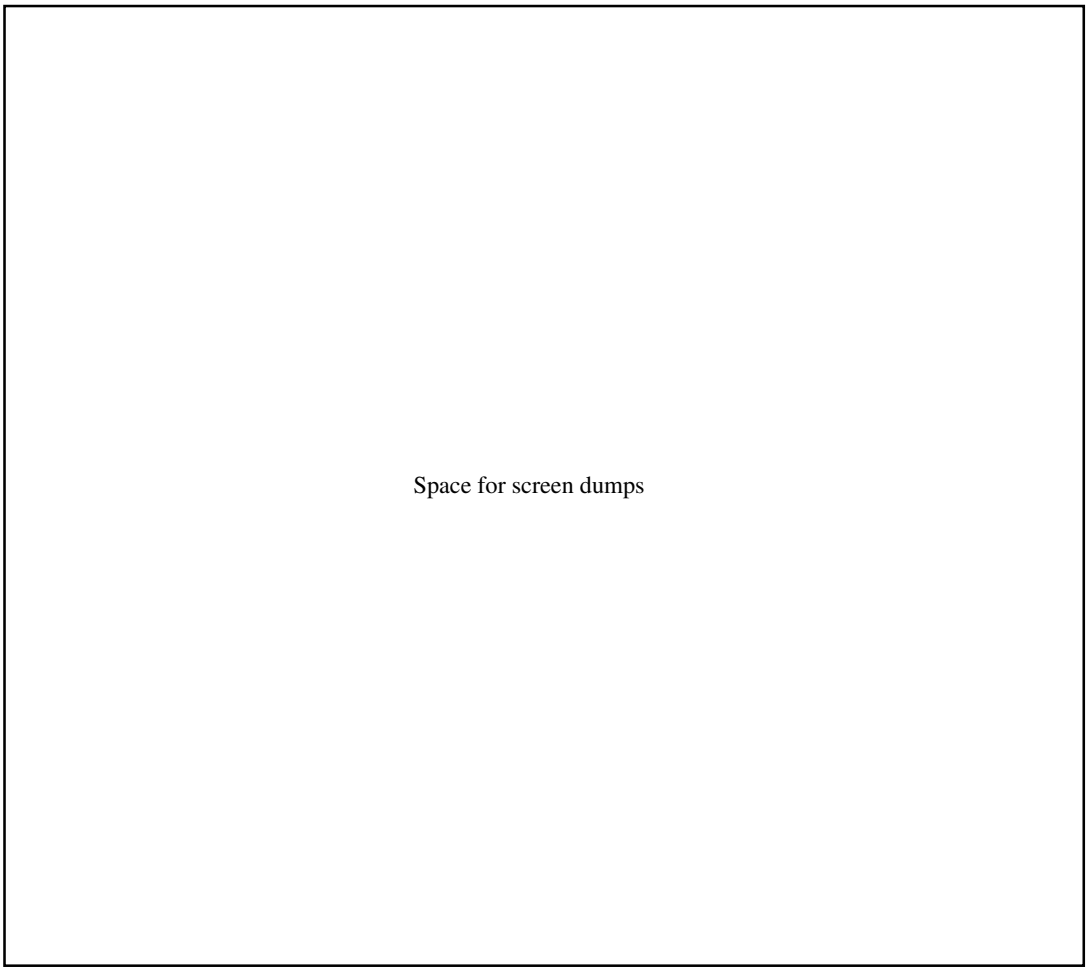


**Figure 5 - requirements critiquing in Cycle-3. Nathalie is presented with explanations and examples of the matched OM domain abstraction, possible functional requirements and detected problem situations from the problem classifier. The critic implements guided exposure to these explanations to aid mental model formation**

## References


[1] Roman G., 1985, 'A Taxonomy of Current Issues in Requirements Engineering', IEEE Computer, April 1985, 14-22.

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

[3] Jarke M., Bubenko Y., Rolland C., Sutcliffe A.G. & Vassiliou Y., 1993, 'Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis', Proceedings of IEEE Symposium on Requirements Engineering, IEEE Computer Society Press, 19-31.

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

[5] Maiden N.A.M. & Sutcliffe A.G., 1992, 'Exploiting Reusable Specifications Through Analogy, Communications of the ACM. 34(5), 55-64.

[6] Fugini M.G., Guggino M. & Pernici B., 1991, 'Reusing Requirements through a Modeling and Composite Support Tool', Proceedings of 3rd International Conference CAiSE'91, Trondheim (Norway), ed. R.

Anderson, J.A. Bubenko & A. Solvberg, Lecture Notes in Computer Science 498, Springer-Verlag, 50-78.

[7] Maiden N.A.M. & Sutcliffe A.G., 1993, 'Requirements Engineering by Example: An Empirical Study', Proceedings of IEEE Symposium on Requirements Engineering, IEEE Computer Society Press, 104-112.

[8] Maiden N.A.M. & Tyndale D.E., 'Reuse of Domain Abstractions During Requirements Engineering: an Explanation is Required', internal document, Department of Business Computing, City University.

[9] Fickas S. & Nagarajan P., 1988, 'Critiquing Software Specifications', IEEE Software, November 1988, 37-47.

[10] Fischer G., Nakakoji K., Otswald J., Stahl G. & Sumner T., 1993, 'Embedding Computer-Based Critics in the Contexts of Design', Proceedings of INTERCHI'93, ed. S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel & T. White, ACM Press, 157-163.

[11] Fischer G., Lemke A.C., McCall R. & Morch A.I., 1991, 'Making Argumentation Serve Design', Human-Computer Interaction 6, 393-419.

[12] Guindon R., 1990, 'Designing the Design Process: Exploiting Opportunistic Thoughts', Human-Computer Interaction 5, 305-344.

[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] Suchman L.A., 1987, 'Plans and Situated Actions: the Problem of Human-Machine Communication', Cambridge University Press.

[15] Maiden N.A.M. & Sutcliffe A.G., 1993, 'The Domain Theory: Object System Definition', Nature Report CU-93-00A, Department of Business Computing, City University.

[16] Johnson W.L., Feather M.S. & Harris D.R., 1992, 'Representation and Presentation of Requirements Knowledge', IEEE Transactions on Software Engineering 18(10), 853-869.

[17] Rosch E., 1983, 'Prototype Classification and Logical Classification: the Two Systems', New Trends in Conceptual Representation: Challenges to Piaget's Theory, edited K. Scholnick, Lawrence Erlbaum Associates, Hillsdale NJ.

[18] Mylopopoulos J., Borgida A., Jarke M. & Koubarakis M, 1990, 'Telos: Representing Knowledge about Information Systems', ACM Transactions on Office Information Systems 8(4), 325.

[19] Jarke M., 1992, ConceptBase V3.1 user manual, RWTH-Aachen, Aachen, Germany.

[20] Maiden N.A.M & Sutcliffe A.G., 1991, 'Analogical Matching for Specification Retrieval', Proceedings 6th Knowledge-Based Software Engineering Conference, IEEE Computer Society Press, 108-116.

[21] De Marco T., 1978, 'Structured Systems Analysis and Specification', Prentice-Hall International.

[22] Maiden N.A.M. & Sutcliffe A.G., 1993, 'The Domain Matcher: Architecture and Algorithms', Nature Report CU-93-00D, Department of Business Computing, City University.

[23] Maiden N.A.M., 1993, 'AIR's Problem Classifier', Nature Report CU-93-00H, Department of Business Computing, City University.

[24] Jarke M., Pohl K., Jacobs S., Bubenko J., Assenova P., Holm P., Wangler P., Rolland C., Plihon V., J. Schmitt, Sutcliffe A.G., Jones S., Maiden N.A.M., Till D., Vassilou Y., Constantopoulos P. & Spandoudakis G., 1993, 'Requirements Engineering: An Integrated View of Representation', Proceedings 4th European Software Engineering Conference, Garmesh-Partenkirchen, ed. I. Sommerville, P. Manfred, Springer-Verlag, Lecture Notes in Computer Science 717, 100-114.

[25] Bolton D., Jones S., Till D., Furber D. & Green S., 1992, 'Knowledge-Based Support for Requirements Engineering', International Journal of Software Engineering and Knowledge Engineering 2(2), 293-319.

[26] Lewis M., 1992, 'Situated Visualisation: Building Interfaces From The Mind Up', Multimedia Review, Winter 1991/Spring 1992, 23-39.

[27] Gick M.L. & Holyoak K.J., 1983, 'Schema Induction and Analogical Transfer', Cognitive Psychology 15, 1-38.

[28] Riesbeck C.K. & Schank R.C., 1989, 'Inside Case-based Reasoning', Lawrence Erlbaum Associates, Hillsdale NJ.

[29] Stasko J., Badre A. & Lewis C., 1993, 'Do Agorithms Assist Learning ? An Empirical Study and Analysis', Proceedings INTERCHI'93, ed. S. Ashlund, K. Mullet, A. Henderson , E. Hollnagel & T. White, ACM Press, 61-66.

[30] Self J.A., 1988, 'Bypassing the Intractable Problem of Student Modelling', Proceedings of ITS-88, June 1-3 1988, Montreal, Canada, 18-24.

[31] Sutcliffe A.G. & Maiden N.A.M., 'Transforming Informal Requirements: Models for Requirements Elaboration and Refinement', internal document, Department of Business Computing, City University.

[32] Maiden N.A.M. & Sutcliffe A.G., 1992, 'Analogously-Based Reusability', Behaviour & Information Technology 11(2), 79-98.

[33] Sutcliffe A.G. & Maiden N.A.M., 1990, How Specification Reuse can Support Requirements Analysis, Proceedings of Software Engineering'90, edited by P. Hall, Cambridge University Press, 489-509.

[34] Puncello P.P., Torrigiani P., Pietri F., Burion R, Cardile B. & Conti M., 1988, 'ASPIS: A Knowledge-Cased CASE Environment, IEEE Software, March 1988, 58-65.

[35] Balzer R., Cheatham T.E. & Green C., 1983, 'Software Technology in the 1990's: Using a New Paradigm', IEEE Computer, November 1983, 39-45.

[36] Curtis B., Krasner H. & Iscoe N., 1988, 'A Field Study of the Software Design Process for Large Systems', Communications of the ACM 31(11), 1268-1287.

[37] Goguen J.A., 1993, 'Social Issues in Requirements Engineering', Proceedings of IEEE Symposium on Requirements Engineering, IEEE Computer Society Press, 194-195.

[38] Scacchi W., 1984, 'Managing Software Engineering Projects: A Social Analysis', IEEE Transactions on Software Engineering 10(1), 18-24.

[39] Rich C. & Waters R.C., 1988, 'The Programmer's Apprentice: A Research Overview', IEEE Computer, November 1988, 10-25.

[40] Schmitt J-R., 1993, 'Product Modeling for Requirements Engineering Process Modeling', Information System Development Process (A-30), ed. N. Prakash, C. Rolland & B. Pernici, Elsevier Science Publishers B.V., 231-245.

[41] Arango G., Schoen E. & Pettengill R., 1993, 'Design as Evolution and Reuse', Proceedings of 2nd International Workshop on Software Reusability (Advances in Software Reuse), IEEE Computer Society Press, 9-18.

[42] Neighbors J.M., 1984, 'The Draco Approach to Constructing Software from Reusable Components', IEEE Transactions on Software Engineering SE-10(5), 564-574.