Supporting Component Matching for Software Reuse

Alistair Sutcliffe and Neil Maiden

Department of Business Computing,
City University,
Northampton Square,
London EC1V 0HB,
U.K.,
Phone: +44-71-253-4399 ext 3420,
E Mail: sf328@uk.ac.city.

Abstract

A mechanism is proposed for analogical matching of specifications for reuse. The process uses generic domain templates with matching heuristics to determine the fit between a source (existing and potentially reusable) specification and the description of the target domain. The design of supporting tools for the matching process is described, with evidence from experimental studies upon which the design was based. The prospects for analogically based software reuse and the requirements for tool support is discussed.

1. Introduction

One of the critical problem in software reuse is finding software components which are appropriate to a new application context and then ascertaining the goodness of fit between reusable components and their target application.

The potential of reusing existing specifications to develop new systems has been brought closer by the CASE tool revolution. It has been suggested that successful specification reuse can assist requirements analysts to develop more complete, consistent and clearly-defined specifications. Intelligent CASE tools require method and domain knowledge to assist the analytic process. For example Ryan [1] reports providing software engineers with method knowledge alone failed to enhance analytic performance. Exploiting the rich seam of domain knowledge captured in reusable specifications is one source of intelligent support which has so far received little attention. Reuse at the specification level offers a new paradigm for requirements engineering by exploiting existing knowledge about a domain in new application contexts. Analogy may be a powerful paradigm which enables matching and retrieval of components for specification reuse, however it has received little attention in the literature [2,3].

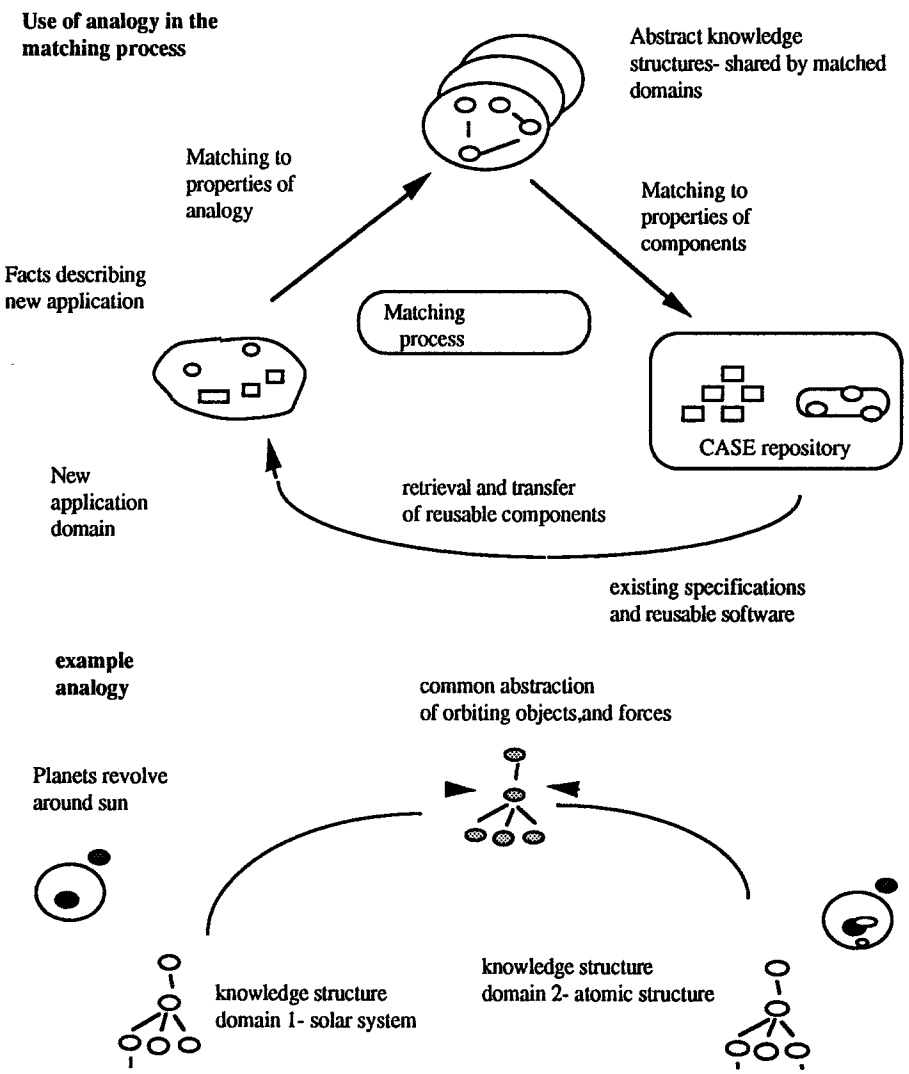
Whilst considerable research is currently focused on the development of knowledge-based CASE tools less attention has been directed to the practical problem of initially eliciting such knowledge. The dilemma is how to economically gather and then subsequently identify the appropriate domain knowledge. Deriving application knowledge through domain analysis can be difficult and time consuming [4,5]. One approach is to describe generic types of systems rather than specific applications. CASE tools endowed with abstract specifications as templates [6], cliches [7] or generalised application frames [8] might provide considerable assistance. However, selection of the most appropriate template is difficult especially as reuse repositories increase in size.

Searching and selecting is one part of the reuse problem; however, equally important is matching to determine the goodness of fit between reusable component(s) and a new application context. This paper reports the development of a tool for specification reuse by analogy which addresses one of the central problems of large scale reuse: the selection and matching of reusable components to a new application context. The paper is organised as follows: first the schema and models of domain abstractions are introduced and illustrated with an example. This is followed by description of the application of these models in a prototype tool for specification retrieval and matching. The structure of the tool and its use are briefly reviewed, followed by a discussion of related work.

2. The Analogical Matching Process

The perceived power of analogy is its potential to retrieve knowledge from one domain and apply it to a different domain [9]. This suggests that analogical reasoning may be able to support reuse across different problem domains, and has the potential to exploit CASE repositories populated with specifications representing a wide variety of applications. Many cognitive theories of analogical mapping exist [10]; however a common factor between the theories is the development of a abstract knowledge structure which represents an inter linked set of facts common to two or more domains. Analogical based reuse therefore aims to identify these abstractions, develop a matching process which can partially automate this task and then a retrieval process which can select appropriate reusable specifications from a CASE repository. The process is summarised in figure 1 which illustrates the concept of structural analogy and its application as a matching mechanism for reusable components. To demonstrate this potential an example of a software engineering analogy is presented.

Fig 1. Summary of the Process of Reuse by Analogy



Theatre Reservation/Course Administration Example

The analogy is between a system for theatre seat reservations and a system supporting applications to a university course. The theatre reservation system allows theatregoers to reserve seats for any performance. They can reserve one or a block of seats, and seats vary in price. Theatre staff use the system to reply to enquiries and to manage reservations. A waiting list is created whenever a performance is over-booked, and theatregoers are transferred from the waiting list to seats when cancellations are made. The context diagram for this theatre reservation system is given in Figure 2(a).

A university course application system manages applications to a full-time and part-time MSc course. The course administrator uses the system to reply to enquiries on place availability and course requirements and to manage the take-up of course places. Candidate students are offered conditional places on either course, which both have an upper limit on total places in any academic year. A waiting list is used for additional students who cannot be offered places immediately. Students on the waiting list have first option on any places which become available due to cancellations. The diagram for this system is given in Figure 2(b).

Figure 2. Context diagrams for the Course Administation and Theatre booking Systems

Figure 2(a) Context diagram for the Theatre Reservation System

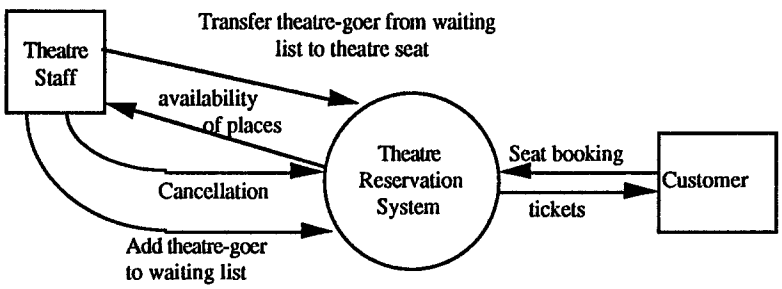
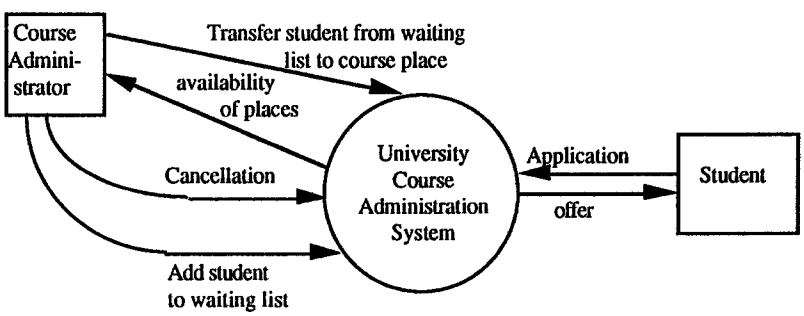


Figure 2(b) Context diagram for the Course Administration System



The two context diagrams (Figures 2(a) and 2(b)) demonstrate the potential reuse which can be exploited from this analogy. Reuse is also possible during more detailed analysis, between processes (e.g. reservation of theatre seat\course place), data stores (e.g. theatre seat\course place) and external agents (theatregoer\student). Although in different domains, the two systems share significant surface features (e.g. reservations, waiting lists, places) which assist analogical recognition and understanding. A common abstraction links the two domains, involving a resource (seats, places) being allocated to clients (theatregoers, student applicants).

Further analogies with the theatre reservation and course administration examples can be identified, for instance car rental and airline seat reservation share significant similarities which make specification reuse possible. Analogy enables mapping between domains, although to be effective a set of domain abstractions is required. Reuse of generic templates has been suggested by several authors [6,10], however templates have a limited reuse value because details have to be omitted whereas implemented specifications contain additional domain knowledge which can be reused. We therefore see analogy and domain abstractions are a mechanism for facilitating reuse of matching specifications rather than providing reusable components per se.

3. Schema and models of domain knowledge.

The analogy matching process utilises partial domain knowledge to reason about the links between applications. A set of domain abstractions have been devised to support this process. The set of abstractions, so far, has been targeted on information system rather than real time domains. The approach is based upon the following propositions:

(i) Application domains are organised into classes sharing general features. Each class level adds more specialised features to differentiate it.

(ii) Domain classes are distinguished by a small number of key determining features.

(iii) Most software engineering problems can be ascribed to one of a tractably small set of domain classes.

Domain knowledge is modelled as classes which are specialised by addition of further knowledge to achieve appropriate targeting in new domains. The schema of knowledge types, as shown in figure 3, is used to define a set of abstract domain models. Currently seven types of knowledge structure are used:

(a) The structure of the domain. This is knowledge of the objects within domains and the sets they naturally fall into. The prediction is that people will perceive sets which pertain to physical structures in the real world. Hence the term 'containment' may be a familiar cue for domain structure, e.g. stock objects are-contained-in warehouse; students are-contained-in a school.

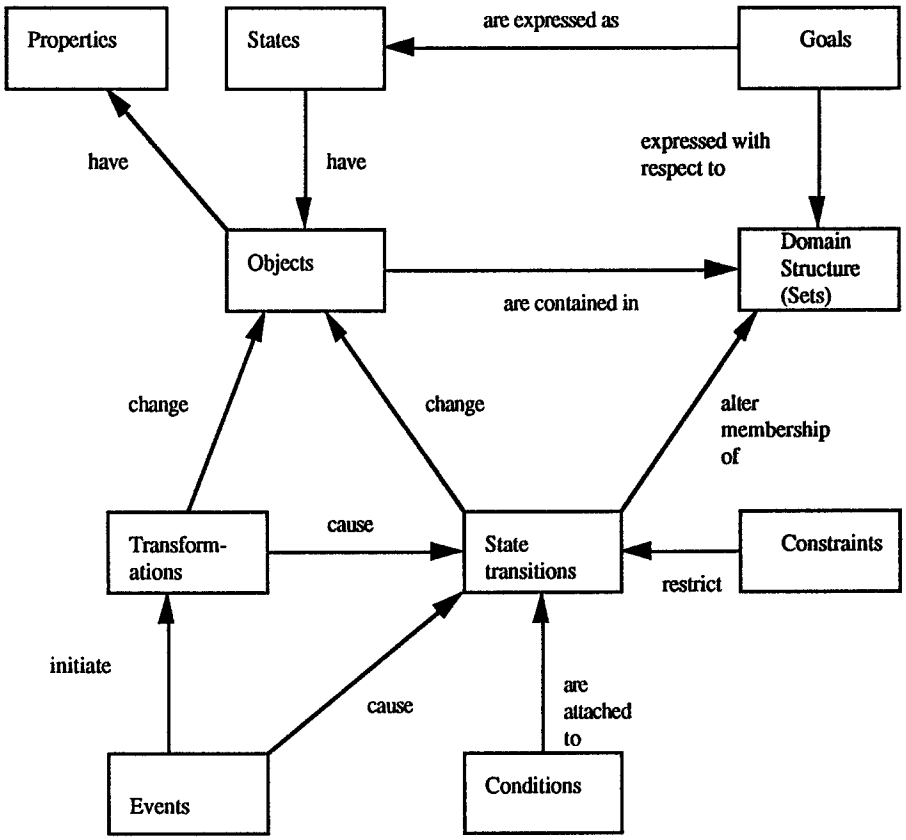
The system structure describes the object sets implicated in input and output (i.e. changes in object status across the system boundary) and any major physical entity sets within the system itself.

(b) State transitions which cause objects to change set membership with respect to the domain structure. Any event which causes an object to change its set membership is a key determining feature. Inter-domain class differences are critically determined by these state transitions more than any other feature.

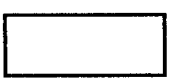
(c) Events: these trigger state transitions. Internal and external events (with an origin outside the system) are recognised. This enables description of the scope or boundaries of the system.

(d) Object properties. High level properties of an object, focusing in particular on their role within the system (e.g. resource, mechanism), movement, and other abstract or concrete qualities. Properties are essentially a type definition of objects, which although currently not formalised, are amenable to such treatment.

Figure 3. Meta-schema of knowledge types and their relationships



The schema is not intended to be a formal representation of domain knowledge, instead it represents the organisation of predicates which describe facts about a domain model.



= schema primitives

= relationships between primitives, the arrow denotes the locus of effect.

(e) Purpose of the system: This is the goal or solution state that the system exists to achieve. Goals and statements of purpose are difficult to describe formally because they are expressed in natural language, differ between users and may be expressed at a variety of levels and with different degrees of precision. Consequently goals are expressed as the system state which must be achieve to satisfy a linguistically expressed goal.

(f) Constraints: These are 'negative rules' which define the object states which the system must prohibit or prevent from happening, i.e. future states which must not happen

(g) Conditions: Stative information or values held in attributes of an object and other non-object states (e.g. time, duration) which are evaluated before a state transition can occur.

(h) Transformations: Procedures and algorithms which cause a state change in the system. Transformations are triggered by events and may result in a state transition of an object in the domain structure.

The components of the schema and their relationships is summarised in figure 3.

Using this schema abstract domain models are defined as an inter-related set of facts, i.e. goals are linked to states, conditions to transitions, transitions to objects and domain structure. The abstract domain classes are differentiated by actions leading to state changes of objects with respect to parts of the system structure. System structure is a set theoretic concept of object-set membership linked to the transactional purpose of the system. To illustrate the concept, in a renewable resource management abstraction, of which stock control is a concrete example; sets of objects (products) are held by suppliers, in an inventory (stock) and with customers (delivered products). A non-renewable resource management abstraction, of which library loans is an example, can be distinguished from a renewable resource management abstraction (e.g. stock control) by the key transition of return. An informal representation of the object-structure-transition semantics of two abstract domain models is shown in figure 4. The return action causes the object (library book) to change state from on-loan to a resource-available state whereas return in the stock control abstraction is an infrequent and non mandatory action (see fig 4).

Within each abstract domain class specialisation occurs by addition of schema components. For instance non renewable resource management applications are specialised by further transformations to differentiate between booking and hiring systems. Other schema components are used to corroborate differences between sub classes. Object types, which have been promoted as determinants of analogical reuse by [6], are categorised according to their role in the domain, for instance stock items and airline seats both act as resources. This provides another determinant for analogical matching and similarity evaluation.

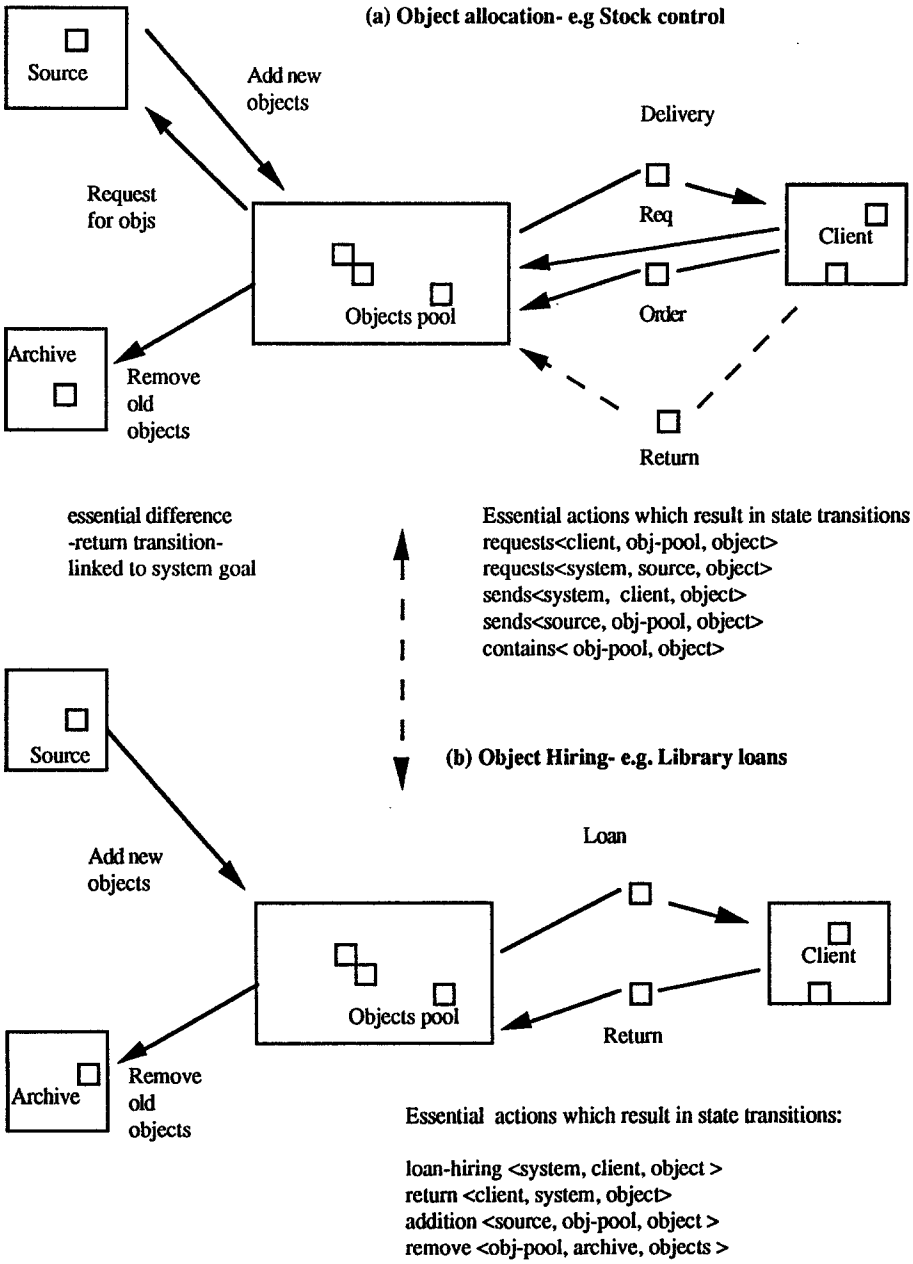
System purpose has been identified as important determinant of abstract models [11], so goal related semantics are defined in terms of states which the system attempts to achieve or maintain. Returning to the example domains, the stock control system attempts to maintain a minimum quantity of items-in-stock, while the library system attempts to maintain stock constancy so all the books-on-loan are returned. Activities (i.e. a set of actions, or algorithms) leading to key state transitions are determined by system purpose.

Transformations are a set of operations leading to a state change of objects within a single procedure. In other words transformations are composed of operations must execute in a single uninterrupted sequence to change the object's state. Transformations may result in state transitions in set membership in the system structure; however, conceptually they alter an object's status. Events caused by the status change may then result in object transition in the system structure. To illustrate the concept, common transformations in information systems are, scheduling, allocation by constraint satisfaction, searching for objects, reservation, etc. These result in conceptual-stative changes to objects such as 'reserved, found, sorted, scheduled.' Transformations are similar to 'methods' in object oriented specifications or procedures in structured methods.

Other knowledge types (triggers, conditions) play a supplementary role in differentiating domain classes. Equivalent state transitions can be distinguished by their triggering events. Each transition is either triggered by the information system or by events beyond that system, which have important influences on the information system. For instance, an allocation action in the airline booking domain can be differentiated from allocation of stock to orders in a warehouse domain because the former allocation is triggered by the information system while the latter is not.

Users are predicted to recognise objects and purpose as the most 'natural' descriptors of domains, hence these features should be more easy to elicit [12]. However, given the ambiguity inherent in many teleological descriptions, system structure and state transitions are predicted to be the most reliable determinants of domain identity. A composite of system structure and related transitions could be taken as the domain/class 'key' in database terms. Matching rules and heuristics enable selection of the appropriate domain abstraction for a set of predicates describing a new application. Use of multiple heuristics and a rich schema for the knowledge base enables sophisticated search strategies to be generated thus avoiding the computational inefficient, and often intractable, approach of linear searching multi variate sets of properties, typified by faceted classification.

Fig 4 Abstract Domain Models: Object Allocation and Object Hiring Abstractions



The domain abstractions are represented semi-formally as sets of typed Prolog predicates. This enables semantic networks to be constructed to store each abstraction a composite knowledge structure of facts, as defined by the schema, and relationships between those facts. The matching engine then searches on the fact primitives in the models, their first order and second order

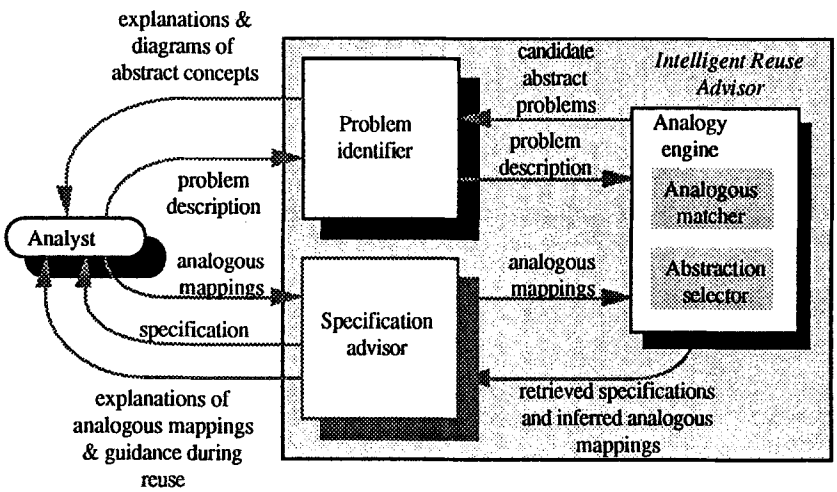
relationships with powerful algorithms which utilise the syntactic and semantic properties of the models.

Effective specification reuse requires intelligent tool-based support for the matching process in large scale repositories. However successful reuse also requires an intelligent reuse advisor to prevent specification copying and enhance understanding of both the analogy and the reusable domain [13]. The intelligence of this advisor is based upon cognitive task models of how software engineers do and should reuse specifications. The intelligent reuse advisor (Ira) has three major components which are examined during the remainder of this paper.

4. The Reuse Advisor

Specification reuse involves three processes: categorisation of a new problem, selection of candidate specifications belonging to the same category and customisation of the selected analogous specification to the new domain. Ira has three main components which support these processes (see figure 5):

Figure 5 - overview of interaction of the three components which constitute Ira (darkened indicates implemented in the prototype)



* The problem identifier obtains a description of a new target problem from the software engineer then explains retrieved abstract domain models so that the categorisation of the new problem can be validated.

* The specification advisor controls interaction between Ira and the software engineer during selection and customisation of an analogous specification. The diagnostic module attempts to identify software engineers' misconceptions about the analogy so that appropriate support can be given. These misconceptions are inferred from a catalogue of error types derived from empirical study [12,13]. The explanatory module acts as the analyst's guide and teacher during specification reuse. The analyst is led to reuse a specification by strategies which encourage understanding and transfer of the analogy, and assisted with explanations of analogous mappings of the reusable specification inferred by Ira.

* The analogy engine reasons with critical problem features to match new problems to abstract domain models, retrieves analogous specifications of the same category and reason alongside the software engineer during specification customisation. The role of the analogy engine is constrained by the domain knowledge available to it. There have been a number computational models of analogical and case-based reasoning [14,15,16,17], however they assume perfect and complete knowledge of the domains. The analogy engine can reason about many domains because it

is equipped with partial domain knowledge in an abstract form, although it is limited by the critical problem features represented in the abstract domain models which it possess (see section 3).

The problem identifier, specification advisor and analogy engine components are described in more detail to demonstrate how analogous specifications are retrieved and reused by Ira.

4.1 The Problem Identifier

The problem identifier supports elicitation of a new problem description and explanation of abstract domain models retrieved by the analogy engine. We view matching of new applications to domain abstractions as an iterative process of retrieval and understanding involving:

* elicitation of facts about the new problem,
* retrieval of abstract domain models to match those facts by the analogy engine, and
* explanation of the abstract domain models to the software engineer.

Initial interaction with the software engineer aims to elicit key problem features which map to critical features of abstract domains. The problem identifier provides a predefined set of predicates to model relationships between domain objects, so the software engineer is required to partially abstract their model of the domain during description. Descriptions, justifications and examples are used to explain retrieved abstractions so that the software engineer can select or reject them as representative of the new problem, as illustrated in fig 6 which shows a sample dialogue session. Explanation strategies help the software engineer understand and abstract concepts which critically determine an analogy, while the specification advisor explains analogous mappings between the new problem and the selected analogous specifications.

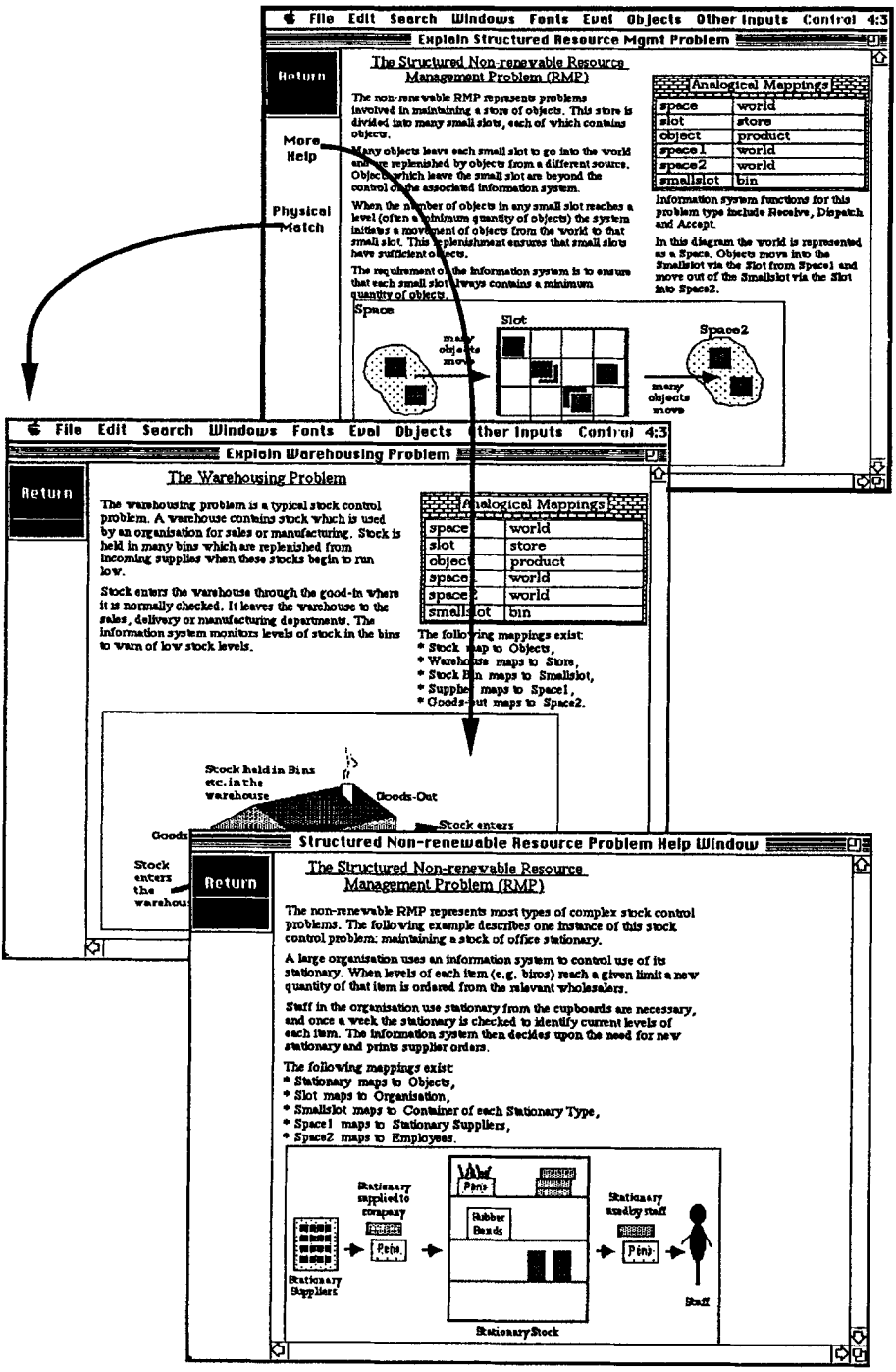
4.2 The Analogy Engine

Currently the analogy engine employs structure-matching [18] and heuristic-based reasoning to identify analogical matches between a problem description and a set of known domain abstractions [12]. Structure-matching identifies an interrelated network of analogical mappings [9] between a problem description and candidate domain abstractions using a structural coherence algorithm similar to the Structure-Mapping [18] and Analogical Constraint Matching Engines [19]. The analogical matcher maps semantically-equivalent predicates representing critical knowledge structures identified by our model of software engineering abstractions, including state transitions and object structural knowledge (see Figure 3). The outcome from this process is retrieval of one or more candidate abstract domains for the new problem.

The analogy engine also employs heuristics which discriminate between abstract domain models. Hierarchical structuring of the abstract domain models ensures that the analogy engine only attempts to match likely abstractions for a new problem. The fact acquisition dialogue requests fact-types motivated by the theory to help discriminate between domains, e.g. critical state transitions and object set membership. Search is thus driven by predicted attributes of superordinate classes in the abstract domain hierarchies and then refine down a selected hierarchy to match an appropriate domain model as further facts are acquired from the user. Each domain model in the hierarchy inherits all the features of its parent and specialises it to represent a sub-type of that software engineering problem. Practical experience with the analogy engine revealed that the similarities between abstract domain models at lower levels in the hierarchy indicated the need for a more finely-tuned retrieval mechanism. The abstraction selector differentiates between candidate abstractions using a set of heuristics which identify critical differences between abstract domain models at each sub-level in the hierarchy. The heuristics calculate the degree of difference between two abstract domains as a percentage of the total differences possible.

Successful categorisation of the target problem is followed by retrieval of reusable specifications belonging to the same domain abstraction using similar analogical matching techniques. The result of this matching process is the retrieval of one or many candidate reusable specifications ranked by their similarity with the target problem. Ranking is achieved by matching analogous features shared by the new and reusable domains, for example, similarities between the physical structure of both domains.

Figure 6 Explanation windows representing a retrieved abstract domain class for a stock control system. The three windows, from top to bottom, represent: the critical abstraction for the analogy; a likely physical application for the domain; an alternative, analogical example of the domain



4.3 The Specification Advisor

This work is currently in the specification stage, so it is reported to give an overall picture about how the reuse assistant may function. Implementation may change some of the details. The specification advisor must support the software engineer during two tasks: (i) specification selection from several candidates, and (ii) specification customisation to fit the new domain. Both tasks require the software engineer to have a good understanding of the specifications, so the specification advisor will explain relevant analogies to the software engineer. In addition, during specification selection, the differences between candidate analogies will be described. Tutoring strategies to support the analyst during both tasks are being developed, although only strategies which support understanding and reuse of a single specification are described in this paper.

The specification advisor will explain and guide the software engineer during analogical comprehension and transfer using strategies derived from empirical studies of expert software engineers during successful reuse of analogous specifications [20,21]. The system employs plan-based, context-independent reuse strategies to ensure that it has control over its environment. A single, prescribed strategy guides inexperienced analysts to reuse specifications, while explanatory and error-correcting tactics support analyst's individual differences within each step of the reuse strategy.

Prior to specification reuse the software engineer will be encouraged to develop a basic analogical understanding necessary to enhance and maximise effective customisation of the specification. This analogical understanding concentrates attention on critical domain objects, functional goals and the boundaries of the problem, and builds on software engineer's understanding of the abstract domain developed during problem categorisation. The software engineer is assisted by explanatory dialogues and narrative descriptions of the reusable problem, and diagramming aids are provided to graphically represent the analogy. Software engineers mappings can be evaluated by using analogical mappings inferred by the analogy engine and the empirically-based error library [22]. Subsequent feedback from the tool can be used to generate a correct understanding of the analogy.

The specification advisor will control the software engineers' access to the specification to encourage further analogical understanding and inhibit mental laziness. Mental laziness is discouraged by consideration of all reusable components and by exposing only the relevant, analogous components in the specification. Learning of individual analogical components is iterative following other studies [23,24,25,26] which suggest an iterative approach promotes more effective problem understanding. Coupling the explanatory dialogue with gradual exposure of the specification seems to be the most effective strategy to encourage analogical understanding.

To summarise, the specification advisor attempts to guide and assist the software engineer by dialogues based on cognitive models of successful reuse behaviour. It employs intelligent tutoring techniques to assist software engineers to overcome the problem of understanding an unfamiliar specification, and uses context-independent strategies to lead the analyst through the complex transfer and testing stages.

5. Implementation of the Prototype Reuse Advisor

A partial prototype of Ira as shown in Figure 6 has been implemented in LPA Prolog on an Apple Macintosh FX. The problem identifier and analogy engine components were evaluated during several studies. The analogy engine was populated with 10 hierarchically-structured abstract domain models supported by approximately 30 heuristics identifying critical differences between them. It has proved effective at retrieving abstractions given only partial or ambiguous problem descriptions. User studies with the problem identifier revealed the need for visualising critical problem features, so the problem elicitation dialogue was modified to encourage more problem visualisation. Subsequent studies indicated that greater problem visualisation enhanced problem description and permitted Ira to retrieve the correct domain abstractions in 75% of trials.

6. Discussion

Specification level reuse can help to overcome the considerable difficulties experienced by inexperienced software engineers during the early stages of software development [27,28]. Formation of mental models is necessary to understand a domain, however as Young [29] and Sein [30] have reported, mental model formation can be error-prone and hard. In addition Sutcliffe and Maiden, [20]

found that initial problem scoping was important in determining success for inexperienced software engineers. Reusable specifications could reduce the analyst's mental load during model formation. Evaluating candidate designs in new scenarios is a key element in successful software development [31,32], hence analogy may help development of alternative scenarios. Reusing specifications will inevitably encourage a more prototypical approach to requirements analysis, as suggested by Luqi [33] and Balzer et al. [34]. Prototyping in turn may encourage more and frequent evaluation of requirement specifications, implying more indirect benefits from specification reuse.

Successful reuse can also enrich the software engineer's own knowledge base, providing experience necessary to solve similar problems or explain further analogous reusable specifications. Viewing CASE environments as both problem solving and learning tools may ease the skills shortage, providing knowledge gained from experienced software developers to help less experienced software developers practice reuse and requirements engineering. However tool support for matching, retrieval and understanding is vital. Analogical based matching enables reuse to effected across domains at the problem level, whereas generic application frames [8] are restricted to evolutionary style reuse within a domain. It also enables active support for matching which can not be achieved by faceted classification schemes [35,36].

However a cautionary note should be sounded. Dependence on specification reuse could discourage innovation, and bring about the mental laziness which we are seeking to avoid. The reuse advisor tool is designed to discourage such practices, based on extensive empirical studies of software engineering's behaviour and reasoning during reuse [13,27]. These studies have proved invaluable in anticipating problems such as mental laziness, as well as providing models and strategies to encourage good design practice. Another open question is the effectiveness of transfer in analogically mediate, or other reuse. Our current system matches on structural knowledge combined with other attributes of what could be expected from the semantics of conceptual models in development methods (e.g. event, entity, relationship, etc. primitives). So far we can achieve matching and transfer of specifications which approximate to a medium sized entity relationship diagram. However transfer of the more dynamic aspects of systems is more problematic [37] so further investigation is required to support transfer of different types of knowledge contained within specifications. We intend to elaborate our model of domain abstraction to deal with matching at different levels of granularity and for different conceptual model components.

Future research directions are two-fold. First, the matching process is dependent on the set of domain abstractions. The completeness and validity of known abstract domain models must be evaluated through case studies of software engineering problems encountered in industrial organisations. Further validation will be achieved by formal representation of the models in a suitable language, e.g. Z [38] to test for isomorphism, consistency and redundancy. The coverage of the current set of 10 domain abstractions is being increased by study of new system types (e.g. real time, process control applications). Secondly, further evaluation of the problem identifier is necessary to assess its usability when eliciting complex application descriptions from inexperienced software engineers. Finally, the specification advisor must be implemented to assess the effectiveness of explanation tactics and reuse strategies described in this paper. Evaluation of such a prototype will provide an important feedback of a collaborative assistant approach to specification reuse, with implications for future research directions.

Acknowledgements

Neil Maiden is a SERC supported research student. The authors wish to thank students on the MSc in Business Systems Analysis and Design who helped evaluate the Reuse Advisor.

References

[1]. Ryan K., 1988, Capturing and classifying the software developers expertise, Proceeding of the International Workshop on Knowledge-based Systems in Software Engineering, UMIST, March 1988.

[2]. Finkelstein A., 1988, Re-use of formatted requirements specifications, Software Engineering Journal, September 1988, pp 186 - 197.

[3]. Karakostas V., 1989, Requirements for CASE tools in early software reuse, ACM SIGSOFT Software Engineering Notes, Vol 14, No 2, pp 39 - 41.

[4]. Arango G., 1987, 'Evaluation of a Reuse-based Software Construction Technology', internal document, Department of Information and Computer Science, University of California, Irvine.

[5]. Prieto-Diaz R. and Freeman P, 1987, Classifying software for reusability, IEEE Software, January 1987, pp 6 - 16.

[6]. Harandi M.T. and Lee M.Y., 1991, Acquiring Software Design Schema: A machine learning perspective. In Proceedings of 6th Conference on Knowledge Based Software Engineering, pp239-250, Syracuse, NY, Sept 1991.

[7]. Reubenstein H.B., 1990, 'Automated Acquisition of Evolving Informal Descriptions', Ph.D. Dissertation (A.I.T.R. No. 1205), Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

[8]. Constantopoulos P., Jarke M., Mylopoulos J., Vassiliou Y. (1991) Software Information Base: A Server for Reuse. Submitted for Publication. Technical Report, FORTH Res Inst, Univ of Heraklion, Crete.

[9]. Gentner D., 1983, Structure-mapping: a theoretical framework for analogy, Cognitive Science 7, pp 155 - 170.

[10]. Russel S.,1988, Analogy By Similarity, Analogical Reasoning, Kluwer Academic Publishers, 1988

[11]. Maiden N.A.M., 1991, Analogy as a paradigm for specification reuse, Software Engineering Journal 6(1), pp 3 - 15.

[12]. Maiden N.A.M. & Sutcliffe A.G., in press, Analogous matching for specification reuse. To appear in CACM

[13]. Sutcliffe A.G. and Maiden N. (1990); Specification reusability: Why tutorial support is necessary. In Proceeding SE 90, BCS Conference on Software Engineering. Ed Hall P.A.V., pp 489-509, Cambridge Univ Press.

[14]. Alterman R., 1986, An adaptive planner, Proceedings of AAAI-86, 5th National Conference on Artificial Intelligence, Philadelphia, pp 65 - 69.

[15]. Hammond K.J., 1986, CHEF: A model of case-based planning, Proceedings of AAAI-86, 5th National Conference on Artificial Intelligence, Philadelphia, pp 267 - 271.

[16]. Hall R.P., 1989, Computational approaches to analogical reasoning: a comparative analysis, Artificial Intelligence 39, pp 39 - 120.

[17]. Schank R.C. and Leake D.B., 1989, Creativity and learning in a case-based explainer, Artificial Intelligence 40, pp 353 - 385.

[18]. Falkenheimer B., Forbus K.D. & Gentner D., 1989, The structure-mapping engine: algorithm and examples, TR No. UIUCDCS-R-87-1361, Dept Computer Science, University of Illinois at Champaign.

[19]. Holyoak K.J. & Thagard P., 1989, Analogical mapping by constraint satisfaction, Cognitive Science, pp 295 - 355.

[20]. Sutcliffe A.G. and Maiden N. (in press); Analysing the Analyst: Cognitive models in software engineering. To appear in International Journal of Man machine Studies.

[21]. Maiden N.A.M. & Sutcliffe A.G., manuscript in preparation(b), Cognitive models of expert software reusers.

[22]. Johnson W.L., 1990, Understanding and debugging novice programs, Artificial intelligence

42(1), pp 51 - 97.

[23] Lewis M.W. and Anderson J.R., 1985, Discrimination of operator schemata in problem solvers, Journal of Experimental Psychology: Learning, Memory and Cognition, Vol 8, No 5, pp 484 - 494.

[24]. Miyake N, 1986, Constructive interaction and the iterative process of understanding, Cognitive Science 10, pp 151 - 177.

[25]. Burstein M.H., 1988, Incremental learning from multiple analogies', in Analogica (Research Notes in Artificial Intelligence), edited by A.E. Prieditis, Pitman, London, pp 37 - 62.

[26]. Jansweiller W., Elshout J.J., and Wielinga B.J., 1989, On the multiplicity of learning to solve problems, in "Learning and Instruction. European Research in an International Context. Vol II & III", ed. by H. Mandel, E. de Corte, N. Bennet and H.F. Friedrich, Oxford: Pergamon.

[27]. Sutcliffe A.G. and Maiden N.A.M. (1991), Analogical software reuse: Empirical investigations of analogy based reuse and software engineering practices. Acta Psychologica 78(1-3), pp 173-197.

[28]. Maiden N.A.M and Sutcliffe A.G. (in press); Analogously based reusability, to appear in Behaviour and Information Technology.

[29]. Young R.M., 1983, Surrogates and Mappings: two kinds of conceptual mappings for interactive devices, in "Mental Models", ed. by D. Gentner and A.L. Stevens, Lawrence Erlbaum Associates, pp 35 - 52.

[30]. Sein M.W., 1988, Conceptual models in training novice users of computer systems: effectiveness of abstract vs analogical models and influence of individual differences, Ph. D. Thesis, School of Business, Indiana University, January 1988.

[31]. Abelson B. and Soloway E., 1985, The role of domain experience in software design, IEEE Transactions on Software Engineering, Vol SE-11, No 11, November 1985, pp 1351 1360.

[32]. Guindon R. & Curtis B., 1988, Control of cognitive processes during software design: What tools are needed ?, Proceedings of CHI '88 conference: Human Factors in Computer Systems, edited by E. Soloway, D. Frye and S.B. Sheppard, pp 263 - 269, ACM Press.

[33]. Luqi, 1989, Knowledge-based support for rapid software prototyping, IEEE Expert, Winter 1988, 9-18.

[34]. Balzer R., Cheatham T.E. and Green C., 1983, Software technology in the 1990s: using a new paradigm, IEEE Computer, November 1983, pp 39 - 45.

[35]. Boldyref, C., Elzer, P., Hall, P., Kabber, U., Keilman, J. and Witt J.,. 1990. 'PRACTITIONER: Pragmatic support for the reuse of concepts in existing software'. In Proceedings SE 90, BCS Conference on Software Engineering. Ed Hall P.A.V., pp 574-591, (Cambridge Univ Press. 1990)

[36]. Prieto-Diaz R., 1991, 'Implementing Faceted Classification for Software Reuse', Communications of the ACM 34(5), 88-97.

[37]. Sutcliffe A.G. (1991), Object oriented systems analysis: The abstract question. In proceedings of IFIP working group 8.1. Conference on Object oriented approaches in Information System Development. Eds Van Assche F., Moulin B., and Rolland C., pp 23-37, North Holland

[38]. Spivey, J.M.,1988, The Z notation: a Reference Manual. Prentice-Hall International, Englewood Cliffs, NJ