Analogical Specification Reuse During Requirements Analysis

Neil Arthur McDougall Maiden

submitted for part of examination for
Doctor of Philosophy

Department of Business Computing,
School of Informatics,
City University,
London.

July 1992

Table of Contents

Thesis Tables

The thesis has the following tables:

Thesis Figures

The thesis has the following figures:

Acknowledgements

I would like to thank my parents for suffering a fulltime student for so long, and Nathalie for understanding why I was never there at weekends.

I would also like to thank my supervisor, Dr Alistair Sutcliffe, for his vigorous assistance over the last three and a half years. Finally I would like to acknowledge the support and assistance of the following during my time at City University: Ms Karen Coates, Mr Paul Collins, Dr Kay Dudman, Mr Peter Faraday, Dr ZhongSu Fu, Dr Simon Grant, Prof. Owen Hanson, Ms Jacqui Hetherington, Dr Allen Long, Mrs Wendy Martin, Ms Uma Patel, Mr Whim Penny, Mr Mark Springett & Ms Lynne Wolseley.

Declaration

The author grants powers of discretion to the University Library to allow this thesis to be copied in whole or in part without further reference to the author.

Statement of Contribution

This disclaimer is to state that the research reported in this thesis is primarily the work of the author and was undertaken as part of his doctoral research. Referenced papers of which the student is not the sole author represent the role of the supervisor in the research, to direct the work and enhance the written style of these papers

Abstract

This thesis investigates analogy as a paradigm for retrieving, understanding and customising reusable specifications during requirements engineering. Cooperation between software engineers and support tools is necessary for effective analogical reuse. Retrieval uses a computational implementation of analogical reasoning to search and match many reusable specifications. On the other hand understanding, transferring and adapting specifications requires cooperation between the tool and software engineer. Cooperative support was designed for less-experienced software engineers with most to gain from successful specification reuse. Deliverables from this research have implications for software engineering, artificial intelligence, cognitive science and human-computer interaction.

Specification retrieval is founded on a framework of software engineering analogies. This framework includes a set of domain abstractions describing key facts about software engineering domains. A computational model of analogical reasoning which matches domain descriptions to these abstractions was designed, implemented and evaluated during user studies with a prototype reuse advisor. An intelligent dialogue acts as a front-end to this retrieval mechanism by acquiring key domain facts prior to retrieving domain abstractions. This dialogue was designed from empirical studies of software engineering behaviour during requirements capture and modelling.

Design of support tools for specification understanding and transfer was based on cognitive task and reasoning models of software engineering behaviour during analogical reuse and mental models of analogical understanding. Two empirical studies of inexperienced software engineers identified problematic mental laziness manifest as specification copying. A third study of expert software engineers who successfully reused specifications identified strategies for effective reuse. Detailed findings from all three studies informed the design of tool-based support for specification understanding and transfer. Findings also have implications for the design of tools to support other requirements engineering activities.

Chapter 1

1: Overview: Specification Reuse by Analogy

The possibility of reusing existing requirement specifications to develop new systems has been brought closer by the computer-aided software engineering (CASE) revolution. Increased automation of software development tasks suggests that repositories containing artifacts from the analysis, design and coding phases are all available for reuse. Indeed, reusing products from the early phases of software development appears to be a natural progression from the code and object libraries which have dominated research into software reuse over the last 30 years. Code reuse has always promised gains in the productivity and quality of software. Successful reuse of specifications and designs early in the development life cycle can increase these gains since omissions and errors in new specifications can be identified and corrected at an early stage (Boehm 1981). Productivity can also be increased by cross-application transfer from a repository containing specifications developed for many different domains. This thesis proposes the reuse of existing requirements specifications to assist specification of new systems, thereby maximising the payoff from specification reuse by focusing on inter-domain reuse early in the software development life cycle.

Analogy is proposed as an effective paradigm for requirements specification reuse. Existing reuse paradigms such as faceted classification schemes (e.g. Prieto-Diaz & Freeman 1987) and abstract templates (e.g. Harandi & Young 1985) have proved ineffective when scaled up to support large-scale reuse between partially-similar systems. Analogy, on the other hand, allows the transfer of knowledge from a previous solution to a current problem solving episode (Hall 1989). This transfer is justified by a common underlying knowledge structure rather than by syntactic similarities between domains (Carbonell 1985, Gentner 1983), thus supporting reuse between specifications which share a common underlying goal and domain structure, regardless of the syntactic differences between their applications. Analogy has been proposed elsewhere as a paradigm for large-scale reuse (e.g. Finkelstein 1988, Dubisy & Lamsweerde 1990, Miriyala & Harandi 1991), however the processes and knowledge required to reason analogically during specification reuse are poorly understood. This thesis proposes a model of analogical specification reuse based on investigations of the process by which reuse may best be achieved, and a definition of analogical matching between software engineering domains. This definition identifies critical determinants of software engineering analogies as a basis for retrieving and explaining specifications. Tool support for subsequent understanding and transfer of retrieved specifications is founded on logical and empirical models of how specification reuse may best be achieved. First however, problems inherent in requirements engineering were investigated as a background to this research.

1.1 Requirements Capture: the Pitfalls

Requirements capture is well-recognised as a complex and error-prone process (e.g. Reubenstein & Waters 1991, Roman 1985, Meyer 1985) encompassing social, communication and technological issues. Requirement specifications are defined as stating the desired functional characteristics of some component independent of any actual realisation, while design specifications describe the component's implemented internal structure and behaviour (Roman 1985). Requirements specifications differ from design specifications in that they state current problems, desired goals and facilitate understanding while design specifications render physical and logical structures that implement these requirements and prescribe the system's functionality. This focus on integration in the real world makes requirements engineering a much *softer*, more difficult task than other steps in software development. For instance, this view of requirements engineering can preclude the *complete* capture of all relevant domain knowledge, thus limiting the expertise and capabilities of any intelligent requirements engineering toolkit.

Requirements engineering can be viewed as a social and communication as well as a technological process. Scacchi (1984) suggests that requirements specifications are inherently social objects in which people find meaning. The development of a requirements specification must be understood as the outcome of a complex social as well as technological process. Curtis et al.'s (1988) extensive studies of 17 large software projects revealed important communication difficulties between software engineers, users and business managers during the early stages of software development:

> *'developing large software systems must be treated, at least in part, as a learning, communication and negotiation process. Much early activity on a project involved learning about the application and its environment, as well as new hardware, new development tools and languages, and other evolving technologies'.*

Curtis and his colleagues also identified the thin spread of domain knowledge between software developers which left the fate of projects in the hands of powerful and knowledgeable individuals. They propose that, at the broadest level, software development environments must become a medium of communication to integrate people, tools and information (e.g. Olson et al. 1990). A social study of requirements capture and investigation of communication difficulties during requirements engineering, although important, are outside the scope of this thesis. Rather, this research investigates specification reuse as one paradigm for overcoming errors, omissions and ambiguities in individual understanding of requirements specifications.

Requirements specification has proven to be a problematic activity. Meyer (1985) identified reoccurring patterns of deficiencies in informal requirements documents routinely produced by industry which he labelled as the *seven deadly sins of the specifier* (see Table 1.1). Roman (1985) also identified characteristics of requirements specifications which are problematic for effective requirements capture, including incompleteness, inconsistency and ambiguity. Formalisation of requirements specifications has been proposed as one solution to these problems (e.g. Meyer 1985, Reubenstein & Waters 1991), however, many of the implicit premises of formal methods do not hold in requirements engineering. Indeed, it may be productive to maintain inconsistencies and incompleteness, for instance conflict and imprecision can foster creativity. Furthermore, it is unlikely that a single formalism can capture the richness of requirements knowledge. Instead, tool support for requirements engineering should encourage intuition and communication among end-users and software developers possibly through continued use of informal structured analytic notations such as data flow diagrams (De Marco 1978). Such a paradigm differs from formal approaches in its recognition of the *softness* of the unstructured and informal task. The denial of the existence of a complete formalism does not mean that it is impossible to formalise requirements engineering. Rather it should be recognised that tool support for requirements capture needs a two-way transition between formal and informal representations during both requirements acquisition and validation.

Noise:	*the presence in the text of elements irrelevant to the problem at hand;*
Silence:	*the existence of problems not covered by the requirements specification, i.e. incompleteness;*
Overspecification:	*inclusion of implementation-level data in specifications;*
Contradiction:	*inconsistency between definitions of two or more elements that define system features in an incompatible way;*
Ambiguity:	*specification definitions that make it possible to interpret a problem feature in at least two different ways;*
Forward Reference:	*the presence of specification components that use features of the problem not defined until later in the specification;*
Wishful Thinking:	*the presence of specification elements that define problem features in such a way that a candidate solution cannot be realistically validated with respect to those features.*

Table 1.1 - Meyer's seven deadly sins in requirements specifications

1.2 A Definition of Requirements

A definition of requirements is needed before reuse of requirements specifications can be investigated. This thesis defines functional requirements as stating the desired functional characteristics of some component independent of its implementation. Functional requirements can be of three types:

- problem-driven requirements are caused by failure of the old system which must be corrected in the new system;
- goal-driven requirements identify new system features to be added in the new implementation;
- constraints describe events, domain states and functions which must or should not occur in the new implementation.

All three types of requirement are expressed using natural language statements rather than formal definitions of needs. Current requirements engineering research has also focused on non-functional as well as functional requirements, however non-functional requirements are poorly understood, thus investigation of analogical reuse of non-functional requirements in this thesis is an unrealistic research aim. Rather, this thesis investigates analogical specification reuse to assist more effective capture and modelling of functional requirements.

It is also important to distinguish between functional requirements and reusable specifications. Specifications held in CASE repositories represent domain knowledge about the problem rather than detailed functional characteristics of the desired system, hence analogical specification reuse provides domain knowledge to assist the completion and disambiguation of functional specifications. Thus, an important emphasis of this research remains the validation of requirements specifications by resolving ambiguities, incompleteness and inconsistency in their domain descriptions. As such, it aims to improve the definition of requirements which are implicit in the description of software engineering domains rather than a direct attempt to capture, model and validate functional requirements. An overview of the proposed analogical reuse scenario is shown in Figure 1.1 on the following page.

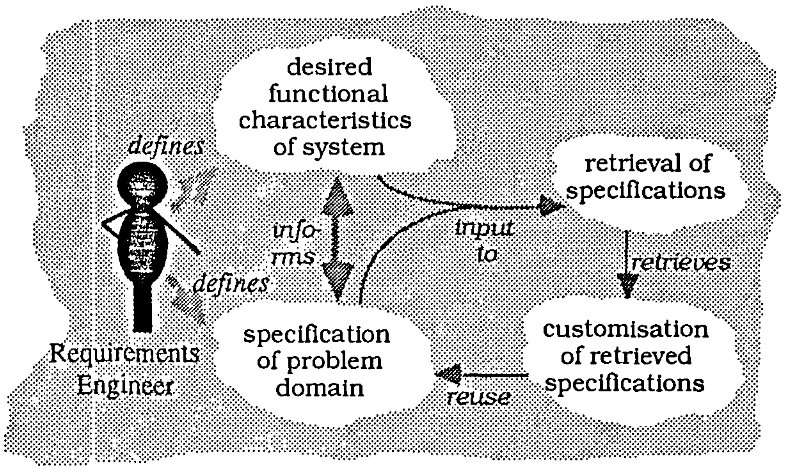

desired functional characteristics of system
defines
retrieval of specifications
info-rms
input to
retrieves
defines
specification of problem domain
customisation of retrieved specifications
Requirements Engineer
reuse

Figure 1.1 - overview of specification reuse during requirements engineering

1.3 Specification Reuse: Intelligent Support

This thesis proposes specification reuse as a paradigm for supporting the requirements engineering phase of software development. Previous research has identified the need for both domain and method knowledge to support the early phases of software development (Frenkel 1985, Fickas 1987, Curtis et al. 1988, Fickas & Nagarajan 1988, Puncello et al. 1988, Loucopoulos & Champion 1989, Sharp 1991). At the broadest level, domain knowledge represents all aspects of the problem domain being analysed (Arango & Freeman 1985) as well as high-level solutions in these domains. On the other hand method knowledge refers to structured analysis techniques such as SADT (Ross 1977) and SSA (De Marco 1978), including procedural steps which guide the analytic process (e.g. the Structured Systems Analysis (SSA) method) and notations (e.g. data flow diagrams) which represent the artifacts from that process (De Marco 1978). In many existing intelligent systems which support requirements engineering (e.g. Lubars & Harandi 1986, Harandi & Lubars 1985, Tsai & Ridge 1988, Loucopoulos & Champion 1989 Katsouli & Loucopoulos 1991, Johnson 1991, Lee & Harandi 1991a, 1991b) domain and method knowledge are stored separately, however, empirical studies have revealed that expert software engineers remember and recall abstract and concrete specifications (e.g. Guindon & Curtis 1988, Guindon 1990) in which domain and method knowledge are integrated in common reusable patterns. This would suggest that reuse of specifications expressed using structured notations is one form of domain and method knowledge which could 'simulate' expert analytic performance. In particular, specification reuse can exploit analogical similarities to identify ambiguity, incompleteness and inconsistency in new specifications as well as improve productivity during the costly requirements engineering phase of software development. They can also provide a basis for communication by describing complex concepts in terms of well-understood, existing specifications.

1.3.1 Different Forms of Specification Reuse

Specification reuse can occur in several forms, although it is only likely to benefit software engineers if specifications are reused in their original format, otherwise the effort required to 'tinker' with specifications beforehand (i.e. *design for reuse*) may offset benefits from reuse. Furthermore, considerable effort is necessary to derive generic designs and templates (e.g. Harandi & Young 1985, Fugini et al. 1991) while the difficulty of modelling domains to support multiple instances of reuse (Arango 1988) may discourage software reuse at its outset. On the other hand, specifications currently held in CASE tools often describe large applications implemented using millions of lines of code. Furthermore, additional specifications for reuse can be derived by reverse engineering code of existing systems. These specifications are documented using structured analysis notations such as data flow or process structure diagrams (e.g. De Marco 1978, Jackson 1983 respectively) as well as narrative descriptions of entities and processes such as logical process outlines (Cutts 1987). As such, reusable specifications provide a rich seam of domain and method knowledge not available using alternative paradigms. Indeed, specification reuse can provide the greatest payoff from minimal preparation effort compared with alternative paradigms for intelligent requirements engineering support (e.g. Dardenne et al. 1991).

1.3.2 Alternative Techniques

Domain analysis and domain modelling are currently the most popular techniques for eliciting and representing knowledge to support the requirements analysis process (Arango & Freeman 1985), despite being difficult and time-consuming to achieve. Previous studies (e.g. Arango 1988) have shown that domain analysis requires experienced domain analysts for lengthy periods, so considerable investment in time and effort is needed to provide long term benefits to the software development process. Unfortunately such investment may not be justifiable in current software engineering environments dominated by project deadlines and shortages in experienced staff. Specification reuse can overcome this knowledge acquisition bottleneck by exploiting domain knowledge held in readily-available specifications. Furthermore, this knowledge can be represented as logical specifications which may be more readily applicable to the requirements specification process. Knowledge- or transformation-based (e.g. Katsouli & Loucopoulos 1991, Barstow 1985) paradigms represent domain knowledge as isolated rules which ignore the complex knowledge structures of software engineering domains. On the other hand, reusable specifications incorporate complex and context-dependent domain knowledge and designed solutions which can be reused in similar software development scenarios. As a result, they are a more available and applicable source of domain knowledge during requirements engineering.

1.3.3 Advantages of Effective Specification Reuse

Successful reuse of specifications can support the requirements engineering process in many ways:

- the consistency, completeness and clarity of a system's functional requirements can be improved through reuse of specifications developed for similar applications;
- non-functional requirements may possibly be reused based on similarities between functional requirements, for example two systems with equivalent functionality and structure may also have similar interface,, performance and operating constraints. Further research beyond the scope of this thesis is required to examine reuse of non-functional requirements;
- reusable specifications can provide domain-specific methods with which to analyse new domains. Ryan (1988) reports that providing software engineers with method knowledge alone failed to enhance analytic performance. Domain-specific knowledge in reusable specifications may be used to guide and control analytic processes more effectively, by highlighting critical system requirements, functionality, structure and scope (e.g. Puncello et al. 1988);
- reusable specifications can provide solution templates for specifying new systems. This may be particularly beneficial for inexperienced software engineers because empirical studies of program design tasks suggest that novice software engineers do not have preformed memory schemata to recall and are unable to structure and scope the domain space effectively (e.g. Adelson 1984, McKeithen et al. 1985, Koulek et al. 1989, Rist 1991);
- prototyping during system design has long been advocated as a paradigm for software development (e.g. Crinnion 1991, Luqi 1989, Luqi & Ketabchi 1988). Specification reuse encourages this paradigm through provision of analogical specifications from which to prototype new applications;
- reusable specifications can also be used to evaluate system specifications developed elsewhere. Empirical studies of analytic behaviour (e.g. Fickas et al. 1988, Guindon & Curtis 1988) suggest that experienced software engineers evaluate solutions successfully through scenario-based simulation, so reusable specifications could

provide inexperienced software engineers with alternative scenarios for testing new specifications;

- matched reusable specifications can also constrain the search space of reusable design modules by suggesting: (a) candidate design modules to meet functional requirements in the specification, and (b) the structure and relationship between design modules. In addition, Lange & Moher (1989) and Green et al. (1992) report that locating design or code modules during reuse is difficult, so analogically-matched specifications may be a useful guide for locating design objects.

These potential advantages over existing reuse paradigms justify analogical specification reuse as a paradigm worthy of investigation. The remainder of this chapter examines the analogical specification reuse paradigm in more detail. First, an example of analogical reuse is presented to demonstrate the scale and potential pitfalls of the paradigm, then a high-level model of the analogical reuse process is proposed to identify the three major aims of this thesis. Finally these aims are examined in terms of the necessary theoretical and empirical bases for tool support.

1.4 Analogy in Specification Reuse

Specification reuse across domains involves analogical reasoning. However, research in artificial intelligence (e.g. Hall 1989), cognitive psychology (e.g. Gentner 1983) and systems management (e.g. Silverman 1983, 1985) suggests that analogical reasoning may be complex and difficult to achieve. Difficulties which can arise are best demonstrated by a simple example which will be referred to and expanded on throughout this thesis. This example analogy supports reuse between a *source* theatre reservation specification and a *target* university course administration specification.

1.4.1 An Analogy between a Theatre Reservation and University Course Administration Problem

A 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 overbooked, and whenever a cancellation is made people are transferred from the waiting list and allocated to the seats made available. The context diagram (De Marco 1978) for this theatre reservation system is given in Figure 1.2(a).

A university course administration system manages applications to a full-time and part-time MSc in Systems Analysis. The course administrator uses the system to reply to enquiries on place availability and course requirements, and to manage the annual takeup of course places. Candidate students are offered conditional places on either course, each of which has an upper limit of places during an 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 context data flow diagram for this system is given in Figure 1.2(b).

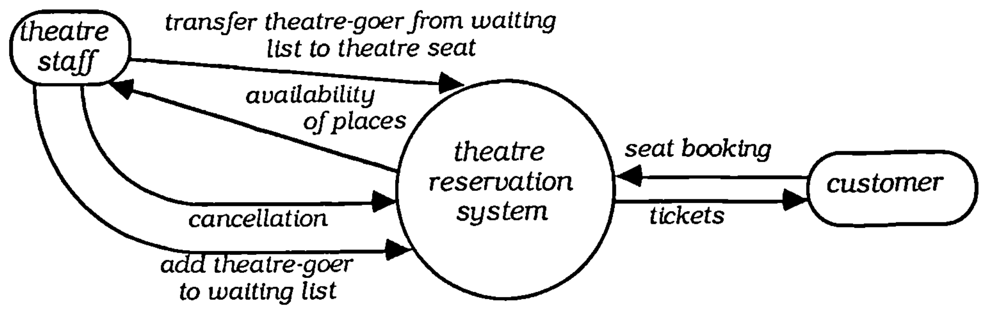

theatre staff
transfer theatre-goer from waiting list to theatre seat
availability of places
cancellation
add theatre-goer to waiting list
theatre reservation system
seat booking
tickets
customer

Figure 1.2(a) - context data flow diagram for the theatre reservation system

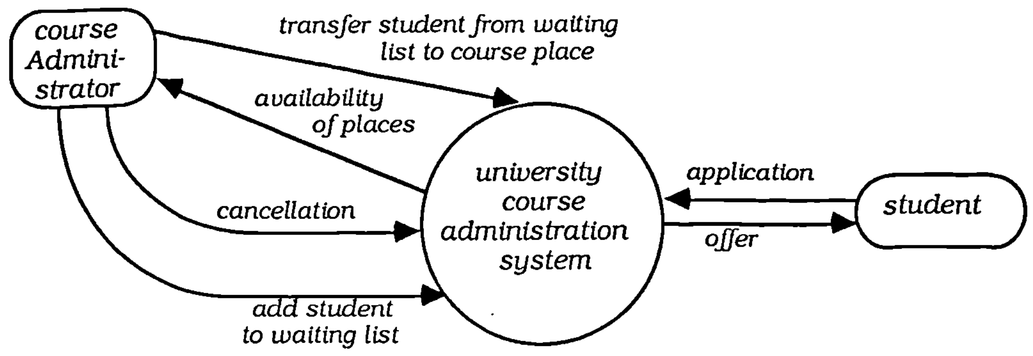

course Admini-strator
transfer student from waiting list to course place
availability of places
cancellation
add student to waiting list
university course administration system
application
offer
student

Figure 1.2(b) - context data flow diagram for the course administration system

Data flow diagrams illustrate the potential reuse which can be exploited from this analogy. Reuse is also possible at more detailed levels within data flow diagrams, between processes (e.g. reservation of theatre seat/course place), data stores (e.g. theatre booking/course application) and external agents (theatregoer/student), and between specifications represented using other notations (e.g. entity-relationship models, entity life histories etc.). These mappings indicate a good analogical match between the two domains.

The theatre reservation/university course administration analogy demonstrates how analogical matching can maximise the potential payoff from reuse by supporting reuse across applications. An important feature of analogy is that similarities occur between deep knowledge structures rather than syntactic similarities (Gentner 1983). For instance, the analogical match between the two domains can occur because both involve the allocation of resources to meet prespecified constraints, although other reasons also exist. Therefore, the first major aim of this research was to identify critical determinants of software engineering analogies as a basis for their retrieval, selection and explanation. This work is elaborated in chapter 3.

1.4.2 A Contrast Between Analogical Reasoning & Other Software Reuse Strategies

This thesis investigates analogy as a paradigm for large-scale specification reuse, so the analogical reasoning process was contrasted with that of conventional software reuse typified by keyword retrieval (e.g. Wood & Sommerville 1988), faceted classification (e.g. Prieto-Diaz & Freeman 1987), abstraction (e.g. Fugini et al. 1991) and object-orientation (e.g. Lenz et al. 1987). Analogical reasoning involves three major steps (Hall 1989), namely recognition of the analogy, comprehension of the analogy by developing and justifying analogical mappings between the source and target problems and transfer of the source analog to the target across these mappings. Each of these steps mirrors a major step in existing software reuse paradigms, namely retrieval of candidate components for reuse, selection of the best-fit candidate, and adaptation of that component to fit the new problem. Correspondences between the analogical reasoning and software reuse paradigms are shown in Figure 1.3. Each is examined in detail.

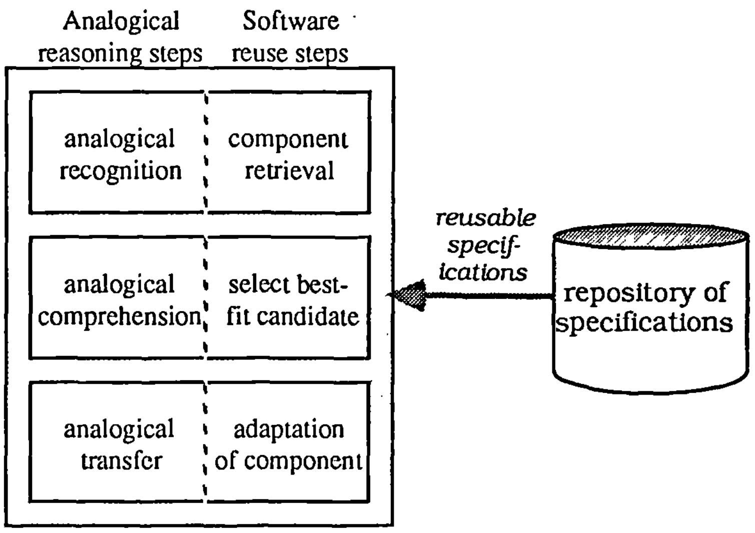


Figure 1.3 - overview of the 3 major phases of analogical specification reuse

Analogical Recognition

Analogical recognition matches each specification in the repository to key facts about the target problem. It is more complex than retrieval of formally-defined code modules (e.g. Burton et al. 1987) since requirements analysis often begins with an incomplete, inconsistent and ambiguous statement of needs. In addition, key features may be difficult to identify, for example the course administration and theatre reservation specifications may be matched by similarities between any number of features, including functional goals (*maximise allocation of theatre reservations/applications*), domain structures (*one theatre contains many seats, one course has many places*) and specification components (*allocated seats, allocated places*). One solution advocated by this thesis is to view recognition as an iterative process of: (i) retrieving candidate specifications; then (ii) elaborating the current statement of system needs in light of the improved domain understanding from the reusable specification. First however, critical determinants of software engineering analogies must be identified as a basis for analogical recognition.

Analogical Comprehension

Retrieved specifications must be understood before successful reuse can occur. However, effective analogical understanding requires extensive knowledge of the source and target domains, for example consider the knowledge required to justify component reuse in the example analogy. Inevitably, reuse of poorly understood specifications will lead to poor transfer of reusable components manifest as omitted components and incorrect mappings. The need to effectively understand reusable artifacts has also been stressed in current software reuse research (e.g. Biggerstaff 1987, Fischer 1987, Elzer 1991, Huff & Thompson 1991). However, few concrete conclusions exist about the support needed for component understanding. Therefore, a second aim of this thesis is to determine how specifications and analogical links between them may be understood.

Analogical Transfer

Analogical transfer involves reuse of specification components to construct a new system. It is equivalent to modifying white-box code modules during software reuse (Biggerstaff & Richter 1987), during which the functionality and structure of the module are understood and customised. However, specification reuse across applications requires extensive customisation to fit a reusable specification to the target domain, for example most data flow diagram components in Figure 1.2 must be changed during reuse in the example analogy. The importance of adaptation during analogical transfer should not be underestimated. However, little is known of how to adapt reusable components to fit new domains, so the third major goal this thesis investigates how effective transfer of analogical specifications can best be achieved.

Differences between Analogical Reasoning and Software Reuse

The three major software reuse steps correspond to the main steps during analogical reasoning, although major differences exist within each step. Analogical recognition requires iterative specification retrieval and elaboration until the needs statement is sufficiently complete. Subsequent understanding of retrieved specifications may be a difficult and error-prone task, similar in nature to comprehension of large and unfamiliar programs (e.g. Pennington 1987), but with additional, complex analogical mapping between domains. Analogical transfer differs from the black-box reuse proposed by many researchers (Dusink & Hall 1991) because it needs extensive adaptation of the specification. Previous approaches to reuse emphasised software construction from many black-box components, thus as a result of this and the low level of current reuse practice, issues affecting customisation of reusable software are poorly understood. These major differences suggest the need for a radically new paradigm which is investigated in the three major aims of this thesis, namely identification of key determinants of software engineering analogies and effective strategies for analogical comprehension and transfer.

1.4.3 How to Achieve Specification Reuse by Analogy

This thesis proposes a human-oriented reuse paradigm in the form of tool support which cooperates with an individual software engineer during specification reuse, see Figure 1.4. Previous research suggests that analogical reasoning is knowledge-intensive (Russell 1989), for instance the theatre reservation/university course administration example needs extensive knowledge of the source and target domains. Some source knowledge may be captured in the repository, however, knowledge of the target domain often is only known to the end-user or the software engineer. As a result, this thesis hypothesises that analogical reasoning requires extensive human involvement to be effective. Unfortunately previous studies (e.g. Gick & Holyoak 1980, 1983, Cheng & Holyoak 1985, Cheng et al. 1986) suggest that analogical reasoning during problem solving is difficult, so tool support is proposed to assist this analogical reasoning. The division of work between tool support and the software engineer depends upon their respective abilities to retrieve, comprehend and transfer reusable specifications analogically, which in turn is determined by their knowledge and reasoning capabilities, as shown in Figure 1.5. This thesis proposes that analogical reasoning will be shared between the software engineer and tool support throughout the three phases of specification reuse, although partitioning of the workload varies within each phase.

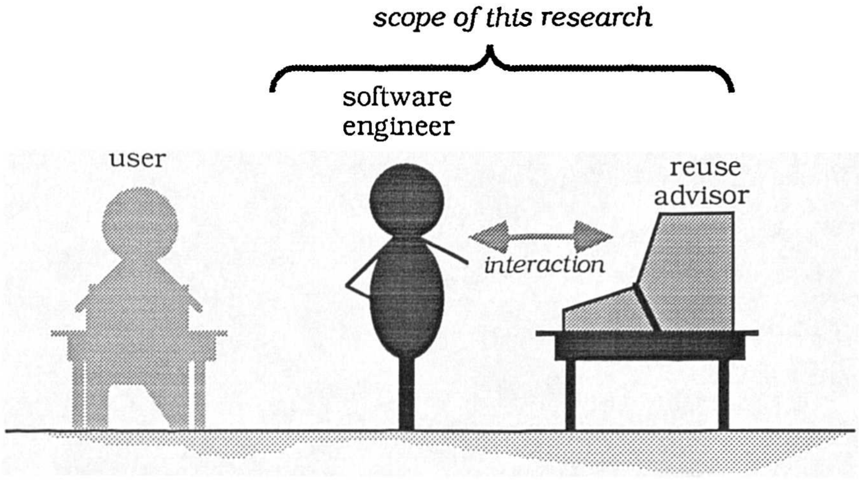


Figure 1.4 - scenario of use with reuse advisor

Analogical retrieval must be a tool-based activity because the search space for analogically-matched specifications is too large for unguided browsing by software engineers, especially if successful uptake of CASE technology leads to organisation-wide specification repositories. As a result, a computational model of analogical reasoning will be needed to retrieve specifications. It will differ from existing models such as the SME (Falkenhainer et al. 1989) and ACME (Holyoak & Thagard 1989) in that it uses minimal domain knowledge during retrieval to maximise the leverage from the resulting analogical match. Some human involvement will be needed during this retrieval phase, to provide domain knowledge about the new problem and agree candidate analogical matches. On the other hand, the limited availability of domain knowledge and ability to reason analogically indicate that more human involvement will be necessary during analogical comprehension and transfer of specifications. Specifications must be understood to ensure their correct selection, transfer and adaptation. Leaving analogical comprehension and transfer to the software engineer permits reuse from many different specification notations including entity-relationship modelling, data flow diagrams or English text, thus maximising the payoff from the analogy. This cooperative paradigm is examined more closely below during an overview of the analogical reuse process.

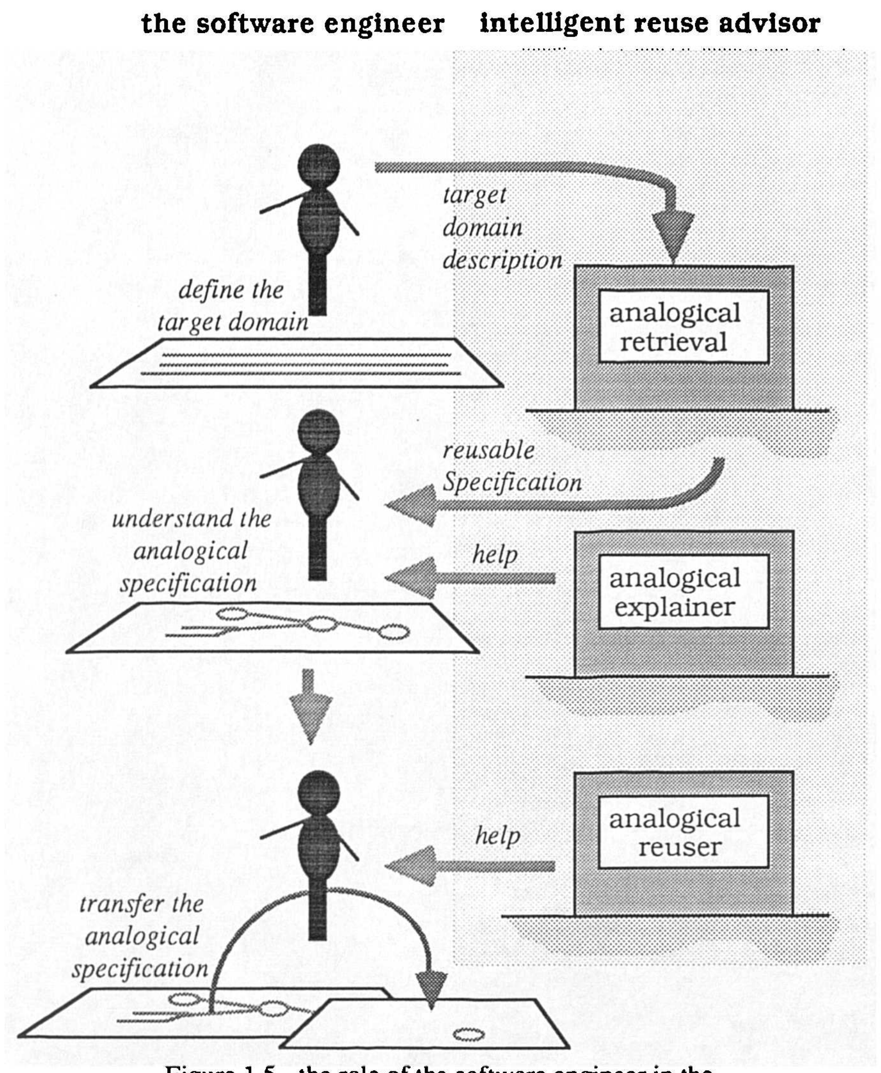


Figure 1.5 - the role of the software engineer in the three phases of analogical specification reuse

1.5 An Intelligent Advisor for Reuse

This thesis proposes an intelligent advisor to support a single software engineer during all three steps of analogical reuse. Three important theoretical and empirical questions were investigated as the major aims of this research: why do analogies between software engineering domains exist, how can software engineers understand such analogies, and how do software engineers reason analogically to exploit reusable specifications:

- tool-based retrieval and explanation of specifications must be founded on a definition which states why analogies exist. The cognitive psychology and artificial intelligence disciplines have proposed many different and often contradictory theories of analogical reasoning. Software engineering analogies will be examined using these existing theories to determine a set of critical analogical determinants for tool-based retrieval and explanation of analogical specifications;

- the tool must also work cooperatively with a single software engineer during analogical specification reuse. To be effective this cooperation must be founded on how software engineers reason during requirements capture and analogical reuse. Unfortunately little is known of how software engineers analyse complex problems or reuse unfamiliar analogical specifications. Empirical investigations of software reuse are limited to single-user studies (Langer & Moher 1989) while cognitive issues during the systems analytic and design processes have received only small scale empirical research (namely Vitalari & Dickson 1983, Guindon & Curtis 1988, Fickas et al. 1988, Guindon 1990). Similarly, scant research has been paid to determining effective strategies and explanation tactics for analogical reuse. This thesis proposes that design of effective tool support must be founded on empirically-derived models of working and reasoning practices of software engineers during specification reuse. This empirical basis for tool design will be derived from empirical studies of reuse of analogically-matched specifications.

An overview of the problems facing analogical specification reuse are shown in Figure 1.6. Three major components of an advisor are envisaged. First, the problem identifier will interact with the software engineer to acquire key facts about a domain. Specification retrieval will be achieved by entering these facts into a computational implementation of analogical reasoning known as the analogy engine. Analogical specifications retrieved from the repository will then be passed to the specification advisor. This must assist the software engineer to understand each analogy, select the most appropriate specification to reuse and customise that specification to fit the target domain.

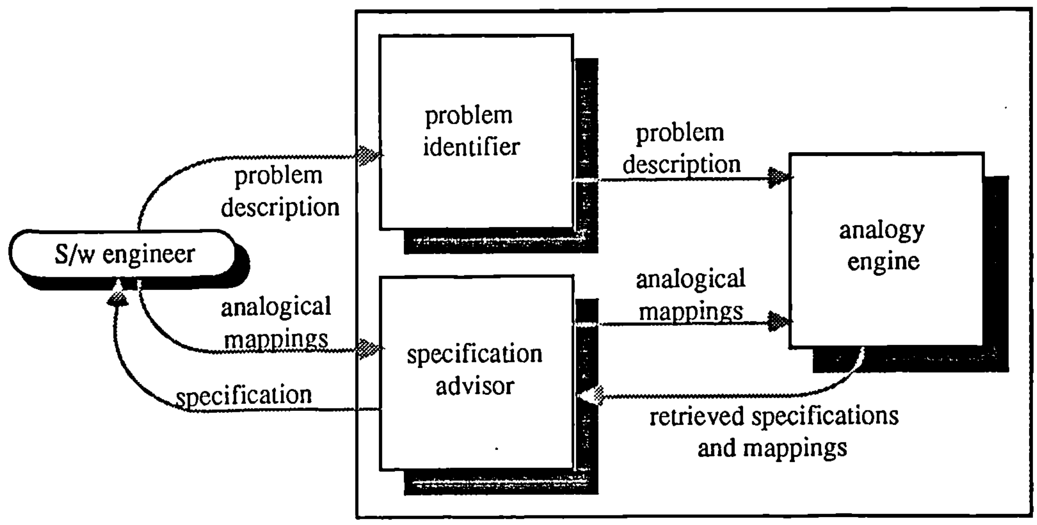


Figure 1.6 - an overview of the analogical reuse problem

1.6 Thesis Rationale & Organisation

The remaining six chapters describe how the three major aims of this research were met, see Figure 1.7. Chapter 2 investigates the applicability of existing reuse paradigms to specification reuse. Examples of analogical specification reuse demonstrate the need for an alternative paradigm for specification reuse and cooperative tool support for reusing specifications. The remaining chapters are outlined in terms of its contribution to development of the intelligent advisor.

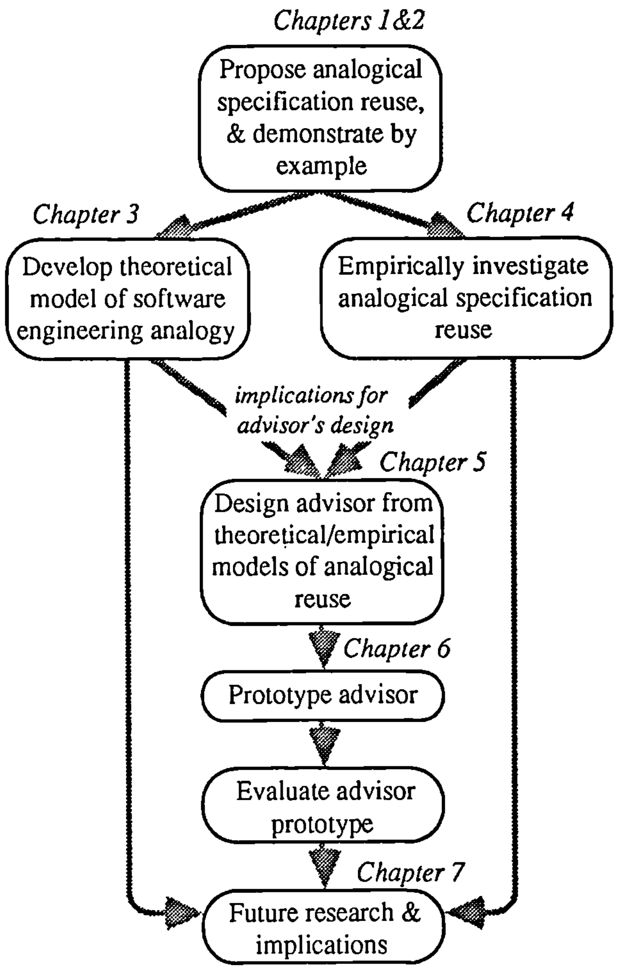

Chapters 1&2
Propose analogical specification reuse, & demonstrate by example
Chapter 3
Chapter 4
Develop theoretical model of software engineering analogy
Empirically investigate analogical specification reuse
implications for advisor's design
Chapter 5
Design advisor from theoretical/empirical models of analogical reuse
Chapter 6
Prototype advisor
Evaluate advisor prototype
Chapter 7
Future research & implications

Figure 1.7 - overview of the rationale behind the author's research

Chapter 3 presents a theoretical framework of software engineering analogies for retrieving and explaining specifications. Example analogies are used to evaluate the meta-schema for representing key domain facts and the logical model of domain abstraction derived from existing cognitive and computational models of analogical reasoning.

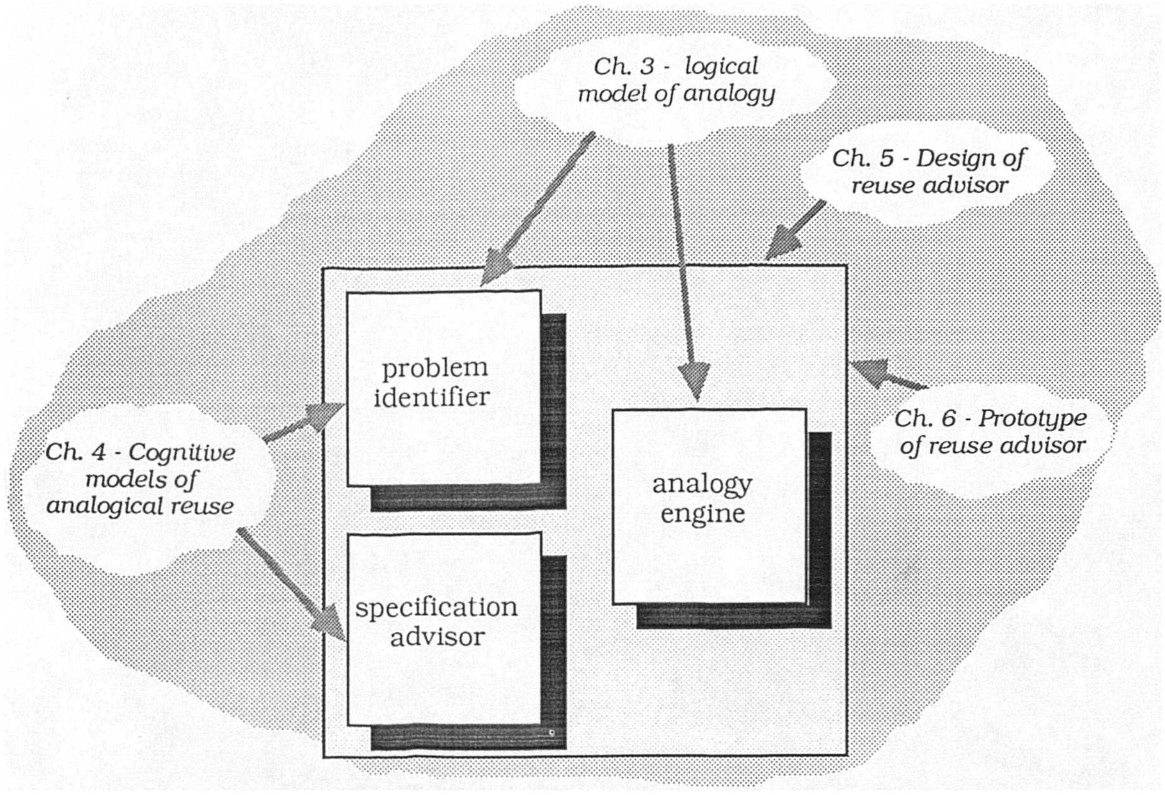
Chapter 4 represents four empirical studies of software engineering behaviour which show analogy to be an effective paradigm for specification reuse and inform the advisor's design. A first study describes a controlled experiment to show the effectiveness of reusing abstract and concrete analogical specifications on the analytic performance of inexperienced software engineers. A second study reports software engineers' reasoning strategies to inform design of the problem identifier (see Figure 1.6) and demonstrate problems which can be overcome by analogical reuse. The third and fourth studies elicit cognitive task, reasoning and mental models of analogical specification reuse. The third study identifies reuse errors to inform the specification advisor's diagnostic and error-detection module. The fourth study identifies experts' successful analogical comprehension and transfer strategies which can be followed by inexperienced software engineers.

Chapter 5 draws on the logical model of software engineering analogies from chapter 3 and the empirically-derived models of analogical reuse practice from chapter 4 to design the advisor's three components:

- the problem identifier acquires key facts about a domain prior to matching reusable specifications. This acquisition process will be informed by the logical model of software engineering analogies developed in chapter 3 and results from the first empirical study in chapter 4;
- design of the specification advisor will be informed by empirical findings from other studies in chapter 4. A diagnostic capability identifies likely misconceptions about analogies based on errors exhibited by inexperienced software engineers in study 3. The advisor will also encourage inexperienced software engineers to adopt effective comprehension and transfer strategies exhibited by successful expert reusers in study 4;
- the analogy engine matches and retrieves analogical specifications using a computational implementation of the logical model of software engineering analogies defined in chapter 3.

Chapter 6 describes a partial implementation of an intelligent reuse advisor and its evaluation in user trials with inexperienced software engineers. Both the problem identifier and analogy engine were implemented and evaluated, thus allowing software engineers to enter key domain facts which could be analogically matched for goodness of fit to candidate specifications. This evaluation indicated the effectiveness of the cooperative paradigm for analogical specification reuse.

Chapter 7 summarises this research and proposes future directions. Implications are discussed for tools which support cooperation between a tool and a single user. In particular, studies to extend and validate the proposed framework of software engineering analogies are outlined and justified. This research is expected to inform future versions of advisors for intelligent reuse and requirements engineering. The contribution of each chapter to the development of the advisor is shown in Figure 1.8.


Ch. 3 - logical model of analogy
Ch. 5 - Design of reuse advisor
Ch. 4 - Cognitive models of analogical reuse
Ch. 6 - Prototype of reuse advisor
problem identifier
specification advisor
analogy engine
Figure 1.8 - focus of research effort by chapter

1.7 Contributions

This interdisciplinary doctoral research feeds off and has implications for research in software engineering, artificial intelligence and human-computer interaction, similar to implications put forward by Basili & Musa (1991):

Software Engineering:

- example-based analysis of specification reuse indicates that analogies are more widespread than existing software reuse research has suggested;
- a framework of software engineering analogies identifies key analogical determinants

for retrieval and explanation of specifications;
- a tentative model of domain abstraction for software engineering is proposed. Current reuse of domain cliches (e.g. Reubenstein 1990, Reubenstein & Waters 1991) and object-oriented approaches (e.g. Coad &Yourdon 1990) are not founded on any theory of abstraction, so knowing *what to abstract* can augment these approaches;
- a prototype version of the advisor has been implemented and evaluated to indicate the effectiveness of analogical specification reuse;
- analogical reuse is proposed and demonstrated as a paradigm for increasing software development productivity and the quality of requirements specifications.

Human-Computer Interaction/Cognitive Science:

- a cognitive task model of inexperienced software engineers' reasoning and analytic strategies is proposed, in contrast to previous studies which examined expert analytic behaviour (e.g. Guindon 1990, Vitalari & Dickson 1983). This model has implications for the design of the reuse advisor and requirements engineering tools;
- cognitive task and reasoning models of analogical specification reuse and mental models of analogical comprehension are also proposed to describe software engineers' behaviour during reuse of analogically-matched specifications. They extend our limited knowledge of how software engineers reuse;
- cognitive task models can inform more general process and activity models of analogical specification reuse;
- existing cognitive and computational theories of analogy are extended. Previous theories may have oversimplified analogical reasoning and ignored inter-individual differences and the importance of domain knowledge in analogical reasoning.

Artificial Intelligence:

- a computational implementation of the logical model of software engineering analogies has been developed, similar to several existing computational engines (e.g. Falkenhainer et al. 1989);
- the success of a cooperative approach to specification reuse is indicated by evaluation of the prototype which augments the problem solving skills of humans with artificial intelligence tools (e.g. Kolodner 1991, Woods & Roth 1988);
- the framework of software engineering analogies can be proven as a computational reasoning process.

Chapter 2

2: Specification Reuse by Analogy

This chapter examines specification reuse in terms of existing software reuse paradigms such as keyword retrieval and domain analysis to demonstrate analogy as a more effective paradigm. A more detailed example shows analogical specification reuse between real-time as well as business information applications.

2.1 The Air Traffic Control/Flexible Manufacturing System Analogy

This example describes an analogy between an air traffic control (ATC) and flexible manufacturing system (FMS). A detailed version is given in Appendix D. Although the ATC and FMS domains may appear quite different, analogy can support extensive reuse between two specifications.

2.1.1 The Air Traffic Control System

The ATC system monitors the position of commercial aircraft flying in the vicinity of an airport. Aircraft may be flying to, coming from, or flying over the airport. The aims of the system are to ensure aircraft do not risk collision by coming too close to each other, and to track aircraft to ensure they are following the agreed flight plan (Perry 1991, Agard 1973).

The sky around the airport is structured to improve the control of aircraft movements. Aircraft fly along unidirectional air corridors at different heights, as described in Figure 2.1. To ensure safety the system must alert the air traffic controller whenever two aircraft come too close. Aircraft are surrounded and protected by an air space which no other aircraft is permitted to enter. This air space is a three-dimensional area which exists within a given air corridor and height (see Figure 2.1). The system monitors each aircraft to ensure it does not deviate from either the air corridor or the flight plan. Each flight plan is divided into a number of flight steps, which are given by the air traffic controller to direct the aircraft to use given air corridors at certain times during the flight.
A level-0 data flow diagram (DFD) representing the air traffic control system is given in Figure 2.2.

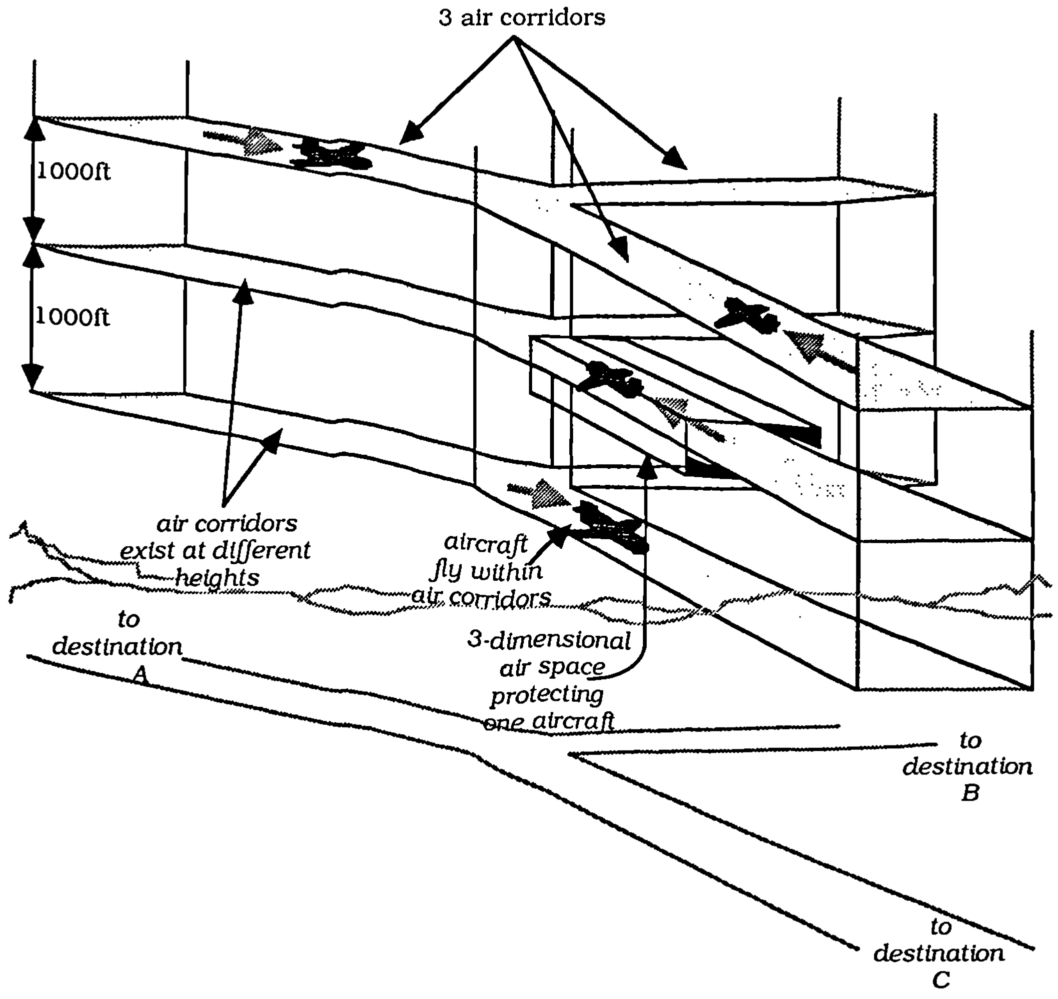

3 air corridors
1000ft
1000ft
air corridors exist at different heights
aircraft fly within air corridors
to destination A
3-dimensional air space protecting one aircraft
to destination B
to destination C

Figure 2.1 - Three-dimensional model of the airways within one geographical area: aircraft fly at different heights along air corridors

2.1.2 The Flexible Manufacturing System

A company manufactures products with the latest computerised production techniques, and keeps human intervention in the production process to a minimum. During manufacture products are passed along lines of machines by a complicated series of conveyor belts and automatic handlers. The aims of the flexible manufacturing system (FMS) are two-fold: (i) to ensure products do not collide, by warning the production controller if two products are in the same manufacturing section, and (ii) to ensure products are manufactured according to the steps in their production plan, which dictates the machine order which each product must follow. The FMS is described pictorially in Figure 2.3 and a level-0 DFD describing the system is given in Figure 2.4.

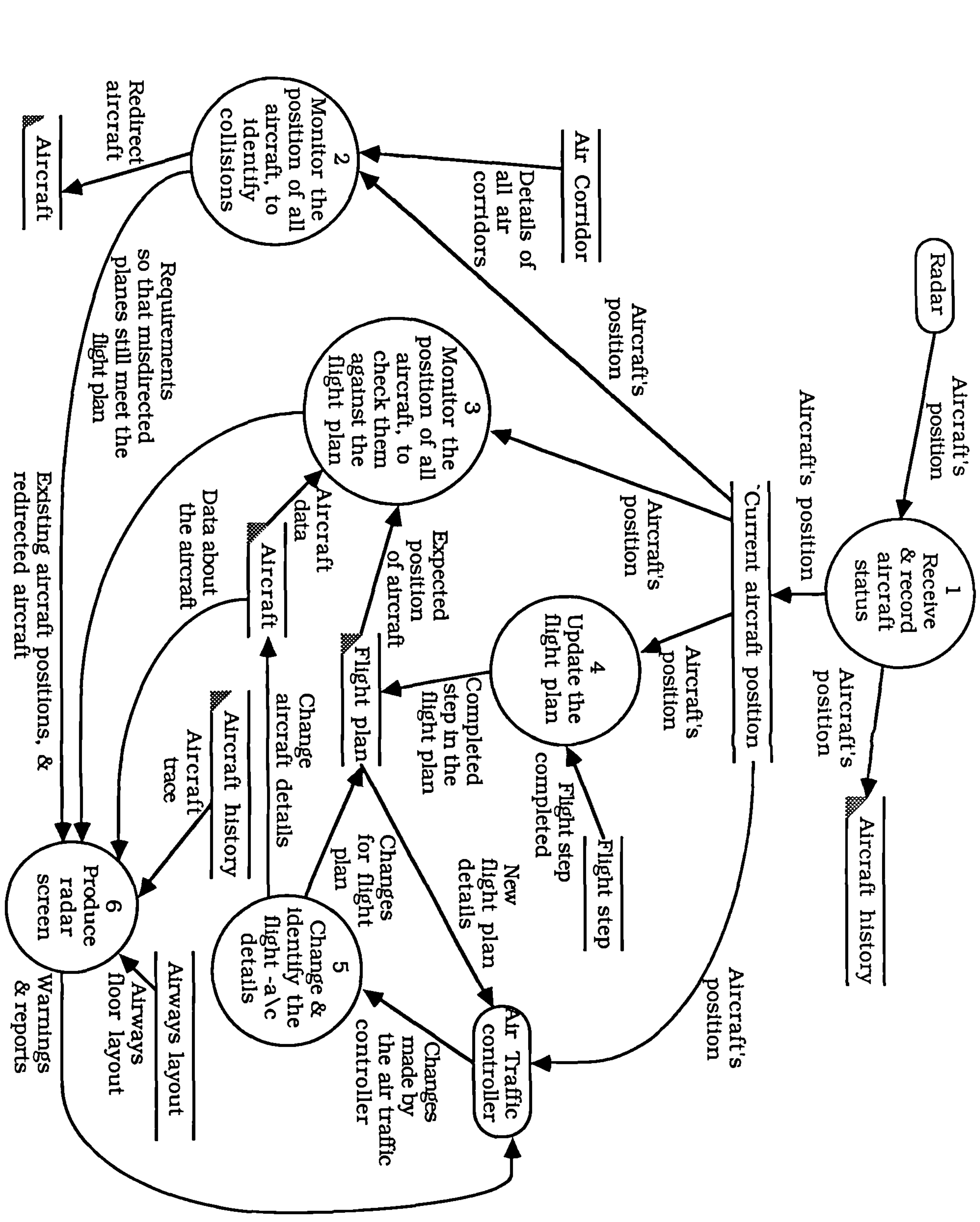


Figure 2.2
level-0 DFD for air
traffic control system

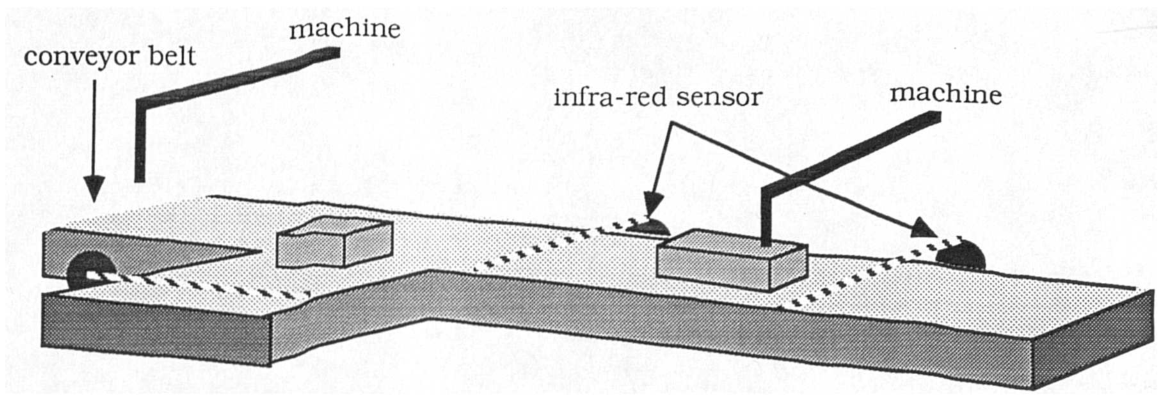


Figure 2.3 - pictorial representation of the flexible manufacturing system

2.1.3 Discussion of the ATC/FMS Analogy

Both the ATC and FMS domains have objects moving in a space, risking collision and being guided by a plan to reach a destination. These basic similarities suggest the existence of a good analogy which can support extensive reuse between the ATC and FMS specifications (see Figures 2.2 and 2.4 - reused DFD components are identified by their similar physical position in the diagrams). Reuse can occur between external entities, inputs and outputs, and data stores, as well as between processes. For example, reuse is possible between the MONITOR and REPORT processes, as well as between the two external entities RADAR *and INFRARED SENSOR, and between the AIR* CORRIDOR and MACHINE TRACK data stores.

2.2 Domain Analysis and Domain Modelling

The ATC/FMS analogy can support extensive transfer of domain knowledge. Domain analysis for application modelling is a commonly proposed alternative technique for eliciting reusable domain knowledge in well-understood domains (e.g. oil-well logging, Barstow 1985). Domain analysis attempts to make domain knowledge available for reuse (Arango & Freeman 1985) by eliciting and modelling this knowledge (Prieto-Diaz 1990) using transformational models (e.g. Balzer 1981, Wile 1983, Smith et al. 1985, Feather 1987). Typical domains include statistics reporting (Neighbors 1980, 1984). However, reusable domain knowledge may be difficult to elicit and apply during requirements analysis:

- analysing and modelling specific domains can only support multiple instances of reuse within single domains while large-scale software development often occurs in diverse,

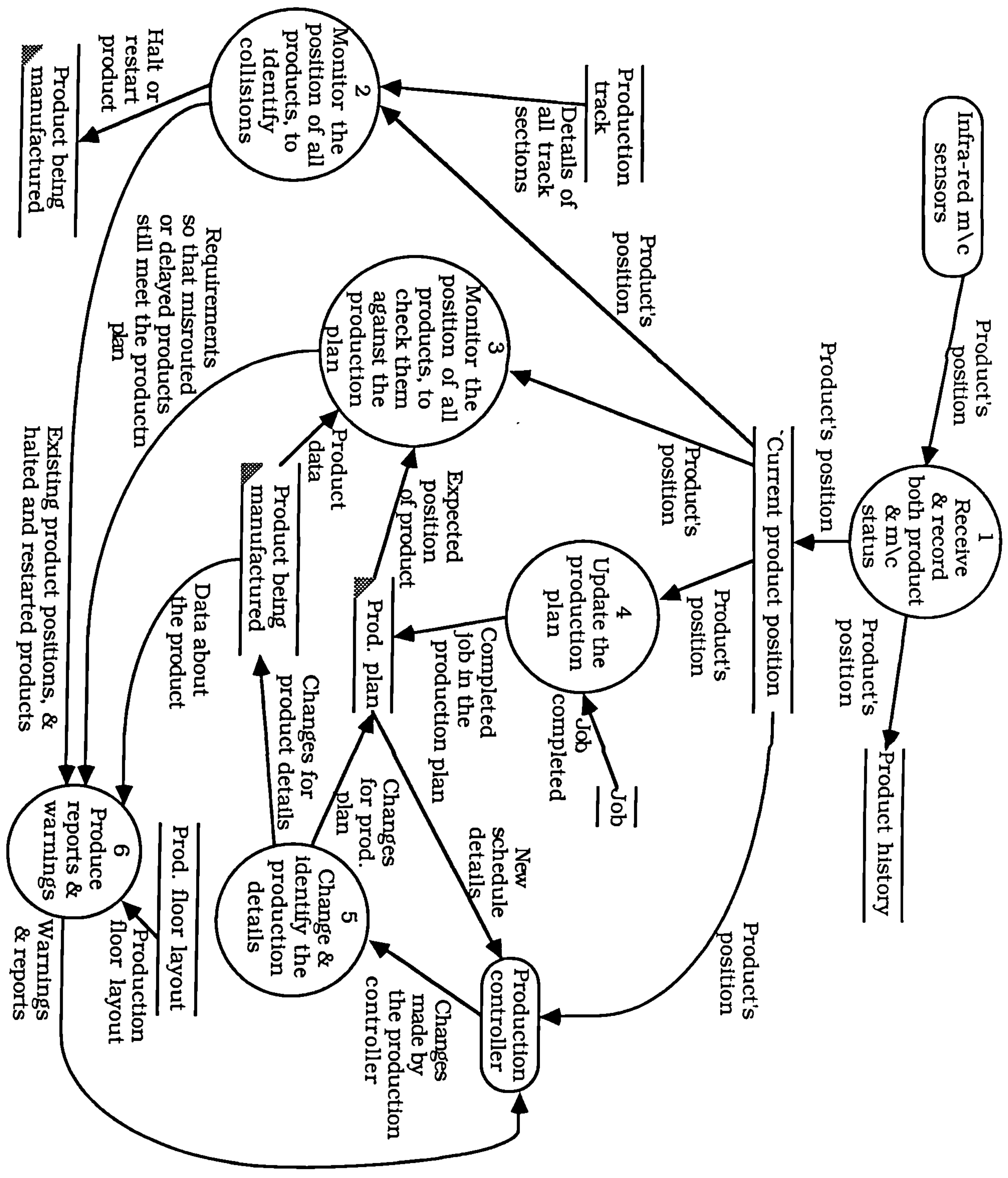

Infra-red m\c sensors
Product's position
1 Receive & record both product & m\c status
Product's position
Product history
Product's position
Current product position
Product's position
Production track
Details of all track sections
2 Monitor the position of all products, to identify collisions
Halt or restart product
Product being manufactured
Product's position
3 Monitor the position of all products, to check them against the production plan
Product's position
Product data
Product being manufactured
Requirements so that misrouted or delayed products still meet the productn plan
Expected position of product
4 Update the production plan
Product's position
Job completed
Job
Completed job in the production plan
Prod. plan
New schedule details
Production controller
Product's position
Changes for prod. plan
Changes for product details
Data about the product
5 Change & identify the production details
Changes made by the production controller
Existing product positions, & halted and restarted products
6 Produce reports & warnings
Prod. floor layout
Production floor layout
Warnings & reports

Figure 2.4
Level-0 DFD for Flexible
Manufacturing System

poorly-understood applications (Prieto-Diaz 1990);

- eliciting generic domain knowledge can support reuse across applications, for example the IDeA environment (Harandi & Lubars 1985, Lubars & Harandi 1986, 1988) provides generic design components for inter-application reuse. However, derivation of domain knowledge has proved difficult and time-consuming. Prieto-Diaz (1990) reveals that success stories with domain analysis are exceptions rather than rules, while Arango (1987) pinpoints obstacles to acquiring reusable domain knowledge as the very high cost of either (i) generating such knowledge, or, (ii) locating and repackaging the knowledge in reusable forms. Unfortunately, effective methods for eliciting and representing reusable domain knowledge still appear to be some way off (Arango 1988, Prieto-Diaz 1991).

Unlike specification reuse, domain analysis is unlikely to lead to effective knowledge reuse across domains. One possible reason for the effectiveness of inter-domain analogies is that reusable specifications transfer many facts represented as complex knowledge structures rather than isolated predicates. The next two sections examine other existing reuse paradigms for specification reuse.

2.3 Paradigms for Software Reuse

Most current paradigms support reuse of code modules rather than higher-level designs or specifications. This has been achieved through domain-independent retrieval mechanisms at the expense of component comprehension and adaptation (e.g. Maarek et al. 1991). Indeed, until recently, the black-box view of software reuse prevailed to ensure that the contents of the software component were unseen and unmodified during reuse (e.g. chapter 2 of Dusink & Hall 1991). Reuse was viewed as a process of composition, during which components were linked using modular interconnection languages (e.g. Goguen 1986). Unfortunately these approaches have failed to achieve the expected increases in software productivity and quality. In addition, management did not recognise the importance of software reuse even in cases where the technical problems were easily solved. The next section examines these problems more closely in the context of specification reuse.

2.4 Current Paradigms for Specification Reuse

Reuse of specifications during requirements analysis and the early stages of software design has been discussed elsewhere (e.g. Balzer et al. 1983, Finkelstein 1988, Czuchry

& Harris 1989, Karakostas 1989, Sommerville et al. 1989, Basili 1990, Mylopoulos & Rose 1991). Tracz (1990) suggests that successful reuse of requirements specifications can also provide the basis for the reuse of design, code and documentation if traceability between a specification and its design and code could be assured. However, few concrete conclusions have been reached about the nature or processes of specification reuse. Existing paradigms have proved moderately successful for reuse of system designs and code, however they appear inappropriate for reusing specifications during requirements analysis. Each paradigm is investigated in terms of the ATC/FMS example to assess its appropriateness for matching and explaining analogical specifications during requirements engineering.

2.4.1 Keyword Retrieval

Retrieval of reusable components based on syntactic similarities between keywords is a much-vaunted approach to software reuse (Burton et al. 1987, Wood & Sommerville 1988, Prieto-Diaz 1991). Keyword retrieval paradigms match software components by their functionality since most software performs a function which characterises the software component. Indeed, Prieto-Diaz's (1985) extensive study of software descriptions concludes: '*Program listings are characterised by describing the function performed by the program...*'. However, large-scale software engineering problems are too complex to be described comprehensively by keywords representing only the functionality of the required computer system. For instance, keywords representing critical functions of the ATC and FMS computer systems include *MONITOR, UPDATE, REPORT & RECEIVE INPUT*. However, these processes are also common in many systems which are not analogous and do not support valid specification reuse. For example, a patient monitoring system also has these functions, and any attempt to reuse the specification of patient monitoring systems during analysis of the FMS would be more likely to hinder than help the software engineer.

Wood & Sommerville (1988) propose a lexicon of semantically-equivalent descriptors of code functions based on natural language descriptions of software components. Lexicons provide richer descriptions but still only match software components by their function. Alternative, more complex specification descriptors which focus on non-functional features of software components appear necessary for effective software reuse. Chief among these approaches is faceted classification schemes.

2.4.2 Faceted Classification of Software Components

Faceted classification schemes derived from library science (e.g. Prieto-Diaz 1991, Prieto-Diaz & Freeman 1987, Prieto-Diaz 1985) attempt to overcome several problems of simple keyword retrieval by describing non-functional features of software components. Prieto-Diaz's facets included objects, medium, system type and setting, which identifies the original application of the software component. However, faceted classification schemes are unlikely to support specification reuse across domains because they fail to provide the general and powerful descriptors needed for cross-domain reuse:

- faceted classification schemes are founded on descriptive, application-dependent facets. For example, the ATC specification may have many settings, including *aircraft management, passenger transport, safety-critical* and *real-time*, while settings describing the FMS specification include *production management, manufacturing* and *real-time*. The only match between the ATC and FMS specifications is *real-time*, which is clearly insufficient as a basis for analogically matching specifications. Wood & Sommerville (1988, p 206) also reported that differences in terminology inhibit the applicability of classification schemes for cross-domain reuse. Indeed, Prieto-Diaz (1991) admits that faceted classification schemes are more effective for domain-specific reuse (p 94), stating that: 'A faceted scheme for a diversified collection [of software components] becomes too general, losing its descriptive precision';
- faceted classification schemes are also plagued by knowledge acquisition problems. Development of a comprehensive scheme requires difficult and time-consuming domain analysis of applications across which reuse is intended to occur (e.g. Boldyreff 1989). Furthermore, Prieto-Diaz's recent experiences (1991) suggest that deriving classification schemes and adding new entries to a component library are labour-intensive activities which cannot be fully automated. Implications for a classification scheme to support specification reuse across domains are clear: the benefits derived from specification reuse may be offset by the effort needed to initially develop and maintain such a scheme;
- finally faceted classification schemes impose discipline on their use since new problems must be described with a constrained lexicon of unnatural terms which has been shown empirically to lead to inconsistent object descriptions (Furnas et al. 1987).

In short, successful specification reuse requires both powerful and generic descriptors to identify analogical matches across domains, however, these descriptors cannot be provided by single or complex sets of keywords.

2.4.3 Abstract Templates

Reuse of abstract solutions is another, much-vaunted approach to software reuse. Reuse of abstract software components has been typified by code-level reuse of program templates (e.g. Mittermeir & Oppitz 1987, Katz et al. 1987, Volpano & Kieburtz 1989), although object-oriented programming can also promote reuse of abstract software components (e.g. Curry & Ayers 1984, Kaiser & Garlan 1987, Lenz et al. 1987, Bott & Wallis 1988). As well as providing reusable solutions, abstractions can be used as a basis for common understanding. The Programmer's Apprentice integrated reusable program components representing well-understood programming concepts (e.g. a *device driver*) into an intelligent assistant for expert programmers (Waters 1985, Rich & Waters 1988).

Reuse of abstract solutions during system specification and design has also been encouraged through provision of design templates (e.g. Harandi & Young 1985, Fugini et al. 1991) and generic specifications (Reubenstein & Waters 1991). The Requirements Apprentice (Reubenstein & Waters 1989, Reubenstein 1990, Reubenstein & Waters 1991) provides software engineers with template specifications (cliches) of the type of system under analysis (e.g. *library* or *object monitoring systems*). These cliches are combined with techniques including dependency-directed reasoning and hybrid knowledge representations to check requirements specifications for consistency and completeness and develop a summary document of the specification to facilitate communication. However, reuse of abstract specifications faces the following problems:

- successful reuse of abstract specifications must provide a complete set of specifications to fit all possible solution scenarios. Given the variety and complexity of requirements specifications, this may be a difficult task. Previous partial attempts to classify system types have proved difficult (e.g. Amadeus 1986), suggesting that reuse of abstract specifications falls foul of the *coverage* problem;
- experiences with developing abstract specifications revealed that constructing sufficiently abstract and detailed templates is difficult, i.e. reuse of abstract solutions must overcome the *granularity* problem to provide both generic and useful solutions;
- a theory of abstraction in software engineering from which to develop domain cliches is currently lacking, i.e. there is a lack of a model for systematic abstraction.

To conclude, predetermined derivation of a complete set of generic and powerful specifications representing reusable solutions may be prohibitively difficult, given the current limits of our knowledge of software engineering problems. On the other hand, reusable specifications are widely-available and may be abstracted as appropriate during reuse, thus going some way to overcome both the coverage and granularity problems.

2.4.4 Design Recovery and Replays

Capturing design decisions behind software components can assist their recovery and reuse (e.g. Dhar & Jarke 1988, Biggerstaff 1989, Fischer et al. 1989, Arango et al. 1991, Fischer et al. 1991c). However, Mostow (1989) asserts that design replays are both difficult to capture and apply when developing new systems because of the incompleteness and imperfection of rationale. Indeed, Parnas & Clements (1986) suggest that software development will never be achieved in a rationale way, because they include initially unknown system requirements, the inevitability of errors during development, preconceived ideas, economic and other influences on the software development process. Indeed, requirements analysis is more complex than other phases of software development since it involves users and software engineers during the identification, exploration and negotiation of hypotheses about diverse topics including functional requirements, implementation tradeoffs and the social impact of the new system. In addition, empirical studies of analytic processes of software engineers reveal a rich diversity of strategies typified by opportunistic reasoning (Guindon 1990). If this were not bad enough, anecdotes suggest that software engineers are notoriously bad documenters, so our ability to faithfully capture, record and reuse requirement specifications based on development rationale is poor. An alternative solution may be to infer the design rationale behind reusable specifications from documentation and the specification itself, although this leads to a new set of problems to overcome. The major advantage of this approach is that it encourages reuse from the wealth of specifications currently held in software development tools, for which no development rationale is likely to exist.

2.4.5 Formal Methods

Formal approaches to software reuse include Paris (Katz et al. 1987) and 'B' (Lafontaine et al. 1991). Formally specifying software requirements may be effective for reuse of small, well-defined software components, however, formalising the needs of a complex system is more difficult. For example, the Paris system (Katz et al. 1987) uses preconditions and postconditions plus assertions about component properties to form clauses to be proved with the Boyer-Moore theorem prover. The resulting proof developed a list of candidate components. However, determining preconditions, postconditions and assertions about reusable specifications is difficult and may not represent critical features of the specification. As such, formal methods for reuse are only likely to be effective if a perfect model of the target domain is already known. They do not encourage reuse of existing software available through CASE technology, nor do they appear to overcome the component understanding problem.

2.4.6 Management Issues

Many technical problems involving design and code-level software reuse have been overcome using the techniques described in this chapter. The currently accepted story is that the lack of wide-spread software reuse may at least in part be attributed to management practices (e.g. Prieto-Diaz 1991). For instance, Biggerstaff (1987) writes:

> *'Technologists are confounded because reusability is a multiorganisation problem and requires a critical mass of components before it can really pay off. These issues prevent spontaneous use of reuse'.*

This is due in part to management's lack of belief in benefits attainable from software reuse, which is not surprising given the intangible nature of software and current management practices which *pay* developers to write new code. However, examination of Japanese software engineering practices reveal the potential benefits for software reuse from management backing. Although management issues concerned with successful specification reuse are beyond the scope of this thesis, analogical specification reuse can demonstrate obvious benefits to an organisation, thus encouraging more positive management attitudes towards reuse.

2.4.7 Summary of Existing Software Reuse Paradigms

Existing software reuse paradigms appear inappropriate for specification reuse during requirements analysis. Simple or faceted classifications of reusable specifications are unlikely to identify domain-independent determinants of reuse while knowledge-intensive paradigms based on domain analysis and design replay have failed to overcome the knowledge acquisition bottleneck. Analogy provides an alternative, intuitively-appealling paradigm for reusing specifications across domains, however it has received little attention in the literature. Analogy can overcome the knowledge acquisition bottleneck since specifications are readily-available as a result of the software development process, thus supporting reuse of existing specifications. Analogy can also provide a rich diversity of reusable specifications from repositories, thus partially overcoming the coverage problem inhibiting reuse of generic templates.

Whilst it is not overtaxing to develop an intuitive understanding of the ATC/FMS analogy, it is more difficult to state why the analogy exists or to justify the analogy as a basis for specification reuse. A model of analogy is necessary to identify critical determinants of specification reuse, in particular to assist retrieval and explanation of a specification to aid its understanding and customisation when fitting it to the target domain. Finkelstein (1988) proposed several determinants of analogy in specification reuse, however these determinants were only supported with simple examples and not explained by any underlying model of software engineering domains, so the scope and nature of specification reuse by analogy remains undetermined. Existing general theories and models of analogy are discussed in the next section as a basis for a framework of software engineering analogies.

2.5 Analogy as a Paradigm for Specification Reuse

Analogical reasoning has been researched by both cognitive psychologists and artificial intelligence researchers. Whilst many different definitions of analogy exist, the definition proposed by Carbonell (1985, p3) suggests that specification reuse is an instance of analogical reasoning:

"Analogical problem solving consists of transferring knowledge from past problem solving episodes to new problems that share significant aspects with corresponding past experience -- and using the transferred knowledge to construct solutions to the target problems"

Analogical reasoning in specification reuse is akin to case-based reasoning (e.g. Rissland & Skalak 1991, Ashley 1991, Branting 1991, Kolodner 1991, Ashley & Rissland 1988), during which a reasoner remembers previous situations similar to the current one and uses them to solve the new problem. Case-based reasoning may be applied to adapting old solutions to meet new demands, explaining new situations, critiquing new solutions or reasoning from precedents, roles which analogical specification reuse can also fulfil. Therefore, specification reuse is also an instance of case-based reasoning. Gentner (1983) emphasises the importance of structure in analogical reasoning when she stated that:

"Analogous reasoning transfers a complete network of knowledge rather than unrelated facts"

Unlike existing knowledge-based approaches to intelligent analytic support (e.g. Harandi & Lubars 1985, Puncello et al. 1988) which tend to apply isolated domain facts or method heuristics to a target application, specification reuse involves the transfer of a network of domain and method knowledge represented as a specification. Gentner demonstrates that mapping an interrelated network of knowledge allows analogical reasoning to occur between domains which are otherwise very different, for example an analogy occurs between the structure of a hydrogen atom and our solar system. This would suggest that analogy is not based on syntactic similarities between problems, but is critically determined by deeper knowledge structures (Gentner 1983), thus suggesting that analogy may be particularly effective during cross-application reuse.

Russell (1989) discussed theories of analogy and identified three types, which he labelled as similarity-based theories, theories based on implicative justification and determination theories. Similarity-based theory only exploits syntactic matches between domains so it is similar to existing reuse paradigms. On the other hand, theories of determination and implicative justification require domain knowledge. The difference between the two is that analogy by implicative justification requires the unrealistic assumption of complete knowledge about software engineering domains. Determination analogies differ from implicative justification in their use of less complete, *weak* domain theories to suggest key features of an analogy. Such a theory only models critical analogical features at the expense of other domain knowledge. Determination theories have been implemented in several artificially-intelligent systems, for instance HYPO (Ashley & Rissland 1988) reasons analogically between legal cases, using dimensions of legal cases to identify useful axes along which cases can vary.

Determination theories identifying key features of analogical specifications may be one approach for effective reuse. They exploit a weak, application-independent model of software engineering domains in which key analogical determinants are represented as complex knowledge structures rather than isolated facts. This thesis investigates a weak determination theory of software engineering analogies. As a starting point, existing models of analogy were examined more closely for key knowledge types in analogical specification reuse.

2.5.1 Previous Research of Analogical Reasoning

Theories and models of analogical reasoning can be divided into four classes: natural language metaphors, cognitive models of analogical problem solving, learning through analogy and computational analogical models.

2.5.1.1 Natural Language Metaphors

Natural language metaphors have often been likened to simple similes such as 'my job is a jail', so their relevance to analogical specification reuse may be limited. Glucksberg &

Keysar (1990) argue that metaphors are class-inclusion assertions, in which the topic of the metaphor is assigned to a diagnostic category. For example '*my job is a jail*' suggests that the topic 'job' is *'an entity that confines one against one's will, is unpleasant and is difficult to escape from*' (p3). This understanding of metaphorical comparisons suggests that analogies may occur between two instances of the same problem type, so abstraction may be important for analogical reasoning.

2.5.1.2 Analogical Problem Solving

Empirical studies of analogical problem solving also revealed the importance of abstract memory schema for analogical retrieval (e.g. Gick & Holyoak 1983, Cheng & Holyoak 1985, Cheng et al. 1986, Keane 1987). Generally, findings suggest that analogical problem solving and training is effectively enhanced by teaching or presenting abstract principles common to the source and target analogs. On the other hand, subjects without relevant schemata exploit superficial similarities and transfer solutions incorrectly (Ross 1987, 1989), leading to mental laziness and solution copying rather than analogical understanding (Novick 1988).

2.5.1.3 Analogical Learning

Empirical studies of analogical learning in unfamiliar domains indicate that analogies can be used to teach complex concepts (e.g. Caplan & Schooler 1990). Analogical learning of programming constructs (e.g. 'variable is like a box', Burstein 1988a, 1988b, Hoc & Nguyen-Xuan 1990) has proved effective. However, du Boulay (1989) identified errors due to the misapplication of analogy, indicating that novice programmers tried to extract more from an analogy than was warranted. Halasz & Moran (1982) also point out that extending an analogy too far may become a barrier to learning. One solution may be to constrain the scope of an analogy by abstraction, so analogical learning should be restricted to analogical inferences also belonging to the underlying abstraction.

2.5.1.4 Derivational Analogy

Derivational analogy solves problems by replaying the plan used to solve a previous problem, modifying it where necessary (Mostow 1989, Carbonell 1985, 1988). Mostow (1989) reviewed existing computational models of derivational analogy to analyse how they redesign complex artifacts like programs and circuits. Their deficiencies indicate that they are insensitive to higher-level aspects of redesign problems and lack a retrieval method that scales up to larger design libraries such as a specification repository. In addition, these systems have focused on redesign in relatively well-defined domains such as electronic circuitry. The difficulties of capturing and replaying reasoning processes during requirements analysis are likely to be much greater, due in part to the lack of domain knowledge which is needed to interpret such rationale. Derivational analogy is similar in nature to design replays, and unfortunately it suffers from the same set of problems when applied to specification reuse.

2.5.1.5 Computational Approaches to Analogy

Computational models of the above types of analogical reasoning have been developed both as implementations of cognitive theories of analogy and as problem solvers in their own right (Thagard 1988, Hall 1989). Typically these models represent domain-specific deductive reasoners which emphasise the role of domain knowledge (e.g. Kedar-Cabelli 1988a, 1988b, Holyoak & Thagard 1989) and abstractions (e.g. Greiner 1988a, 1988b) in analogy. They provide important clues for knowledge and structures which are analogically-matched and transferred.

2.5.1.6 Summary of Previous Research of Analogical Reasoning

Both computational and cognitive models of analogical reasoning emphasise the importance of abstraction, although the knowledge structures which are represented in these abstractions remain largely undetermined. Computational models of analogy emphasise the importance of knowledge types and structure in analogy, so the ATC\FMS example was investigated more closely.

2.5.2 Knowledge in Software Engineering Analogies

At their broadest level, software engineering analogies may match at least three different types of knowledge which were examined for their role in determining the ATC/FMS analogy:

- solution knowledge, representing information system concepts of the reusable specification;
- domain knowledge, representing problem domain and real-world knowledge;
- goal knowledge, describing purposes of new and reusable systems.

2.5.2.1 Solution Knowledge

Solution knowledge differs from domain knowledge in that it describes information systems using structure charts and other notations. Solution knowledge is defined using notation syntax in the same way that program code is represented using language syntax rather than underlying domain constructs. Analogical matching on the syntactic notation of structured diagrams permits structural matching (e.g. Gentner 1983), however this syntax provides few clues about the semantics of the domain. As a result the syntax of structured notations may support analogical transfer but fail to justify analogical matching and retrieval of specifications.

2.5.2.2 Goal Knowledge

Several researchers (e.g. Kedar-Cabelli 1988a, 1988b) have suggested the importance of analogical mappings between goals or purpose. For example, two analogical goals of the FMS and the ATC systems include *the ATC system should monitor aircraft to meet the flight plan*, and *the FMS should monitor products to meet the production plan.* The need for analogical system goals is not surprising, since there is a close relation between a system's goals and its functionality. Indeed, goals can be said to be high-level statements of functionality. However, system functionality is inappropriate as a basis for analogical matching for two reasons. First, functional goals are difficult to define precisely, for example the goal *monitor products to meet the production plan* may also describe the purpose of a non-analogous system for scheduling production plans. Second, goals do not represent the interrelated knowledge structures fundamental to analogical recognition and transfer (Gentner 1983), hence they fail to capture key analogical concepts. Simple measures of similarity such as functional goals may be suitable for reuse in the small, however they negate the importance of domain knowledge which is essential in requirements engineering. This distinction emphasises the difference between the domain and design spaces, so models of software engineering analogies must consider domain knowledge.

2.5.2.3 Domain Knowledge

Many computational models reason analogically with domain knowledge (e.g. Hall 1989), albeit using constraints on the categories of domain knowledge which can be mapped. This thesis hypothesises that analogical specification reuse is critically determined by domain knowledge. For example the ATC/FMS analogy is understood in terms of aircraft travelling unidirectionally along air corridors while products move along conveyor belts. However, domain knowledge can represent a wide diversity of concepts.

Both Winston (1980, 1982) and Gentner (1983) hypothesised that analogy transfers a network of relations that hold between concrete objects in two domains, while attributes describing objects are discarded, see Figure 2.5. Applying this structure-mapping approach to software engineering analogies is appropriate because domains tend to be well-structured, with well-defined objects and relations. Furthermore, Gentner constrained analogical matches to object-relations which supported a higher-order causal structure. However, the ATC/FMS example revealed that causality within domain structures alone was too general to constrain analogical matching, for instance Figure 2.6 describes causally linked object-relations which correctly describe the domain but do not support the analogy. Therefore, further constraints on analogical matching between software engineering domains are required as a basis for a framework of software engineering analogies. However, as Russell (1989) indicates, these constraints must be founded on a weak theory of the domain of interest. In this case, the domain of interest is software engineering, and analogical matching and explanation requires a model of domain knowledge in software engineering.

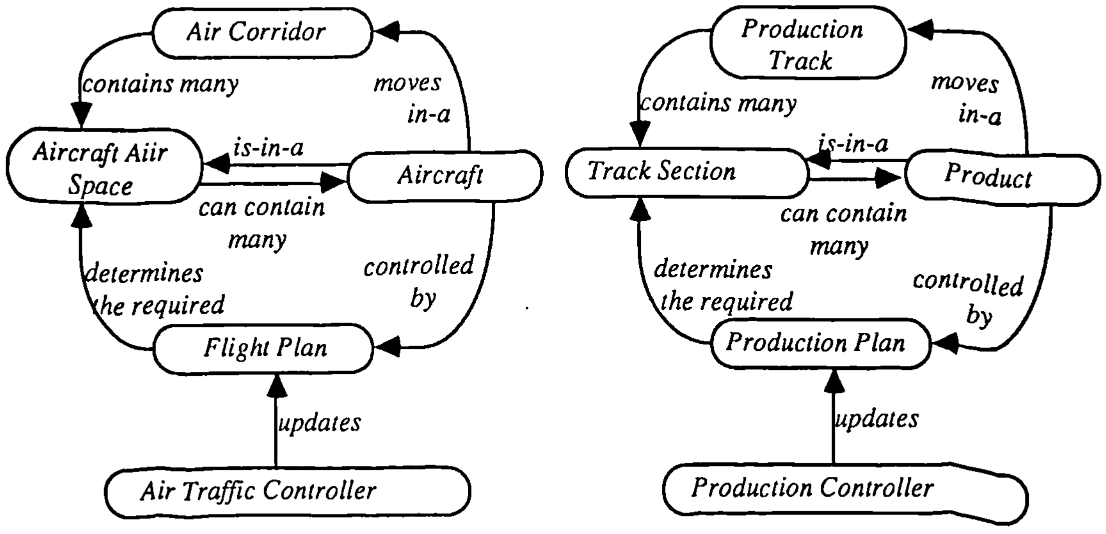


Figure 2.5 - corresponding causal relations representing the ATC and FMS domains

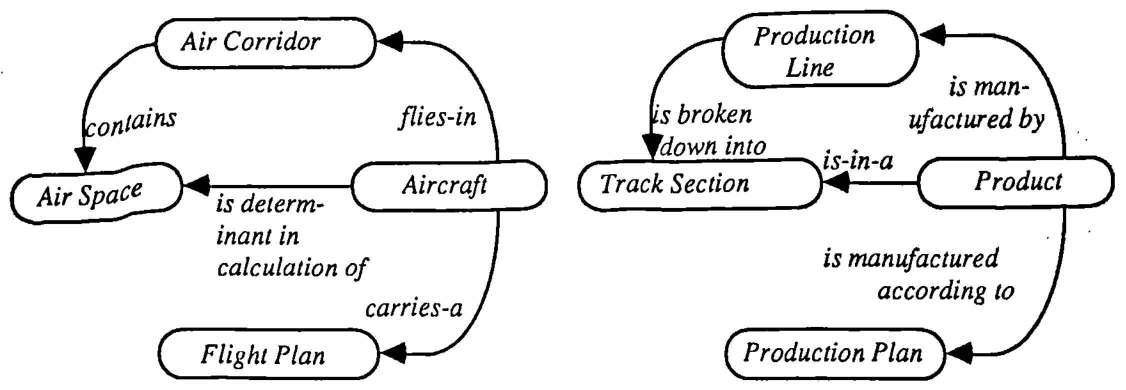


Figure 2.6 - causal relations which fail to support analogical mapping between the two domains

2.5.2.4 Summary of Knowledge in Software Engineering Analogies

Recognition and explanation of software engineering analogies requires solution knowledge, goal knowledge and domain knowledge. The ATC/FMS example suggests that specification reuse exploits solution and goal knowledge, however existing models of analogy indicate that domain knowledge is critical. A weak theory of software engineering domains is needed to determine software engineering analogies. Object relations which can be analogically-mapped and causally linked may be one approach to successful analogical mapping, although further constraints appear necessary. This framework is developed and evaluated in the next chapter.

2.6 Conclusions

Specification reuse has two advantages over alternative paradigms for knowledge and software reuse during requirements analysis. First, domain knowledge is difficult to elicit while reusable specifications store domain knowledge in a readily-applicable form. Second, the potential benefits from reusing domain rules or transformations have tended to be small and within single applications while specification reuse can support large-scale knowledge transfer across different domains. These advantages are due to the structure and assumptions stored implicitly in reusable specifications but missing from rule- and transformation-based paradigms.

Example-based studies indicate that analogy is an alternative paradigm for reusing specifications because they provide old solutions to new problems represented as complex knowledge structures (Carbonell 1985, Gentner 1983). Such analogies are critically determined by key similarities between the underlying domains rather than their reusable specifications. The next chapter expands this definition of analogy into a framework of software engineering analogies as a basis for their retrieval and explanation.

Chapter 3

3: A Definition of Analogy in Software Engineering

This chapter presents a logical model of analogy between software engineering domains to determine why analogies occur and how they can be identified during specification reuse. It has two components: a meta-schema of knowledge types for representing key facts about software engineering domains, and a model of domain abstraction which determines the scale of analogical matching during specification reuse. This model of analogy represents a logical definition rather than the more psychologically-plausible representations of analogical reasoning discussed in chapter 4.

The first component of the model, the meta-schema of knowledge types, defines key domain facts about software engineering problems as a basis for analogical retrieval and explanation. These domain facts were identified during example-based analyses and using constraints on analogical matching applied from existing cognitive and computational models of analogical reasoning (e.g. Gentner 1983, Greiner 1988a, Holyoak & Thagard 1989). The second component is a logical model of domain abstraction for software engineering. Domain abstractions are central to analogical matching, hence a model is proposed to identify key similarities and differences between abstractions to assist analogical retrieval and explanation. The meta-schema and logical model of domain abstraction provide the analogical expertise for specification retrieval and explanation.

The proposed logical model of software engineering analogies is defined in several stages throughout the chapter. First, the meta-schema of knowledge types is described as a basis for representing software engineering domains and analogical similarities between them. Knowledge types in the meta-schema are justified using constraints on analogical mapping derived from existing cognitive and computational models of analogy. The meta-schema is then demonstrated and evaluated using several detailed analogical examples described more fully in Appendix A. The remainder of the chapter defines the proposed model of domain abstraction to provide a framework for identifying and justifying analogical matches between domains. Domain terms for instantiating the meta-schema are derived from a subset of software engineering domains which were investigated. Finally the proposed definition of analogy in software engineering is demonstrated further using several, non-simple examples of analogical domains.

3.1 An Initial Definition of the Model

Chapter 2 identified three basic knowledge types which are mapped between analogical software engineering problems. Each software engineering problem incorporates three models: a *solution* model, a *goal* model and a *domain* model:

- the solution model defines a specified information system;
- the goal model depicts functional requirements to be fulfilled by the new system;
- the domain model represents all aspects of the underlying problem. Chapter 2 revealed that analogical similarity between domains can occur between interconnected domain structures (Gentner 1983), hence domain structure may be important when defining analogies.

To summarise, although knowledge about information systems and their functional goals is important for analogical reuse, it is knowledge describing their underlying domains which determines the existence of an analogy. Therefore, this chapter primarily investigates analogical mapping between software engineering domains. However, before this can happen, the knowledge which is mapped between software engineering domains must be defined formally. A knowledge meta-schema is proposed for representing software engineering domains and analogical mappings between them. Its definition is followed by justification of the meta-schema using constraints imposed by existing cognitive and computational models of analogical reasoning.

3.2 A Meta-schema of Knowledge Types for Software Engineering Domains

The proposed framework of software engineering analogies may be described as a hybrid model of analogy (Russell 1989) because it integrates several existing cognitive and computational models of analogy to form a framework of software engineering analogies. This model claims that an analogical specification is critically determined if:

- two domains share a network of interrelated, semantically-matched terms;
- these domains are described with similar state transitions between object structures;
- these source and target descriptions are both instantiations of the same domain abstraction representing a generic class of software engineering domain, see

Figure 3.1.

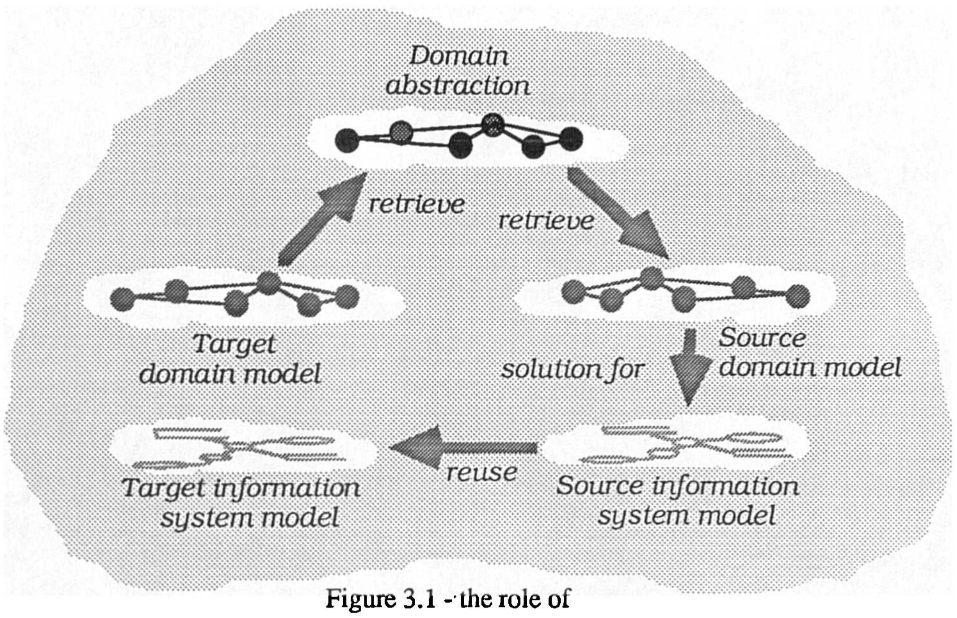


Figure 3.1 - the role of domain abstraction in specification retrieval

Software engineering domains are described using a meta-schema consisting of the seven knowledge types shown in Figure 3.2. Each knowledge type is specified using typed predicates. The first five knowledge types describe the problem domain while the other two define aspects of the information system linked to that domain. This thesis hypothesises that analogy is critically determined by key state transitions between object structures, so the knowledge meta-schema describes each software engineering domain in terms of these state transitions. In the next section each knowledge type in the meta-schema is described and elaborated using several simple software engineering examples.

object structure:	**< object, object, structural-relation >**
domain requirement:	**< object, object, structural-relation, value >**
state transition:	**< object, source, destination, transition >**
object type:	**< object, object-type >**
conditions on state transition:	**< precondition, object, source, destination, transition>**
function/domain event:	**< function/event, object, source, destination, transition >**
function achieving transition:	**< function >**

Figure 3.2 - the knowledge meta-schema definition using formal typed predicates

3.2.1 State Transitions between Object Structures

Central to this model are state transitions with respect to a domain structure. They represent system intervention in the domain to maintain or change states which are defined as object structures. Gentner (1983) constrained analogical matching by imposing a higher order causal structure on such matches. Similarly, state transitions are linked causally through specification to system functionality. One example of a state transition is an allocation in the theatre domain which causes an object, such as a theatregoer's booking, to change state from an unoccupied state to an occupying-resource state by being allocated to a theatre seat. This can be represented using the meta-schema as:

< booking, reservation, seat, many >

The final term of the predicate indicates that *many* seats may be changed by one booking transition from an unoccupied to occupying-resource state. A second example is the *reserve* action in the university course administration domain which causes an object, such as a student application for a course, to change state from an unreserved to a reserved state by guaranteeing a place on that course:

< application, pending, place, many >

State transitions are central to the model of software engineering domains, and all other knowledge types defined in the meta-schema elaborate this basic definition of state transitions.

3.2.2 Object Structures

Domain states are defined with reference to object structures representing object-relationship predicates similar to the notations used by Gentner (1983) and Sowa (1984). For example, object structures representing the theatre domain include the theatre having many seats, the waiting list containing a number of unmet bookings and the world having one theatre. Structures define object membership in sets which model high-level components of the domain. Cardinality constraints on set membership are expressed as properties of object structures using the meta-schema, for example:

< theatre, seat, has-many >
< waiting-list, unmet-bookings, contains-many >
< world, theatre, has-one >

3.2.3 Domain Requirements

Domain requirements elaborate object structures through addition of language statements identifying goals and constraints for the required system. Future research will unpack the nature of these requirements further, however for the purposes of this thesis, domain requirements were represented in simple form as high-level linguistic statements of needs, functions and constraints. For instance, needs can be represented as high-level functions which are a long way from the operationalised state. Within the meta-schema domain requirements are associated with object structures and convey additional facts about system goals. For instance, a requirement in the theatre reservation domain is *theatre seat contains-one booking*, but this description fails to convey the complete requirement. The system aims to maximise the use of theatre seats for any performance, so the domain requirement is specified through addition of the linguistically-expressed goal, so that:

< theatre seat, booking, contains-one, maximise-use >

Domain terms which express functional requirements in this chapter include maximise-quantity, minimise-quantity, same-properties and date or time-limits.

3.2.4 Preconditions on State Transitions

Preconditions on state transitions further elaborate the definition of these key transitions. The proposed model hypothesises that state transitions between domain states can be triggered by different conditions. For instance, stock replenishment in a stock control domain occurs when stock levels reach a minimum. On the other hand, the allocation state transition in the theatre domain only occurs if the reservation and allocated seat have similar constraints such as non-smoking, price <£20, seat is unreserved, etc.. These two instances of condition can be represented using the meta-schema notation as:

< minimum-level, stock, supplier, warehouse, many >
< same-properties, booking, reservation, seat, many >

3.2.5 Object Types

Constrained typing of objects is also permitted by the meta-schema, for instance stock items in a stock control domain and theatre seats in the theatre reservation domain both act as *resources* in relation to their key state transitions. On the other hand bookings act as *inputs-to-be-met* in that domain. Using the meta-schema we can say:

< stock-item, resource >
< theatre-seat, resource >
< booking, input-to-be-met >

Two types of object can be identified from domain descriptions. Key objects are those which move with respect to a structure in a state transition. Structural objects on the other hand define the static structure in which key objects move. This fundamental distinction between object types in software engineering domains will be elaborated later in this chapter. Object types may be recognised easily by software engineers, so enhancing the comprehensibility of domain models. Indeed, they already have been used as a basis for analogical matching in design reuse (Lee & Harandi 1991).

3.2.6 Domain Events & System Functionality

State transitions can be defined further by domain events and the functionality of the information system expressed as language. This elaboration assists analogical matching between key state transitions by linking each transition to a prominent event or function. Events in the domain cause state transitions, for instance the *lend* event in a lending library system causes a transition of books from an in-library to an on-loan state. System functions on the other hand are associated with state transitions, for example the function *allocate* is associated to a state transition moving a theatre booking from a waiting-list to an occupying-resource state. Other examples of domain events include *lend, return and goods-in*. Example system functions are *monitor* and *check-status*. Each state transition is linked to the event or function associated most prominently with the transition.

Furthermore, system functions can be linked to state transitions in two ways. First, an existing system function can be associated directly with events which cause transitions between states, for instance the *allocate* function in the theatre domain is associated closely with changing a reservation from an unoccupying to an occupying-seat state. Thus, system functions are associated with a change in the domain expressed as a state transition. Second, state transitions can identify the need for the function through the process of system specification, for instance in the ATC domain, the state transition representing aircraft movements between airspaces causes the functional requirement to *MONITOR* for potential aircraft collisions to be specified. In this case, state transitions imply functionality through the process of specification. However, for the purposes of defining the meta-schema, the distinction between system functionality and domain events is removed to permit simple attribution of linguistic terms to improve definition of key state transitions. Therefore domain events and system functions are defined using the meta-schema as:

< allocate, booking, reservation, seat, many >
< allocate, application, pending, place, many >

Domain events, system functions and state transitions are defined in a single predicate during the examples in the rest of this chapter.

3.2.7 The Role of Functions in State Transitions

Finally, state transitions can be differentiated by the role of associated system functions in the transition. State transitions may be achieved by functions, for instance in the theatre domain the *allocate* function is needed to achieve the transition of the booking from the unoccupied to occupying-resource state. A similar distinction was made between internal and external actions equivalent to state transitions in Dardenne et al.

(1991). Therefore functions' roles can differentiate between key state transitions, although this knowledge type in the meta-schema plays a less important role in analogical matching than other types.

3.2.8 Summary of the Proposed Meta-schema

This meta-schema of seven knowledge types has been used to represent many different domains. The next section justifies the proposed meta-schema by examining constraints on analogical mapping from existing cognitive and computational models of analogy. For instance, many features of the FMS domain described in chapter 2, such as the two-dimensional layout of production tracks, cannot be transferred analogically to the ATC domain. Two computational and cognitive models of analogical reasoning (e.g. Gentner 1983) were used to constrain analogical matching between software engineering domains and validate the knowledge types defined in the meta-schema.

3.2.9 Constraints on Analogical Mapping

The meta-schema of knowledge types was validated by examining constraints on the analogical mapping of domain knowledge taken from cognitive and computational models of analogy. The use of these constraints is also demonstrated by three non-simple examples of analogy presented at the end of this chapter. They are structural and semantic constraints (Gentner 1983, Holyoak & Thagard 1989), which ensure that analogy only maps an interrelated, semantically-equivalent network of facts, and abstraction, which ensures that analogy only maps facts belonging to a known class of domain (Greiner 1988a). The aim of this exercise was to validate the knowledge types defined in the meta-schema using computational and cognitive models of analogical reasoning. Both constraints are examined in turn.

3.2.9.1 Isomorphic Constraints on Analogical Mapping

Isomorphism has been suggested as an important constraint on analogical matching (Weitzenfeld 1984, Holyoak & Thagard 1989). Gentner's structure-mapping theory (1983) constrains analogical mapping by transferring an interrelated knowledge structure rather than unrelated facts. Her systematicity principle states that a source domain predicate that belongs to a mappable system of mutually interconnecting relations constrained by higher-order causal structures is more likely to be imported into the target system than an isolated predicate. Similarly, this thesis proposes that analogical reuse transfers a network of interrelated domain knowledge predicates defined in the meta-schema.

Gentner (1983) transfers interrelated knowledge structures by semantically matching a set of relations which link equivalent objects in the source and target domains. These relations are independent of specific domains, thus providing a syntactic basis for identifying analogical similarity. Systematicity in software engineering analogies is possible because the knowledge types in the meta-schema define semantic relations between domain objects, as shown graphically in Figure 3.3. Using Gentner's model as a starting point, analogical matching between software engineering domains requires a restricted set of domain terms for instantiating the knowledge types in the meta-schema. These terms are dependent upon the domain classes to be represented using the meta-schema, hence a taxonomy of domain types is needed in the framework of software engineering analogies.

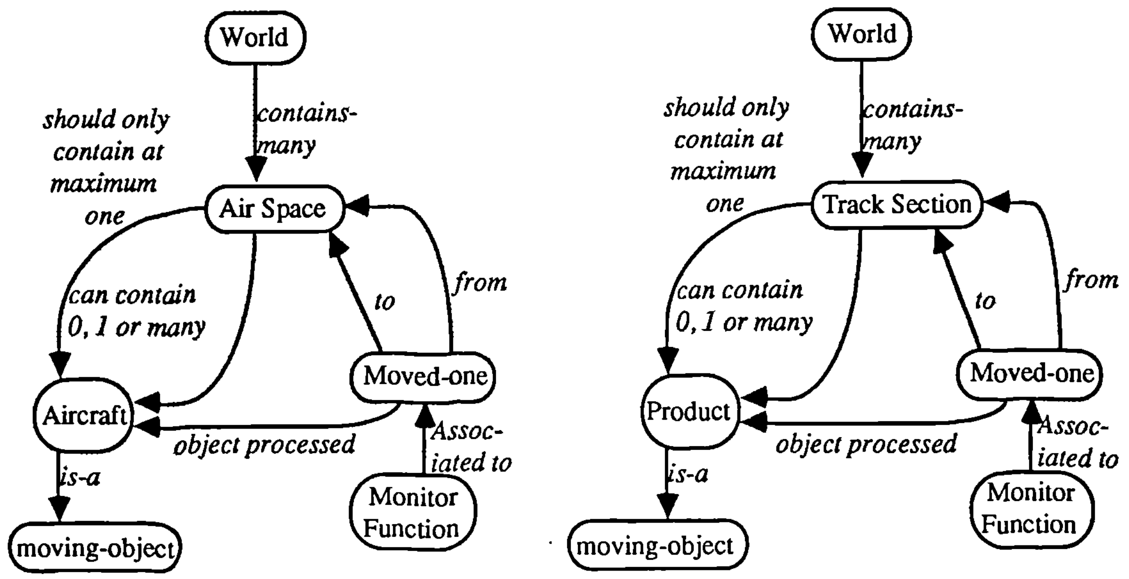


Figure 3.3 - systematicity between domain terms representing the ATC and FMS domains (labels on object relations represent domain terms)

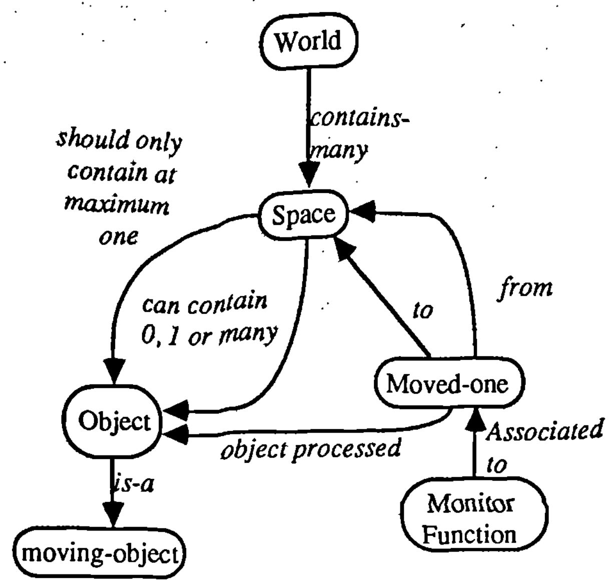


Figure 3.4 - domain abstraction representing knowledge structures common to the ATC and FMS domains

An alternative approach to imposing systematicity between analogical software engineering domains is abstraction. Two software engineering domains are analogous if they are both instances of the same abstraction. Abstractions can impose isomorphism on analogical matching by mapping a source and target domain to their shared abstraction, see Figure 3.4. Furthermore, these abstractions represent known domain types, so they may identify a set of terms able to represent domain instantiations and their abstractions in the meta-schema. Therefore, abstraction appears to be an important concept during analogical matching.

3.2.9.2 Abstract Constraints on Analogical Mapping

Gick & Holyoak (1983) and Greiner (1988a, 1988b) constrained analogical matching by only mapping knowledge belonging to an abstraction shared by the target and reusable domains. Studies of natural language metaphors and analogical problem solving reported in chapter 2 also emphasised the importance of class- or type-inclusion during analogical reasoning. Returning to one of the example analogies, the ATC and FMS domains can be expressed as two instantiations of an abstract domain class in which objects move in a space, risk collision if they fail to follow a predetermined plan and are controlled remotely by people. For software engineering domains, analogy is more likely to support specification reuse if the two analogically-matched domains belong to a shared abstraction. Indeed, the desire for software reuse arose from the recognition that similar problems were constantly being re-solved, suggesting that specification reuse may only occur between applications belonging to the same domain class.

Furthermore, by combining abstraction with structural isomorphism, domain abstractions can be endowed with critical features belonging to all instances of a domain class, see Figure 3.4. This however, necessitates one set of terms for describing both domain abstractions and their instantiations, to permit recognition of analogical similarity. The other, obvious drawback is the need for a predetermined library of appropriate *domain abstractions*, so a taxonomy of software engineering domains is needed for both analogical matching and determining terms to represent these domains.

It is important to emphasise the difference between reuse of domain abstractions and generic templates (e.g. Harandi & Young 1985) discussed in chapter 2. Templates represent abstract solutions in the design space while this thesis proposes use of domain abstractions for matching between reusable domain spaces. Deriving a complete and correct set of template solutions to even a small set of problems has proven difficult. Any one problem may be solved in many ways, so the space of candidate specifications is large and potentially difficult to identify. For instance, the scheduling problem in the production planning/video hiring analogy described at the end of this chapter may be solved using the following algorithms:

- simple matching of unordered resources against unordered requirements;
- prior sorting of resources and requirements to ensure difficult allocations are made first;
- linear programming techniques;
- controller intervention in the allocation process, for instance the controller has the ability to make priority allocations before running the scheduling routine.

It may be easier to derive a *useful* set of domain abstractions representing a tractable set of domain types than to derive a set of design abstractions. These abstractions provide an analogical bridge for recognising, understanding and transferring similarities between two software engineering domains, see Figure 3.1. Distinguishing critical determinants of reuse from knowledge held in specifications has important implications for how effective reuse can be achieved.

A classification of software engineering domains also allows the identification of terms for representing these domain instances. The current set of domain abstractions is shown in Appendix A. It is derived from many sources including textbooks and academic case studies. Examples reveal that many domains can be represented effectively by a small set of domain terms, suggesting that, at the proposed level of abstraction, many domain classes only have a few key differences.

To sum, it is hypothesised that abstraction can constrain analogical specification reuse, although reuse is limited to between instantiations of known domain abstractions. Induction of generic domain classes from many solution instances has been proposed as one solution to deriving these abstractions (Lee & Harandi 1991a, 1991b, Harandi & Lee 1991), however automatic generation of abstractions appears unrealistic for two reasons. First, genuine automatic induction can only be achieved from very large numbers of analogical specification instances which may not be available in many CASE repositories. Second, current machine-learning techniques require complete domain theories to induce facts about simple domains such as blocksworld (e.g. Chenoweth 1991, Gutpa & Nau 1991), so their applicability to software engineering domains is limited. Instead, domain abstractions must be derived and evaluated manually.

3.2.10 Domain Terms Defining the Meta-schema

Domain terms are required to define the meta-schema, represent known software engineering domains in Appendix A and allow analogical matching and explanation. For the purposes of this chapter, 10 domain abstractions were selected to derive and demonstrate a workable subset, although no claim to completeness and extent of coverage of software engineering domains is made. The current set of terms are not specified formally, although this remains a distinct possibility in future work.

3.2.10.1 Object Structure

Object structures describe the relationship between domain objects as set memberships which identify the cardinality and optionality of this membership. Object structures are described using 6 terms divided into *has* and *contain* relations. The *has* relations represent object structures unaffected by state transitions while *contains* relations describe states which may alter as a result of transitions. The *has/contains* distinction is similar to relation optionality in entity-relationship diagramming:

A has-no B:	this states that A is an empty set with regard to B;
A has-one B:	one object B is always found in A, for example the theatre domain example is always populated by one theatre;
A has-many B:	many B objects are always found in A, for example the theatre is always populated by many seats;
A contains-no B:	this states that A may be an empty set with regard to B, for example a

	theatre seat may contain no booking, implying that the seat is available;
A contains-one B:	one instance B may be found in A, for example a theatre seat may contain a customer booking;
A contains-many B:	one A may be populated by many Bs, for example the performance waiting list may contain many unmet reservations.

3.2.10.2 Domain Requirements

Domain requirements elaborate object structures through addition of language statements identifying goals and constraints for the required system. The current set of requirements represent functional requirements and required domain states. Four requirements were identified for the 10 domain abstractions:

maximum-quantity:	the system is aiming to achieve a *maximum quantity* of the specified set membership, for example in the theatre reservation system the system requirement is to maximise bookings in theatre seats ,i.e. the number of booked seats in the set;
minimum-quantity:	the system is aiming to achieve a *minimum quantity* of the specified knowledge state;
same-properties:	set membership is dependent upon the containing and contained objects having the same properties, for example customer bookings and theatre seats must share similar properties before allocation occurs,
date-limit:	the system should only permit the required knowledge state until a specific date or time period. For example, in a library lending domain, books should be returned by borrowers by the end of the loan period.

3.2.10.3 State Transitions

Two terms exist to describe state transitions between object structures B and C:

move-one A from B to C:	only one object A is moved from B to C by the state transition, for instance movement of one aircraft in the ATC domain is independent of movement from other aircraft, so each transition only moves one aircraft;
move-many A from B to C:	several objects are moved instantaneously from B to C, for example in the theatre domain one reservation can book many seats, so several bookings are allocated to theatre seats by the state transition.

3.2.10.4 Object Types

Object types can be divided into key and structural domain objects, as described earlier in this chapter. Three key and three structural object types were identified for the 10 domain abstractions. Key object types are:

resource:	the object is used by the system for meeting its requirements, for example theatre seats in the theatre reservation system acts as system *resources* for meeting its requirements;
input-to-be-met:	the object acts as a need to be met by resources, for example a theatre seat booking can be typed as a need to be met by theatre bookings;
moving-object:	the object moves in relation to object structures so that it occupies different positions in a space.

Structural object types were:

list:	the object is a list containing an ordered set of other objects, for example the waiting lists in the theatre reservation and university course administration domains are instances of *list*;
container:	the object acts as a container for other objects (often resources), for instance the video library may be a *container* since it contains available video copies;
receptacle:	the object acts as the final destination of processed objects moved by key state transitions.

3.2.10.5 Preconditions on State Transitions

Four preconditions describing triggers for state transitions were identified:

minimum-quantity:	the state transition only occurs when the level of objects in its initial position reaches a *minimum quantity*, for example in a stock control domain, goods are only restocked when the level of goods reaches a prespecified minimum (see Appendix A);
maximum-quantity:	the state transition only occurs when the level of objects in its initial position reaches a *maximum quantity*;
same-properties:	the state transition only occurs when properties of the object moved in the transition *match* those of objects in the final position of the transition. For example, in the theatre reservation domain, bookings are only allocated to seats if booking and seat share the same properties, such as non-smoking, price, location etc.;
date/time-limit:	the state transition occurs at specific times or dates, for example, in a library lending domain, books are often returned by borrowers at the end of a loan period. Similar terms can be developed for other state attributes such as pressure, temperature etc..

3.2.10.6 Domain Events & System Functions

Domain events and system functions represent the most salient characteristics of key state transitions. Domain events describe state transitions while system functions represent high-level design algorithms for solving software engineering problems. A lexicon of combined functions and events has been developed for the 10 example abstractions. These are: *loan, borrow, dispatch, send, lend, goods-out, receipt, input, goods-in, arrival, addition, allocate, assign, place, correct, join, return, finish-loan, check-position, monitor and record.* To repeat, these domain events and system functions elaborate the definition of key state transitions, thus improving the likelihood of an analogical match with a domain description.

3.2.11 Examples Demonstrating the Meta-Schema of Knowledge Types

The meta-schema of knowledge types was evaluated by the examples shown in Appendix A. These examples were drawn from many sources, including the author's previous software development experiences, textbooks, academic case studies and benchmark applications offered by attendees at conferences to test the current set of domain abstractions. They represent domains more often found in business rather than real time information systems which is indicative of their source. The meta-schema is demonstrated using four example software engineering analogies taken from Appendix A. Each example is represented using terms defined as typed predicates. Each of the four example analogies is described in turn.

The Theatre/Course Administration Example

The instantiated meta-schema for the theatre and course administration domains is represented graphically in Figure 3.5.

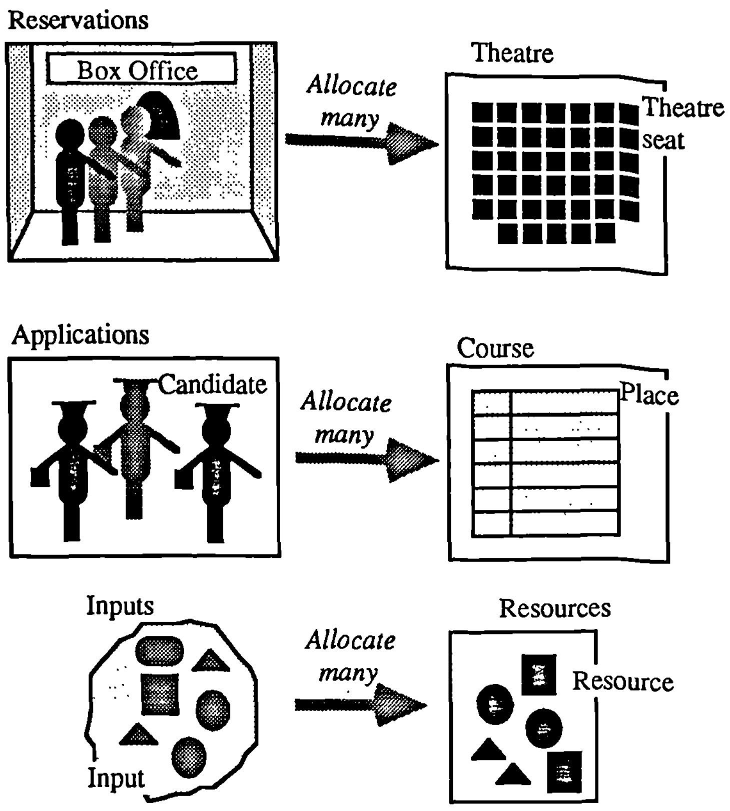


Figure 3.5 - graphic representation of the theatre reservation, course application and object allocation domains

Typed predicates which provide the basis for analogical matching between domain descriptions are highlighted in **bold**:

function/event/transition (**allocate**, booking, reservation, seat, **many**)
condition (**same-properties**, booking, reservation, seat, **many**)
object structure (world, reservation, **has-many**)
object structure (reservation, booking, **contains-many**)
object structure (world, theatre, **has-one**)
object structure (theatre, seat, **has-many**)
object structure (seat, booking, **contains-one**)
object structure (seat, booking, **contains-no**)
domain requirement (seat, booking, **contains-one**, **same-properties**)
object category (booking, **different-object-types**)
object category (allocation, **different-object-types**)
function achieving transition (allocate)

function/event/transition (**allocate**, application, candidates, place, **many**)
condition (**same-properties**, application, candidates, place, **many**)
object structure (world, candidates, **has-many**)
object structure (candidates, application, **contains-many**)
object structure (world, course, **has-one**)
object structure (course, place, **has-many**)
object structure (place, application, **contains-one**)
object structure (place, application, **contains-no**)
domain requirement (place, application, **contains-one**, **same-properties**)
object category (booking, **different-object-types**)
object category (allocation, **different-object-types**)
function achieving transition (allocate)

The abstraction representing critical facts belonging to both domains is:

function/event/transition (**allocate**, requirement, requirements, resource, **many**)
condition (**same-properties**, requirement, requirements, resource, **many**)
object structure (space, requirement, **has-many**)
object structure (space, resources, **has-one**)
object structure (resources, resource, **has-many**)
object structure (resource, requirement, **contains-one**)
object structure (resource, requirement, **contains-no**)
domain requirement (resource, requirement, **contains-one**, **same-properties**)
object category (requirement, **different-object-types**)
object category (resource, **different-object-types**)
function achieving transition (allocate)

The Lending Library/Car Hire Example

The instantiated meta-schema for the lending library and car hire domains are:

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

function/event/transition (**lend**, car, hirecentre, client, **many**)
function/event/transition (**return**, car, client, hirecentre, **many**)
condition (**date-limit**, car, client, hirecentre, **many**)
object structure (world, client, **has-many**)
object structure (world, hirecentre, **has-one**)
object structure (hirecentre, car, **has-many**)
object structure (client, car, **has-many**)
domain requirement (client, car, **has-many**, **date-limit**)
object category (car, **resource**)
object category (hirecentre, **resource-container**)

The abstraction representing critical facts belonging to both domains is:

function/event/transition (**lend**, object, resource-holder, borrower, **many**)
function/event/transition (**return**, object, borrower, resource-holder, **many**)
condition (**date-limit**, object, borrower, resource-holder, **many**)
object structure (world, borrower, **has-many**)
object structure (world, resource-holder, **has-one**)
object structure (resource-holder, object, **has-many**)
object structure (borrower, object, **has-many**)
domain requirement (borrower, object, **has-many**, **date-limit**)
object category (object, **resource**)
object category (resource-holder, **resource-container**)

The Stock Control/Car Pool Maintenance Example

The meta-schema representing the stock control and car pool maintenance domains are:

function/event/transition (**delete**, stock, bin, customer, **many**)

function/event/transition (**resupply**, stock, supplier, bin, **many**)
condition (**minimum-quantity**, stock, supplier, bin, **many**)
object structure (world, customer, **has-many**)
object structure (world, supplier, **has-many**)
object structure (world, warehouse, **has-one**)
object structure (warehouse, bin, **has-many**)
object structure (customer, stock, **contains-many**)
object structure (supplier, stock, **contains-many**)
object structure (bin, stock, **contains-many**)
domain requirement (bin, stock, **contains-many**, **maximum-quantity**)
object category (stock, **resource**)
object category (bin, **resource-container**)

function/event/transition (**delete**, car, carpool-type, out-of-service, **many**)
function/event/transition (**resupply**, car, dealer, carpool-type, **many**)
condition (**minimum-quantity**, car, dealer, carpool-type, **many**)
object structure (world, out-of-service, **has-many**)
object structure (world, dealer, **has-many**)
object structure (world, carpool, **has-one**)
object structure (carpool, carpool-type, **has-many**)
object structure (out-of-service, car, **contains-many**)
object structure (dealer, car, **contains-many**)
object structure (carpool-type, car, **contains-many**)
domain requirement (carpool-type, car, **contains-many**, **maximum-quantity**)
object category (car, **resource**)
object category (carpool-type, **resource-container**)

Again, the abstraction representing critical facts belonging to both domains is:

function/event/transition (**delete**, object, small-container, source, **many**)
function/event/transition (**resupply**, object, sink, small-container, **many**)
condition (**minimum-quantity**, object, sink, small-container, **many**)
object structure (world, source, **has-many**)
object structure (world, sink, **has-many**)
object structure (world, large-container, **has-one**)
object structure (large-container, small-container, **has-many**)
object structure (sink, object, **contains-many**)
object structure (source, object, **contains-many**)
object structure (small-container, object, **contains-many**)
domain requirement (small-container, object, **contains-many**, **maximum-quantity**)
object category (object, **resource**)
object category (small-container, **resource-container**)

The Air Traffic Control/Flexible Manufacturing Example

The meta-schema instantiated in the air traffic and flexible manufacturing domains are:

function/event/transition (**monitor**, aircraft, airspace, airspace, **one**)
object structure (world, airspace, **has-many**)
object structure (airspace, aircraft, **contains-one**)
object structure (airspace, aircraft, **contains-many**)
object type (aircraft, **moving-object**)
domain requirement (airspace, aircraft, **contains-one**)

function/event/transition (**monitor**, product, track-section, track-section, **one**)
object structure (world, track-section, **has-many**)
object structure (track-section, product, **contains-one**)
object structure (track-section, product, **contains-many**)
object type (product, **moving-object**)
domain requirement (track-section, product, **contains-one**)

The abstraction representing critical facts belonging to these final two domains is:

> function/event/transition (**monitor**, object, space, space, **one**)
> object structure (world, space, **has-many**)
> object structure (space, object, **contains-one**)
> object structure (space, object, **contains-many**)
> object type (object, **moving-object**)
> domain requirement (space, object, **contains-one**)

Summary of the Four Examples

Key domain terms (shown in **bold** in the above examples) provide the basis for analogical matching during specification retrieval and explanation. The likelihood of an analogical match is enhanced by a coherent structural match between mapped predicates. The four examples demonstrate that, according to the proposed model, analogical matching occurs between software engineering domains equivalent in scale to medium-sized entity-relationship diagrams. They also indicate domain terms for representing and matching software engineering domains defined more fully in the next section.

3.2.12 The Meta-Schema of Knowledge Types: A Summary

A meta-schema of knowledge types for representing key facts about software engineering domains is a major component of the framework of software engineering analogies. It differs from existing meta-schema (e.g. Greenspan 1984, Lubars 1988) in its focus on key domain facts rather than comprehensive domain models which identify both key and non-critical domain facts for analogical matching (e.g. Dardenne et al. 1991). Knowledge types defined in the meta-schema were validated using constraints on analogical matching borrowed from cognitive and computational models of analogy, namely structural isomorphism and instantiation of the same domain abstraction. The meta-schema was evaluated by example and demonstrated in this chapter using four pairs of analogical software engineering domains. Abstraction is central to analogical matching, so known domain abstractions are also represented using the knowledge meta-schema. The overall paradigm for analogical reuse shows how domain abstractions bridge between a new domain and reusable specifications, see Figure 3.1. These abstractions are separated from reusable specifications, thus maximising the potential payoff from any single match. Finally, the subset of defined domain terms are capable of representing a wide range of software engineering domains, indicating that they are powerful domain descriptors.

The meta-schema of knowledge types provides few clues about the range of domain abstractions. A process for categorising and representing these domain abstractions is needed to assist analogical matching and explanation. The next section proposes a logical model of domain abstraction which identifies a framework for structuring and instantiating domain abstractions to support analogical matching and explanation.

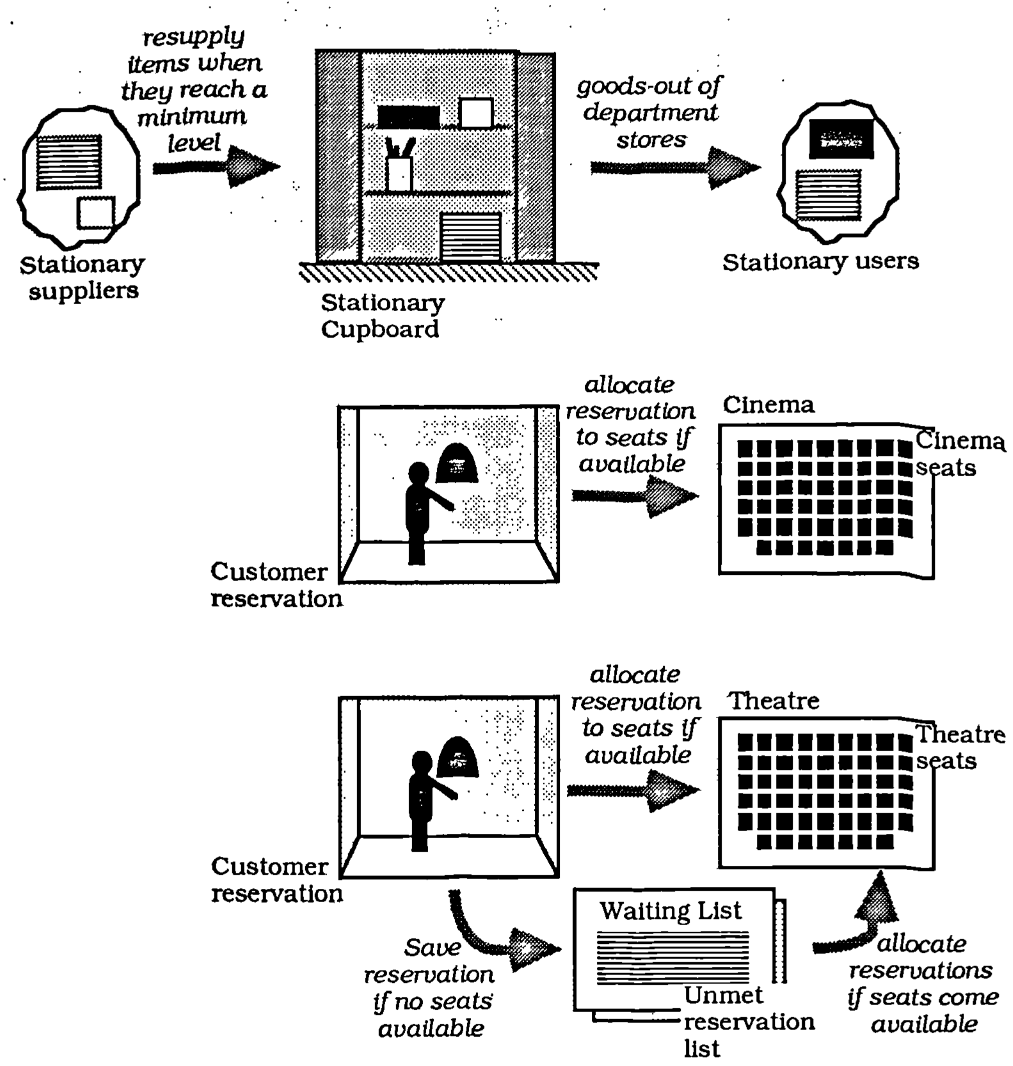
3.3 A Logical Model of Domain Abstraction

A logical model of domain abstraction is proposed to identify similarities and differences between domain abstractions based on knowledge types defined in the meta-schema. It is described in two parts. First, a model of abstraction is proposed, then domain abstractions are instantiated to identify common generic domain worlds, for instance the object monitoring and plan adherence abstractions underlying the ATC domain can also be instantiated in other safety-critical transport worlds including train signalling and monitoring of harbour shipping movements. First however, similarities and differences between domain abstractions are defined.

3.3.1 The Structure of Domain Abstractions

Knowledge types in the meta-schema provide a theoretical basis for distinguishing between as well as identifying domain abstractions. Domain abstractions are represented in a hierarchy, grouped first by key state transitions then specialised at lower levels of the hierarchy by other knowledge types. The assumptions that this model rests upon are drawn from cognitive models of memory, and in particular the hierarchical models of natural categories (Rosch et al. 1976) and hierarchical memory schema (Anderson 1990). These assert in slightly different forms that human memory is organised in an informal hierarchy of classes. Each domain model in the hierarchy inherits all the features of its father and specialises it to represent a sub-type. This hierarchical form assists the retrieval, selection and explanation of single domain abstractions when matching domain descriptions of equivalent scale and detail.

The central hypothesis of the model is that domain abstractions are differentiated by key state transitions in respect to an object structure, hence a non-renewable resource abstraction, of which library *loans* is an example, can be distinguished from a renewable resource abstraction (e.g. stock control) by the key transition of *return*. Similarly, a simple object allocation abstraction, of which a cinema booking domain is an example, can be differentiated from a complex object allocation abstraction such as the theatre domain by the inclusion of key transitions which *send* and *remove* bookings from a waiting list, see Figure 3.6.

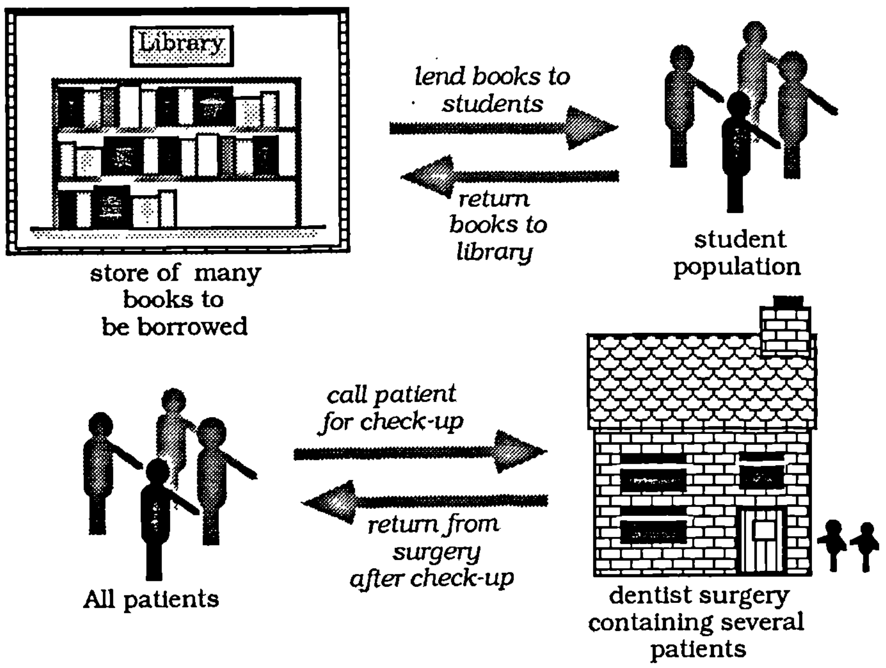


resupply items when they reach a minimum level
Stationary suppliers
Stationary Cupboard
goods-out of department stores
Stationary users
allocate reservation to seats if available
Cinema
Cinema seats
Customer reservation
allocate reservation to seats if available
Theatre
Theatre seats
Customer reservation
Save reservation if no seats available
Waiting List
Unmet reservation list
allocate reservations if seats come available

Figure 3.6 - example stock control, local cinema and theatre domains

Object structures describing domain states can also distinguish between different abstractions. The library and stock control domains and can be differentiated, for instance only the latter domain has the *supplier* concept linked to the *goods-in* action and its key state transition, see Figure 3.6. Similarly, the theatre domain implements a waiting list policy for unmet bookings while the local cinema does not, so the former domain model includes the waiting-list structure to support the key *send* and *remove* actions. At least two domain abstractions can be identified to represent this concept of object allocation (see Appendix A).

Preconditions on state transitions identify further differences between domain abstractions. Consider the example of the lending library and dental patient domains shown in Figure 3.7. Both domains have similar key state transitions with respect to object structures, however preconditions on these transitions differ. Books must be returned to the library by a certain date or time. On the other hand, calls for patient check-ups are triggered periodically, hence the preconditions on the two state transitions are fundamentally different. As a result, preconditions on state transitions can distinguish between domain abstractions with similar state transitions and object structures.



Library
lend books to students
return books to library
store of many books to be borrowed
student population
call patient for check-up
return from surgery after check-up
All patients
dentist surgery containing several patients

Figure 3.7 - example library and dental surgery domains

Information system functions and domain events can also help to distinguish between domain abstractions. Analogy aims to support reuse between functionally-equivalent system specifications, so it is not surprising that functions and events may be able to distinguish between otherwise similar abstractions. The two examples shown in Figures 3.6 and 3.7 demonstrate this. In the first example functions and events such as *stock-out & meet-order* differ from *allocate* or *assign*, while differences between *lend*, *return* and *call* events exist in the second example in Figure 3.7.

Finally objects types can help distinguish domain abstractions. The abstractions underlying the library lending and dental patient check-up domains have equivalent state transitions and object structures, however matched objects in both domains have very different roles. In the library, books act as *resources* which are stored in a *resource-container* known as the library. On the other hand, patients act as *customers* which are served in the *service-area* which is the dentist's surgery (note that these terms were not defined in the meta-schema because the abstraction was not was to define the meta-schema). As such, a closed set of object types can provide additional terms to distinguish between domains which cannot be conveyed using other knowledge types. The derivation of such a set is dependent upon the range of domain abstractions incorporated into the framework of software engineering analogies.

To summarise, knowledge types defined in the meta-schema both determine and differentiate between domain abstractions. Abstractions are grouped by key state transitions with respect to object structures to identify key differences between abstract domain classes. Within these major groupings domains are distinguished further by preconditions on state transitions, domain events and system functions linked to state transitions, triggering events of these transitions and object types. An example of a class hierarchy of 10 domain abstractions is shown in Appendix J. It identifies four main domains differentiated by key state transitions. The object monitoring abstraction monitors the movement of objects between spaces to avoid collisions (e.g. *air traffic collision avoidance*). The object positioning abstraction examines object movement to ensure that they occupy required positions. The object allocation abstraction assigns demands for objects to available resources assuming that they meet prespecified constraints. Furthermore this abstraction can be specialised to represent a more complex multiple object allocation abstraction by adding object structures (e.g. *theatre seat allocation*). Finally, the object containment abstraction describes the movement of objects out of a container. It can be specialised to the non-renewable (e.g. *lending library*) and basic renewable resource abstraction, the latter of which can be specialised further by addition of object structures (e.g. *warehouse has-many bins*) to represent abstractions of either a complex stock control or personnel domain.

3.3.2 Generic Domain Worlds

The proposed framework for domain abstraction is developed further by specialising domain abstractions to generic domain worlds, see Figure 3.8. Application-independent domain abstractions are necessary for analogical matching and explanation, however domains can also be modelled at lower levels of abstraction. The model is extended to include intermediate levels of abstraction for each domain class to provide more concrete knowledge about software engineering domains, thus improving analogical retrieval and explanation of specifications.

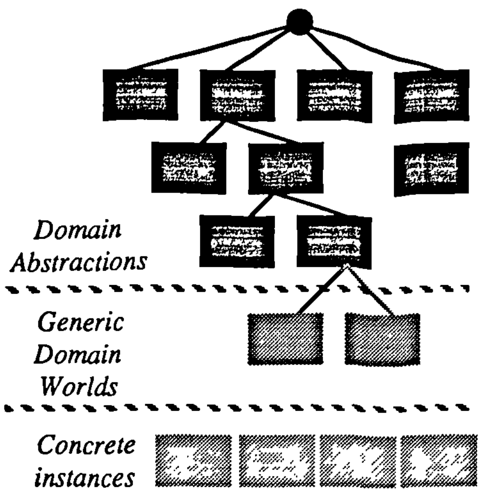


Figure 3.8 - overview of the model of domain abstraction, demonstrating how generic domain worlds provide additional domain knowledge to assist analogical matching and explanation

Chapter 2 documented the considerable research interest in application templates and domain modelling indicative of the reuse potential at lower levels of abstraction. While this thesis does not propose to model specific applications for reasons stated in chapter 2, it models generic domain worlds to assist analogical matching and explanation. Two new knowledge types are added to the meta-schema to incorporate generic domain worlds for each domain abstraction. They impose physical attributes on each key object in that class:

domain abstraction < domain, **generic-domain-world** >
object property < object, **physical-property** >

The role of generic domain worlds is best demonstrated by several examples. The air traffic control domain is one instance of a domain class in which objects must avoid collision and accidents are likely to lead to loss of life, so the following extensions can be made to the domain abstraction:

domain abstraction < object-monitoring, **safety-critical-transport** >
object property < object, **manned-vehicle** >
object property <space, **safety-zone protecting manned-vehicle** >

The abstraction can be instantiated as:

domain abstraction < air-traffic-control, **safety-critical-transport** >
object property < aircraft, **manned-vehicle** >
object property < airspace, **safety-zone protecting manned-vehicle** >

Other equivalent generic domain worlds which instantiate the same domain abstraction include train safety and ship movements in a harbour, so many applications can be instantiated to the safety critical transport generic domain world. As such, analogical matching would favour retrieval of specifications belonging to the same generic domain world over those which do not, indicated in **bold** in the above example. A second, similar example represents instantiation of the non-renewable resource abstraction to a car hire domain. The generic domain world is defined as:

domain abstraction < non-renewable-resource-management, **rental** >
object property < object, **single-rentable-item** >
object property < resource-container, **source-of-rental-items** >
object property < resource-borrower, **borrower-of-rentable-items** >

An instantiation of this generic domain world can be:

domain abstraction < care-hire, **rental** >
object property < car, **single-rentable-item** >
object property < carpool, **source-of-rental-items** >
object property < client, **borrower-of-rentable-items** >

Furthermore, other knowledge types can be added to the meta-schema to represent facts about generic domain worlds which help lower-level analogical matching. For instance, some facts about the physical layout of space in airlanes, shipping lanes or railway lines can be transferred between all instances of object monitoring in a safety-critical transport world. These physical structures can be instantiated for the FMS and harbour shipping control domains respectively to assist lower-level matching as shown in chapter 5:

physical structure < production-track, track-section, **in-sequence** >
physical structure < sea-lane, lane-section, **in-sequence** >

To conclude, representation of known domain classes at intermediate levels of abstraction, such as generic worlds, can extend the logical model of domain abstraction and refine analogical matching and explanation. However, several problems still remain, most important among which are the coverage and granularity problems discussed in chapter 7.

3.3.3 The Logical Model of Domain Abstraction: A Summary

This logical model identifies similarities and differences between domain abstractions. The overall model is shown pictorially in Figure 3.8. Domain abstractions are key to analogical retrieval and explanation while generic domain worlds provide additional, non-critical domain knowledge to supplement and assist this analogical matching. As such, the model attempts to overcome the granularity problem which has hindered development of successful software component libraries. General heuristics of software reuse (Biggerstaff & Richter 1987) indicate that larger abstractions can provide greater payoff in terms of more extensive specification reuse, at the cost of fewer analogical matches. On the other hand, small abstractions may increase the likelihood of analogical matching at the expense of smaller knowledge transfer. This research has tentatively identified the scale of domain abstractions which might maximise reuse, although the validity of this claim remains to be determined. The remainder of this chapter demonstrates the logical model of software engineering analogies using three non-simple examples.

3.4 Non-Simple Examples of Software Engineering Analogies

The first example demonstrates the role of domain terms during analogical matching. The second example also demonstrates the need for domain terms defining the meta-schema during selection of a source domain which shares few syntactic similarities with the target domain. The third example demonstrates the issues involved when scaling up analogical matching to larger and more complex software engineering domains.

3.4.1 The Underground Railway Signalling (RS) Example

This example investigated an analogical match between an underground railway signalling domain and the ATC and FMS domains described in chapter 2. It demonstrates the importance of the defined domain terms for analogical matching.

An underground railway system has a number of lines and several stations on each line. Trains move unidirectionally along these lines. Each tunnel section may only permit one train, however if this rule is violated then the signal controller is warned immediately of impending danger. An overview of the railway signalling system is given in Figure 3.9, and a data flow diagram of the required system is given in Figure 3.10. Extensive reuse between the specifications is possible, for instance reuse occurred between the MONITOR and REPORT processes, as well as between the external entities AIR TRAFFIC and SIGNAL CONTROLLERS, and between the data stores TRAIN LOG and FLIGHT PLAN. The potential extent and depth of reuse can be demonstrated by examining detailed actions in the analogical MONITOR processes (see Figure 3.11), for example both systems flag a train or aircraft position as either safe (ignore) or a clash (inform the system operator). Similar extensive reuse was also possible between the FMS and RS systems.

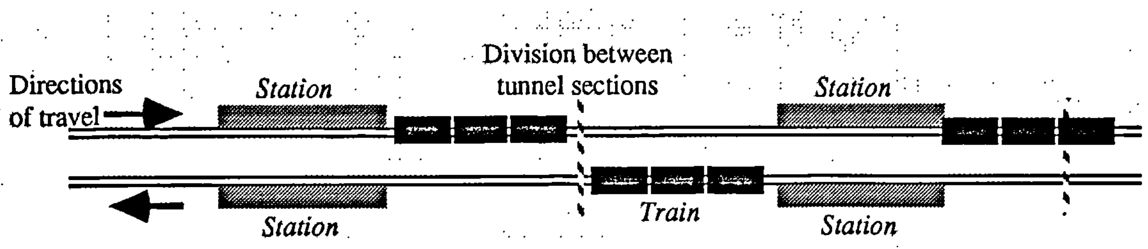


Figure 3.9 - overview of the underground railway signalling domain, showing trains moving unidirectionally between tunnel sections

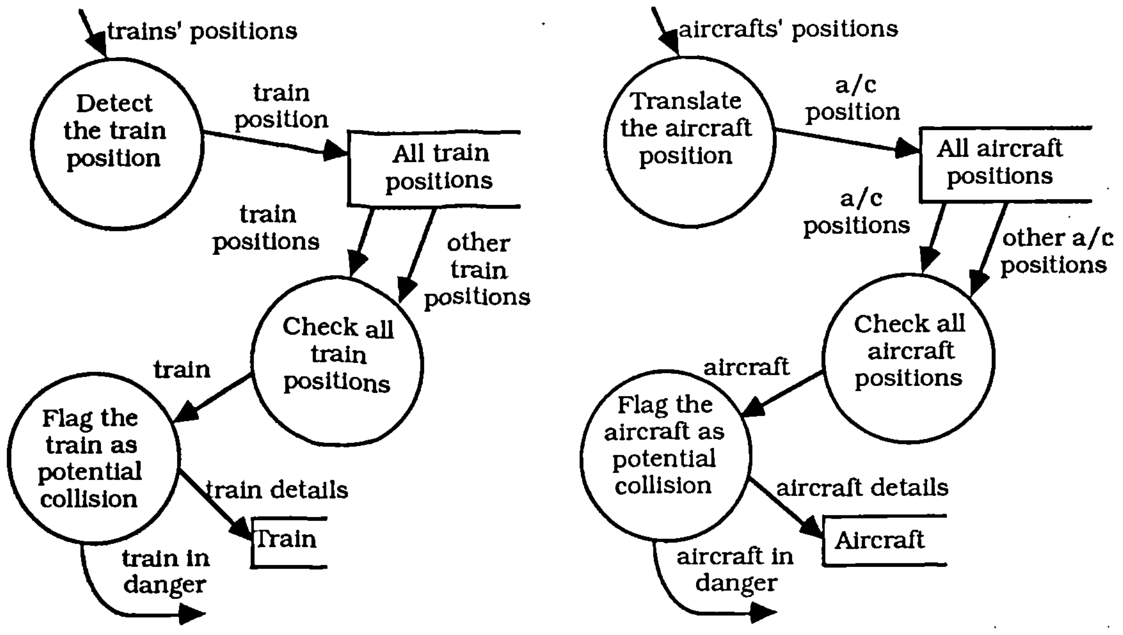


Figure 3.11 - level-2 DFDs demonstrating potential reuse of lower-levels of the RS and ATC specifications

The importance of domain terms is demonstrated by the following description of the railway signalling domain using the meta-schema. Again, key terms for matching domain descriptions are shown in **bold**. This description matches the ATC and FMS domain descriptions described earlier in this chapter.

function/event/transition (**monitor**, train, tunnel-section, tunnel-section, **one**)
object structure (world, tunnel-section, **has-many**)
object structure (tunnel-section, train, **contains-one**)
object structure (tunnel-section, train, **contains-many**)
object type (train, **moving-object**)
domain requirement (tunnel-section, train, **contains-one**)

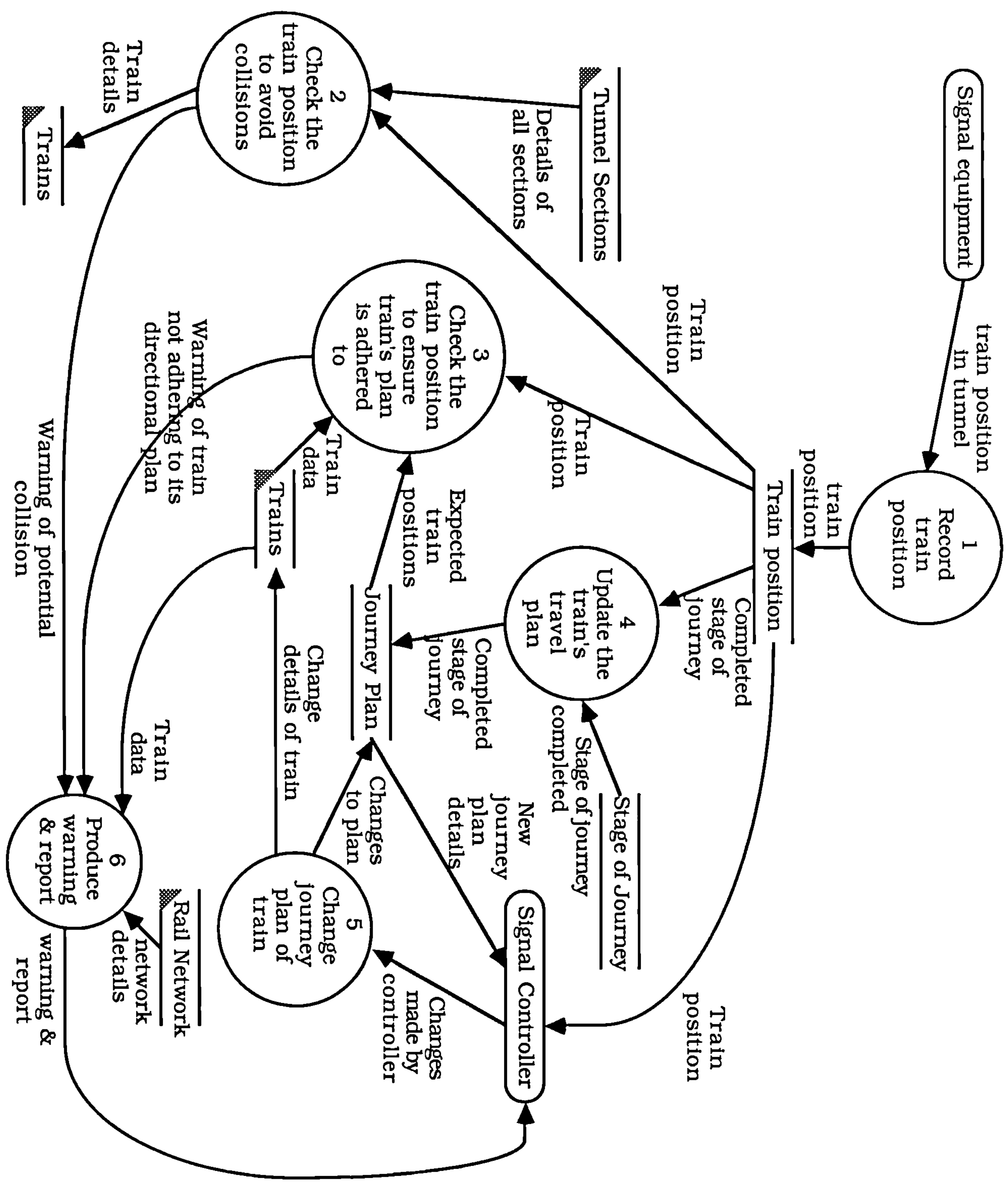


Figure 3.10 - level-0 DFD for the underground railway system

3.4.2 The Video Hiring/Production Planning Example

The analogy between a video hiring and production planning domain is explained more fully in Maiden (1991). It demonstrates the need for knowledge types defined in the meta-schema as a basis for analogical matching and selection. It is investigated in a reuse scenario in which a software engineer is required to specify a production planning system scheduling jobs to machines. This can be achieved by choosing between either the FMS specification or the specification of a video hiring system described below.

3.4.2.1 The Production Planning Domain

A company manufactures a wide range of industrial products using complicated production machinery, robots and conveyor belts which allow partial automation of the process. Production is planned weekly, and at the beginning of each production cycle the production planning system allocates production jobs to manufacturing machines. This allocation process attempts to maximise production output and minimise the idle time of machines and is constrained by the manufacturing requirements of the job and the availability of machines and skilled operators. A JSD process structure diagram describing part of the production planning system is given in Appendix C.

3.4.2.2 The Video Hiring Domain

An organisation rents videos to hotels for use on their internal video systems on a monthly basis. A computer system allocates video copies to hotels within constraints determined by each hotel's requirements (e.g. videos for a VHS system only) and by details of each film (e.g. the length of the film). This allocation function must maximise use of the existing stock of video copies, and ensure that the needs of all hotels are satisfied. A JSD process structure diagram representing part of the video hiring system is given in Appendix C.

3.4.2.3 Analogical Mapping in the Production Planning/Video Hiring Analogy

Uninformed inspection of the two candidate reusable specifications may suggest that the FMS problem is a better analogical match since both domains include a factory layout involving a complex network of conveyor belts along which products move. However, reuse of the FMS specification to specify the production planning system would fail because the two systems are fundamentally different. On the other hand, analogical matching can support reuse with the video hiring specification since both information systems aim to assign resources to inputs to be met effectively. The description of the production planning problem below can be contrasted with the very different FMS description represented earlier in this chapter. Again, structural matching occurs, with semantic similarity defined by typed predicates shown in **bold**.

function/event/transition (**allocate**, job, job-specification, machine, **many**)
condition (**same-properties**, job, job-specification, machine, **many**)
object structure (world, job-specification, **has-one**)
object structure (world, machine, **has-many**)
object structure (job-specification, job, **contains-many**)
object structure (machine, job, **contains-one**)
object structure (machine, job, **contains-no**)
domain requirement (machine, job, **contains-one**, **same-properties**)
object category (machine, **different-object-types**)
object category (job, **different-object-types**)

function/event/transition (**allocate**, hotel-need, hotel-requirements, video-copy, **many**)
condition (**same-properties**, hotel-need, hotel-requirements, video-copy, **many**)
object structure (world, hotel-requirements, **has-one**)
object structure (world, video-copy, **has-many**)
object structure (hotel-requirements, hotel-need, **contains-many**)
object structure (video-copy, hotel-need, **contains-one**)
object structure (video-copy, hotel-need, **contains-no**)
domain requirement (video-copy, hotel-need, **contains-one**, **same-properties**)
object category (video-copy, **different-object-types**)
object category (hotel-need, **different-object-types**)

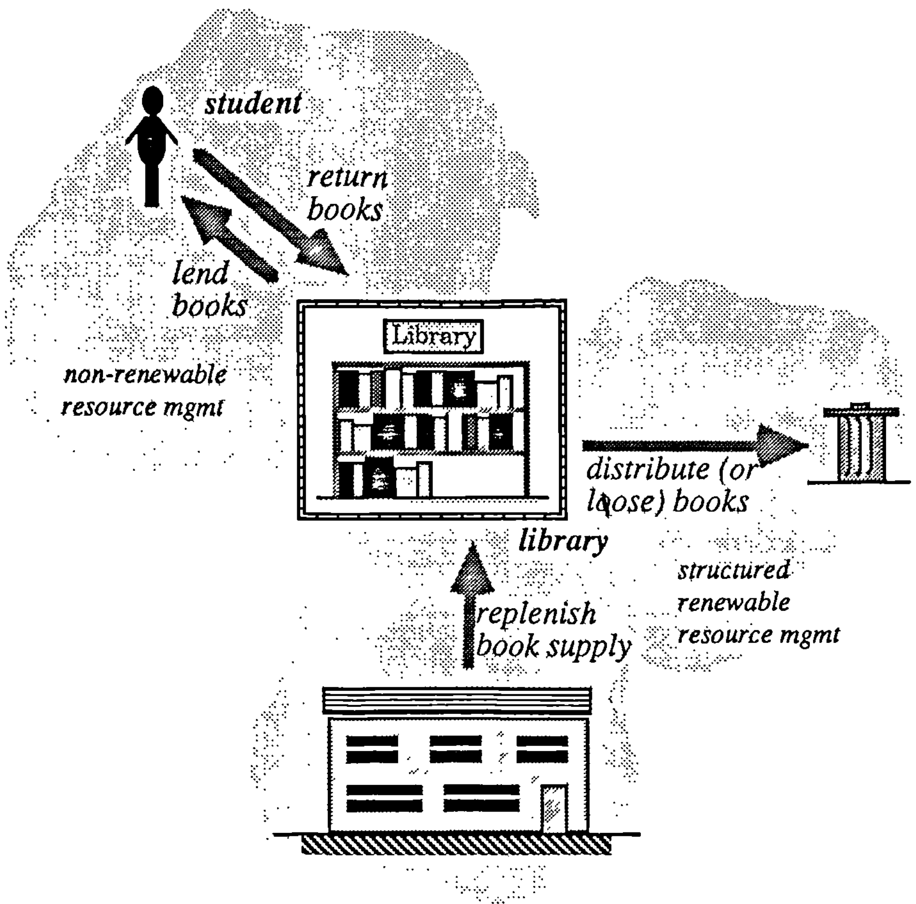
This example demonstrates the importance of identifying key domain facts for both analogical matching and selection, even between domains which share misleadingly similar syntactic properties such as the physical layout of the production floors.

3.4.3 The Local Library/Builders' Supplier Analogy

The third example demonstrates matching, selection and reuse of a larger, more complex specification through several analogical matches, each equivalent to single instantiations of defined domain abstractions. The analogy occurs between a local library system and a system being developed for a supplier of building equipment for both hire and sale.

3.4.3.1 The Local Library Domain

A local library lends books, magazines and videos to the community for fixed periods of time. All borrowers must be members of the library. Reminders and fines are levied on overdue loans, increasing in severity with the length of the overdue loan. The principle aim of the system is to support this lending activity, although it must also maintain a stock of good quality, up-to-date books and videos. As such, the system monitors stock levels to warn of potential short falls in available book and video categories. The library domain is shown pictorially in Figure 3.12.


student
return books
lend books
non-renewable resource mgmt
Library
distribute (or loose) books
library
structured renewable resource mgmt
replenish book supply
Figure 3.12- library lending domain

3.4.3.2 The Builder's Supplier Domain

A supplier to the building trade sells building material and lends equipment to local builders. Equipment is lent over fixed periods of time, and late or damaged return of equipment incurs penalties related to the extent of damage or lateness of return. Sales occur over the counter from the supplier's warehouse. The warehouse must maintain sufficient stock to ensure availability of major items at all times. To this end, stock levels are monitored by the system to ensure that no stock falls below a minimum quantity. The supplier's domain is shown pictorially in Figure 3.13.

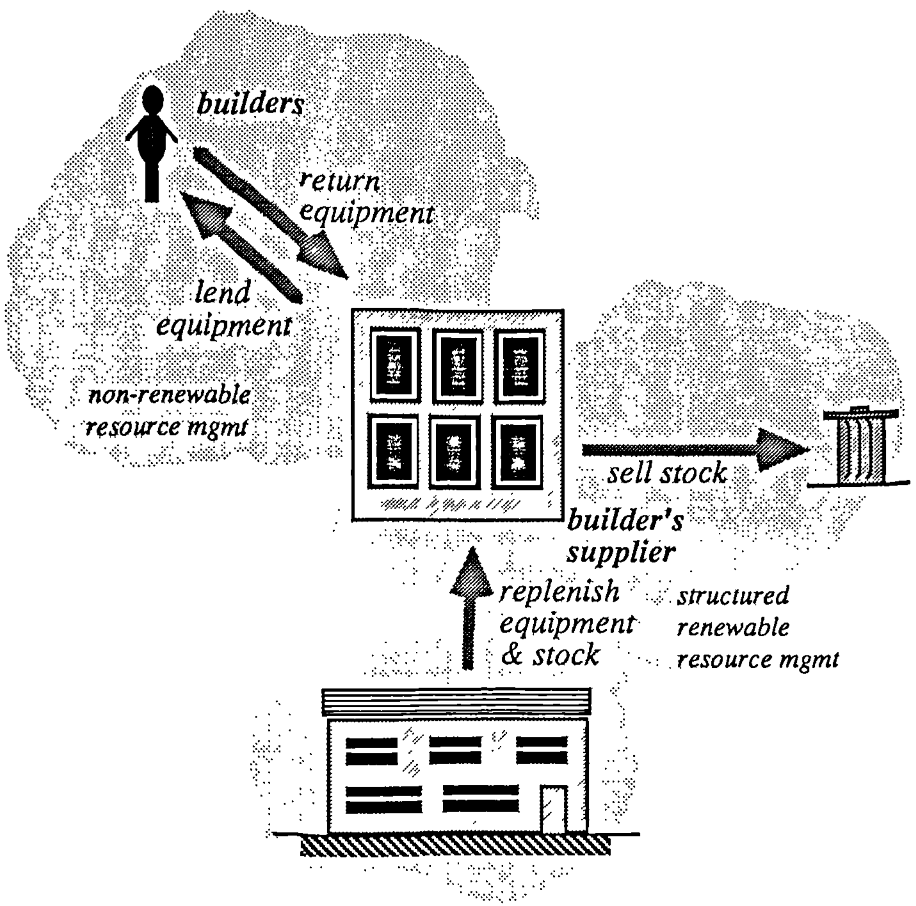

builders
return equipment
lend equipment
non-renewable resource mgmt
sell stock
builder's supplier
replenish equipment & stock
structured renewable resource mgmt

Figure 3.13 - builder's equipment domain

3.4.3.3 Analogical Similarity between the Two Domains

Two abstractions shared by both domains identify the analogical match. These abstractions are defined separately as the non-renewable resource and structured renewable resource abstractions in Appendix A, and shown on the domain descriptions represented pictorially in Figures 3.12 and 3.13. However, these domain abstractions can be aggregated into a larger lending domain which must also maintain its stock of objects to be lent. Such an aggregation can be applied to a variety other lending problems, such as car hire and costume rentals, thus providing additional knowledge to support reuse between the two instantiated domain abstractions. Extension of the software engineering analogy model to incorporate domain aggregations is discussed further in chapter 7.

3.5 Summary: A Logical Model of Software Engineering Analogies

A logical model of key determinants of software engineering analogies was proposed to permit their retrieval and explanation. This model was developed in two parts. First, a meta-schema of knowledge types for representing key facts for analogical retrieval and explanation of specifications was proposed and justified in terms of existing cognitive and computational models of analogical reasoning. Second, a model of domain abstraction was proposed to classify known types of software engineering domains which act as a bridge for analogical matching and explanation.

Analogical matching is constrained by structural isomorphism and abstraction, so that two domains are analogical only if they both belong to a predetermined abstraction representing the key features of that domain class. Currently these abstractions are similar in size to medium-sized entity-relationship diagrams, thus limiting the scale of analogical matches to domains of equivalent scale. Domain abstractions can be specialised to identify common generic domain worlds. A computational implementation of part of the model to support analogical retrieval and explanation is described in chapter 5.

Domain knowledge is needed to provide intelligent support during requirements engineering as well as during analogical reuse. Knowledge structures in requirements engineering have received scant attention thus far, but the meta-schema defined in this chapter provides a foundation for intelligent support for the requirements engineering process. The logical model of software engineering analogies enables tool support for effectively matching and explaining reusable specifications. It defines why analogies occur and how they can be identified. The next chapter investigates analogical comprehension and transfer to prove the specification reuse scenario. Empirical studies of analogical understanding and customisation of specifications are reported with implications for design of support tools during these activities.

Chapter 4

4: Empirical Studies of Software Engineering Behaviour

Chapter 1 proposed the need for tool support to assist software engineers to understand, select and customise analogical specifications. Empirical studies are needed to prove the analogical reuse paradigm and inform the process of analogical specification reuse. Existing empirical studies of analogical problem solving indicate that people are poor analogical reasoners unless assisted by reasoning or memory aids (Gick & Holyoak 1983). Little is understood of how software engineers reuse specifications or reason during requirements engineering, so empirical studies are needed to inform support tool design. This chapter reports four empirical studies of software engineers' behaviour during requirements engineering and analogical specification reuse, then summarises these findings to inform the design of tool support during the comprehension and customisation of specifications.

Experimental studies are needed to evaluate the analogical reuse paradigm, determine inputs to the specification retrieval mechanism and examine how reusable specifications are and can be reused most effectively. To this end, four empirical studies are reported, each of which informs the design of tool support:

- a first study investigated how inexperienced software engineers analysed a complex software engineering problem and developed a high-level system specification to solve that problem. This study was intended to inform design of the fact acquisition dialogue which precedes specification retrieval. In particular software engineers' analytic and reasoning processes were investigated and a preliminary cognitive task model of the requirements engineering task was developed;
- a second study investigated the experimental hypothesis that analogical specification reuse improves the analytic performance of inexperienced software engineers. The experiment investigated the effect on analytic performance of reusing analogically-matched templates and specifications. The null hypothesis was rejected, indicating that analogical specification reuse does help inexperienced software engineers to specify new domains. However, analogical specification reuse was error-prone, indicating the need to assist software engineers during specification understanding and customisation;
- the third study examined analogical specification reuse by investigating how

inexperienced software engineers' understood and transferred an analogical specification to fit a new problem. In particular, software engineers' analytic strategies and misconceptions about the analogy were modelled, resulting in cognitive task and reasoning models of analogical reuse and a mental model of analogical comprehension. These models informed strategies for more effective reuse practice and design of a diagnostic component intended to determine software engineers' analogical errors and misconceptions;

- the final study investigated successful reuse by expert software engineers in the same scenario as the previous study to derive expert cognitive task and reasoning models of specification reuse and a mental model of analogical comprehension. These models informed design of tool support by indicating effective strategies for reuse as well as the cognitive limitations of analogical understanding during reuse.

Each empirical study is reported in detail then their conclusions are applied to the design of the reuse advisor. This chapter ends with an outline tool specification which is expanded throughout chapter 5.

4.1 Study 1: Analytic Behaviour of Inexperienced Software Engineers

This study investigated how inexperienced software engineers analysed a complex software engineering problem and developed a high-level system specification to solve that problem. Reasoning topics were also investigated in this study to examine software engineers' ability to identify critical problem features from an ambiguous and incomplete problem statement (see Appendix B). The study is reported more fully in Sutcliffe & Maiden (1992). It contrasts with previous studies of systems analysis and high-level software design which only identified correlations between frequencies of mental behaviours and expertise exhibited by experienced software engineers (e.g. Vitalari & Dickson 1983) or higher-level social, organisational and experience-based factors linked to effective software development (e.g. Curtis et al. 1988, Rosson et al. 1988).

There has been little study of the cognitive processes underlying requirements engineering, although better planning, more effective gathering of domain information, better formation of structured diagrams of the problem domain and more critical testing of hypotheses have been suggested as qualities which differentiate expert from novice software engineers (Vitalari 1981, Vitalari & Dickson 1983, Fickas et al. 1988, Guindon & Curtis 1988, Guindon 1990). Furthermore, experts appear to use better heuristics and retrieve richer knowledge structures from memory (Guindon & Curtis 1988). More extensive psychological studies of program designers have demonstrated that novices have fewer preformed memory schemas and that novices tend to focus on the surface aspects of the problem (i.e. lexical/syntactic features of the programming language) rather than the semantic level of the problem itself (e.g. Jefferies et al. 1981, Adelson 1984, McKeithen et al. 1985, Holt et al. 1987, Koulek et al. 1989). Studies of program debugging suggested novices fail to scope problems, resorting to a strategy of bug isolation and repair (Nanja & Cook 1987), whereas expert strategies are directed towards building multiple domain models (Pennington 1987).

Requirements engineering involves different and more demanding skills than programming. The software engineer has to acquire information and build a model of the domain before design can proceed. Analysis, acquisition and comprehension of domain knowledge is a challenging task only partially supported by structured analysis techniques. Currently these approaches are limited to providing procedural guidelines for software engineers and diagram notations to represent problem domains. Although certain mental qualities (e.g. poor gathering of domain information) are purported to result in poor analytic performance, no thorough investigation of novice software engineer's reasoning and factors underlying failure has been undertaken. The objective of this study was to investigate cognitive factors underlying their performance and build a cognitive task model of that process to inform design of the problem identifier and specification advisor.

4.1.1 Method

Protocol analysis was used to investigate problem-solving behaviour of 17 novice software engineers (14 Male & 3 Female MSc students in Business Systems Analysis, with a maximum of 6 months structured analysis experience). Three pilot subjects undertook the analytic task beforehand, to refine the problem and the experimental procedure. During the experiment 3 subjects were unable to attend, whilst a fourth failed to verbalise sufficiently and was discarded. Data from the remaining 13 subjects (11M, 2F) provide the basis for the results discussed in this section.

Subjects were asked to develop a specification for a delivery scheduling system. All subjects had background domain and method knowledge necessary to develop a specification. They knew the delivery scheduling problem through experience on a case study, during which they used the Structured Systems Analysis (SSA) method and its main representation technique, data flow diagrams (DFDs). These techniques had been recently taught and practised as a part of the subjects' MSc curriculum. Relevant subject experience prior to undertaking the MSc course varied; 5 subjects had no previous exposure to developing computer systems, whilst 7 subjects had some exposure to structured development methodologies or flowcharting techniques.

4.1.1.1 Experimental Material

All subjects were given a 400-word narrative describing the problems of a manual delivery scheduling system (see Appendix B). The objectives for a computerised delivery system were also outlined, together with an example report describing the required delivery schedule. Subjects had no access to other material during the protocols.

4.1.1.2 Experimental Design

Subjects were requested to think aloud and their verbal protocols were recorded on audio tape. Beforehand all subjects were given the chance to practice thinking aloud whilst solving a simple puzzle. During the protocols subjects were advised to take their time when verbalising, and not be afraid of verbalising too much, following the practice of Ericsson & Simon (1980, 1984). The experimenter also recorded important bouts of physical behaviour such as reading the problem narrative or drawing a structured diagram. Instructions for subjects were read by the experimenter. Each subject was requested to develop a specification of a computerised system using data flow diagrams.

Subjects were given 35 minutes to develop a specification as the pilot studies with 3 subjects indicated this was sufficient time to complete the task. All subjects were informed of this time limit before beginning the task, and were expected to complete a solution by the end of it. All subjects were halted after 35 minutes. While each subject performed the task a protocol was recorded. Upon completion of the task a 10 minute retrospective protocol elicited further details of reasoning strategy and behaviour. Retrospective questioning was driven by a checklist of different behaviours which were expected to occur during the task (e.g. "Why did you model the current system before the required system ?"). Care was taken not to prejudice the retrospective protocol, so the experimenter only asked open-ended questions, following Ericsson & Simon's practice. Previous computing experience of each subject was obtained from the post-test questionnaire and scores for completeness/accuracy of the solution (DFDs and lists of requirements reported by the subjects against the list provided by expert judges) were provided by an expert judge. These scores are shown in Table 4.1. Experience ranged from subjects D, E, O, R and S who had no experience with computer systems development to subject K who had a BSc degree in mathematics for business (including final-year project experience of SSA techniques) and 16 months programming experience. Average experience of the subjects was approximately 6 months programming experience but only 6 weeks training in SSA techniques prior to the MSc course.

4.1.1.3 Analysis

Protocol transcripts were analysed by matching mental behaviours to speech segments, usually sentences and incomplete utterances (see Ericsson & Simon 1984 for further details of the method). Six major categories were used, divided into mental and non-mental behaviours.

Mental behaviours:

Recognise goal-	statement of high level functional requirement (e.g. "..And there is a need to improve the delivery system");
Assertions-	verbalisation of a belief or statement of facts about the problem domain directly attributable to the problem narrative (e.g. "Lorries leave the depot half empty");
Reasoning-	verbalisation of the creation, development and testing of hypotheses about the problem and its proposed solution (e.g. "..he also wants... to know about urgent orders, so again, that should be marked in some way, he will be able to pick it up straight away..");
Planning-	a meta-level of control over the analytic process. Two types of plan were distinguished by their dependence on method knowledge. A SSA method plan is "I'll list the inputs, outputs and sources, then draw a logical new DFD". General plans are, "I'll read the problem once, then construct a specification".

Reasoning was distinguished from assertions by the degree of inference applied and concurrent non-mental behaviour (e.g. reading). Recognition of a goal implied an understanding of the required functionality of the system. Planning behaviour differs from goal recognition in that it is domain-independent: goals state what the system must achieve whilst plans state how the subject develops a specification to meet those system goals.

Non-mental behaviours:

Information acquisition-	searching for and retrieval of data in the problem text or elsewhere (e.g. reading the problem text);
Model recording -	physical construction of the system specification recorded as a DFD.

Figure 4.1 and Appendix B present sample protocols identifying mental and non-mental behaviours. Ericsson & Simon (1984) have suggested that protocol analysis is a useful technique for eliciting sequential models of human problem solving. The consistent verbalisation of our subjects suggests that their reports were generally representative of the underlying mental behaviour, although this cannot be guaranteed. Retrospective questioning was also used to elicit mental behaviour which may not have been verbalised by the subject during the analytic task.

4.1.1.4 Protocol Categorisation

Protocol categorisation was validated through cross-marking by two independent observers. Each observer allocated a behavioural category to each utterance in 5 randomly-selected protocols. Categories were allocated to over 99% of all identified speech segments. Initial inter-observer agreement was 79% of the utterances categorised by both observers. Resulting differences were attributable to identifiable discrepancies between the observers. These differences between protocol categorisations were discussed and where necessary changes were agreed and reconciled by both observers.

4.1.1.5 Solution Completeness

Completeness scores were allocated to each subject's solution specification to represent their success or otherwise in solving the problem. Incompleteness was identified by Meyer (1985) to be a major *sin* of most requirements specifications. Completeness scores were considered an appropriate measure of analytic success over other *sins* for two reasons. First, subjects were required to identify the scope and major functional requirements of the delivery scheduling system, so one important measure of analytic success was their ability to recognise problem boundaries and major system components, i.e. processes, and entities. Indeed, pilot studies revealed these measures to be an important determinant of the quality of pilot subjects' solutions. Second, quantitative measurement of alternative requirements 'sins' such as *contradiction* and *wishful thinking* was difficult and thus omitted from this analysis.

Dev hyp	*So we've got external, the external agent;*
Ext hyp	*first one is our customers.*
Draw	
Ext hyp	*Makes the order,*
Test hyp	*and er,*

Dev hyp	the process order department which will, er,
Draw	
Ext hyp	*will see if the order is valid,*
Test hyp	*and er,*
Ext hyp	*and about the credit worthiness of the customer,*
Ext hyp	*will proceed the order,*
Ext hyp	*which will be stored in a file, data store, order file*
Draw	
Ext hyp	*and each order, the order will be some kind of a record,*
Ext hyp	*which will have information about er, the customer name,*
Ext hyp	*um, you can have only the account number of the customer,*
Test hyp	*which will, who can,*
Ext hyp	*it will the main address of the customer,*
Ext hyp	*the item he needs,*
Test hyp	*and er,*
Ext hyp	*the day that he needs, he needs the items.*
Gather Info	
Ext hyp	*Pink scheduling note,*
Test hyp	*okay.*

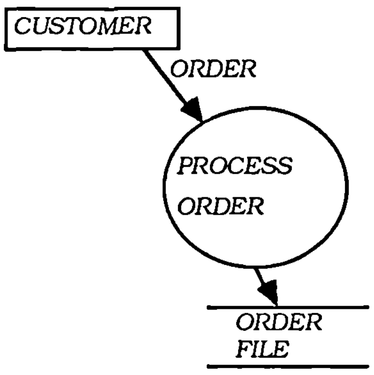


Figure 4.1 - example protocol from subject J, including part of data flow diagram developed during this segment

A marking scheme was created from solutions developed by 3 experienced software engineers with considerable knowledge of the delivery scheduling problem and SSA techniques and who were considered very capable for the task. The scheme contained a list of components to be included in a specification, and focused on semantic features of subjects' solutions rather than on the syntax of the data flow diagramming representation. These components included the processes, data stores, system inputs, outputs, functional and non-functional requirements of the system. Components were included on the marking scheme list if they were included in any one of the 3 expert solutions. Subjects received a score if a component was included in the resulting data flow diagram or if the subject verbally stated that the component was to be included in the system. Each completeness score was representative of the subject's ability to recognise the scope of the problem and the major functional requirements of the system. A data flow diagram representing a composite solution developed by the experienced software engineers is given in Appendix B.

4.1.2 Results

Subjects' overall performance was poor, averaging only 11.4% of the 'expert' score (see Table 4.1). Most subjects' low scores possibly reflected both their inexperience and the limited time available to complete the problem (35 minutes was allowed based on expert completion times of about 20 minutes). This suggested that they found the analytic task difficult. One subject (D) drew nothing, eight subjects drew context diagrams, and ten subjects drew data flow diagrams, all of which were incomplete. The most exhaustive attempt was a set of five diagrams from subject E labelled as current logical and new logical systems. Two sets of diagrams were reworked by the subjects and many were used as informal recording devices for the development of drawn specifications. Potential reasons for poor analytic performance were investigated.

Frequencies of mental behaviours were counted for 5 minute segments. This was considered to be a convenient time period to yield suitable frequencies for analysis. Information gathering, assertions and planning decreased with time and reasoning increased in the later stages of the protocols, see Figure 4.2. The rise in planning behaviour was caused by two subjects (G, M) who ended their protocols by stating how they would have approached design of the delivery scheduling system. The decline in information gathering before the increase in goal recognition behaviour suggested that subjects required time to assimilate the problem before system requirements could be identified. Individual differences were apparent in total occurrences of each particular behaviour (see Table 4.2), notably subject H who had low levels of planning, goal recognition and assertions.

Subject	%age completeness score achieved by subjects	%age completeness score achieved for DFD only	Analysis Experience	Programming Experience
D	7.8	0	nil	nil
E	12.4	7.2	nil	nil
G	6.9	7.2	6 wks taught	15 mths taught
H	4.7	4.3	1 wk taught	12 mths prac.
I	13.2	10.1	1 yr flowchrts	nil
J	17.1	24.6	3 mths taught	18 mths taught
K	43.4	7.2	5 mths project	16 mths taught
M	9.3	4.3	6 mths taught	6 mths prac.
N	11.6	2.9	3 wks taught	12 mths prac.
O	8.5	10.1	nil	nil
P	10.1	2.9	taught only	3 mths prac.
R	1	1.4	nil	nil
S	2	4.3	nil	nil
Average	11.4	7.1		
Range	10.01 - 43.4	0 - 24.7		

Table 4.1 - completeness scores, as %ages of the entire expert solution and of the expert DFD, and subjects previous computing experience (all subjects had an additional 6weeks training with SSA techniques)

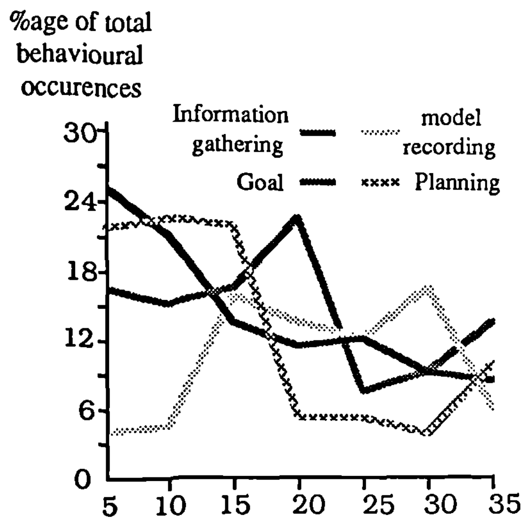


Figure 4.2(a) - frequencies of occurrence of all behaviours in 5-minute segments of protocols, for all subjects

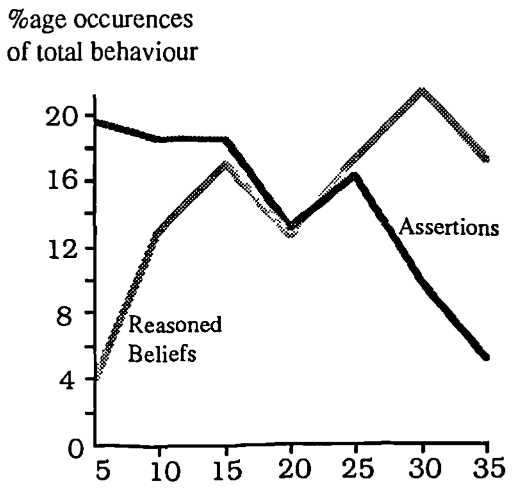


Figure 4.2(b) - frequencies of occurrence of Assertion and Reasoning behaviour, analysed in 5-minute segments for all subjects

subject	information-gathering behaviour, in mins & secs	number of goals/ requirements identified	number of assertions made	number of reasoning inferences made	number of plan details made
D	20 mns 11secs	7	21	4	17
E	3 mns 20 secs	7	40	18	13
G	5 mns 18 secs	8	33	23	11
H	17mns 46 secs	0	0	21	0
I	12 mns 3 secs	8	8	21	6
J	12 mns 8 secs	2	14	35	0
K	8 mns 37 secs	9	17	60	11
M	10 mns 35 secs	5	28	13	14
N	12 mns 36 secs	6	24	14	8
O	11 mins	0	31	6	15
P	11 mns 40 secs	10	19	32	0
R	16 mns 24 secs	3	17	5	4
S	18 mns 17 secs	2	20	22	0
average	12 mn 18.1 scs	5.2	20.9	21	8.46
range	3 mns 20 secs - 20 mns 11 secs	0 - 10	0 - 40	5 - 60	0 - 15

Table 4.2 - total occurences of behaviour by subject. All values are the number of occurences of a behaviour except for information gathering which is given in seconds duration.

Sequential dependencies between behaviours for all subjects were analysed by casting the behaviours in a transition matrix (occurrences of A following B and vice versa) then constructing a network model of the temporal relationships between behaviours. The results for all subjects are shown in Figure 4.3. The strongest associations were between gather information, assertions, and recording structured diagrams. These associations probably represent the analytic side of understanding the problem domain as software engineers acquire information, understand it, and organise facts in data flow diagrams. In comparison, there was little association between reasoning and planning behaviours, suggesting that subjects exhibited little systematic reasoning about the domain.

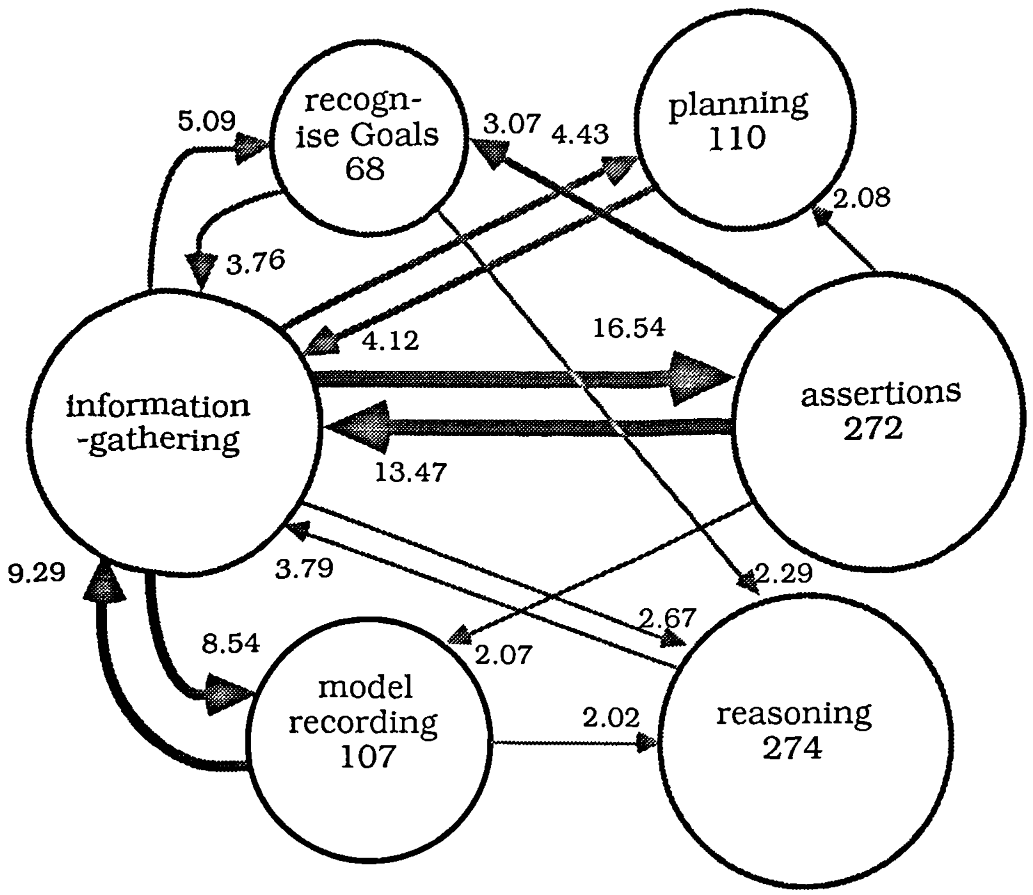


Figure 4.3 - averaged sequential dependencies (>2%) between behavioural categories for all subjects. Total occurrences of each behaviour are shown in the circles and the circle size has been scaled accordingly

No correlation was found between experience scores and solution completion (Spearman Rank Order Coefficient). Similarly no relationship was found between totals of different types of behaviour over time (two-way analysis of variance, time analysed as 5-minute segments). Correlations between totals of each type of behaviour and individual's experience and solution completion were also non-significant apart for reasoning with completeness (Spearman Rank Order Coefficient, $p \leq 0.05$). At the fairly crude level of totals of behaviour, this suggested that reasoning ability may be linked to improved analytic performance, although only subject (K) performed competently at the task. To understand this link further, models of analytic reasoning were derived from protocol transcripts, as described in the next section.

4.1.2.1 Models of Reasoning Behaviour

Reasoning behaviour was investigated in terms of the development of hypotheses following the generally accepted development-and-test model (e.g. Akin 1986). The life history of each hypothesis was traced by its thematic content until eventual rejection or resolution. Hypothesis reasoning behaviour was categorised as generate, develop, test, confirm, modify and discard (see Figure 4.4). All subjects showed well-developed networks linking Generate-Develop-Test Discard or Modify, apart from subjects R and S who showed only weak associations between generate, develop and discard. It was noticeable that these subjects also had low completeness scores (1% and 2% respectively) and that weaker subjects (D, G, H, R and S) used reasoning strategies less effectively, i.e. by testing only general hypotheses and applying poor tests which resulted in vague conclusions. Stronger subjects generated domain scenarios to evaluate hypotheses. Many subjects immediately discarded hypotheses once they had they been generated, suggesting that in general, subjects were poor reasoners unable to develop and test hypotheses about the domain.

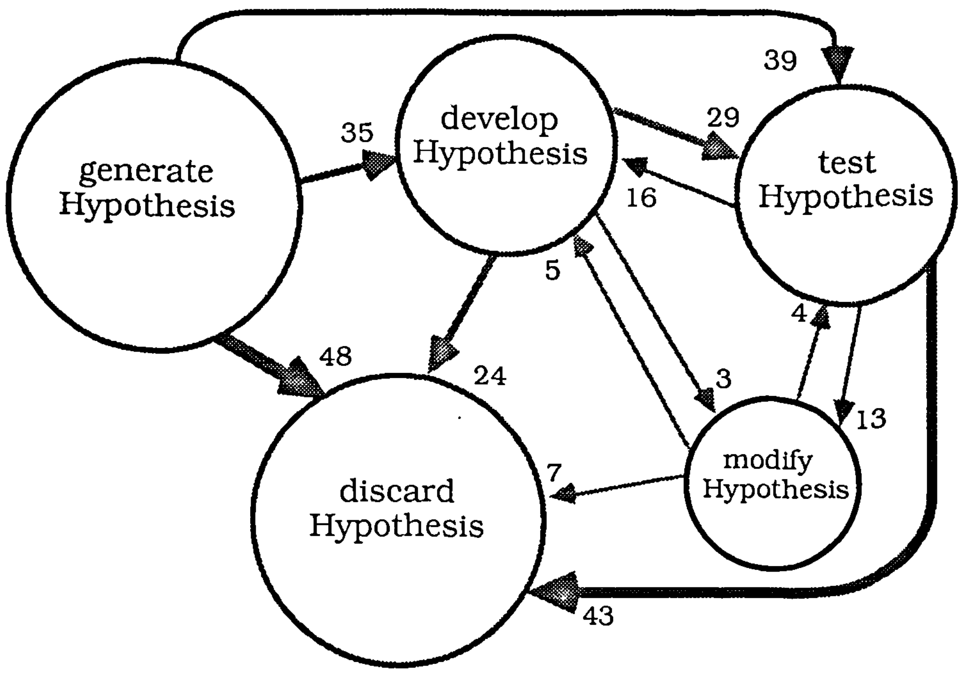


Figure 4.4 - network model of hypothesis life histories for all subjects: figures on the arcs represent the total number of transitions which occurred, and circle sizes have been scaled accordingly

All subjects showed little tendency to reconsider old hypotheses, as 81% of all hypothesis topics were never retrieved for further development. Of those that were, development of 11% was triggered by reference to facts in the problem statement, while 9% were linked to development of system requirements. Eighty five percent of the hypotheses followed information gathering from the narrative hence most reasoning was linked to domain knowledge extracted from the problem statement and guided by the contents of the problem narrative. This indicated that subjects tended only to reason about topics closely related to existing system problems.

4.1.2.2 Reasoning and Structured Diagrams

A major recommendation for use of structured methods is the formalisation of domain knowledge using structured notations such as data flow diagrams, entity relationship models, etc (e.g. De Marco 1978). These diagrams can be used to describe new target domains, so this study examined how subjects reasoned with these structured model notations. Modelling was analysed by examining reasoning behaviour which occurred concurrently with diagram formation. Model-based reasoning involved the generation and development of hypotheses linked by a single thematic strand related to components added to the diagram.

Six subjects (E, G, I, J, K, N) exhibited model-based reasoning while another five subjects partially did so (H, P, R, S, O). On the other hand subject M, along with the subject D who did not develop a DFD reasoned with a series of disjoint, unrelated hypotheses and consequently were judged not to have shown model-based reasoning. Subjects who constructed full diagrams produced more complete solutions (average score 17.4% modellers) than subjects who exhibited partial or no diagramming behaviour (average score 5.26% for partial modellers, 8.5% for non-modellers). Four subjects who exhibited model-based reasoning were among those with more experience of SSA techniques. In addition 5 of the 6 model-based reasoners were among those who produced the best data flow diagrams, suggesting either that building and reasoning with structured diagrams may improve analytic performance, or that greater experience with structured analytic notations may promote model-based reasoning. Construction of data flow diagrams may have been influenced by the analytic strategies adopted by subjects, hence these strategies were investigated.

4.1.2.3 Planning

Control over the analytic process was manifest as planning behaviour. Seventy-three plans were verbalised, six of which summarised reasoning which had already taken place. Of the remaining 67 plans, 52 were carried out during the protocol while 15 were abandoned. The majority (97%) of implemented plans were short term describing the next sub problem task and were executed within 5 minutes. Retrospective questioning about overall plans and strategies also revealed that all subjects had difficulty verbalising coherent plans, suggesting either the absence of long term planning or the unconscious nature of such plans.

Forty-one percent of all plans involved method steps and procedures (specified by the

SSA methodology, e.g. "Model the current system before modelling the required system"), while the remaining 59% employed general knowledge. Subjects with weaker completeness scores exhibited more planning (see Tables 4.1 and 4.2) and a greater reliance on method procedures while other subjects exhibited less planning and largely ignored structured analysis guidelines. This indicated that method procedures may have been more actively employed by the most inexperienced and least successful subjects, although as Ericsson & Simon (1984) suggested, subjects more experienced with these techniques may fail to verbalise frequently-applied knowledge.

These quantitative analyses of subjects' behaviour revealed some determinants of good and poor analytic performance. The empirical findings reported thus far provide clues to a general cognitive task model based on subjects' reasoning behaviour. However, one more specific objective of this study was to investigate the semantic content of their specifications to suggest likely inputs to the specification retrieval mechanism. This objective was met by investigating the semantic content and errors of subjects' completed specifications.

4.1.2.4 Solution Quality and Errors

Subjects' protocols and completed data flow diagrams were analysed for both errors and their underlying misconceptions. Analysis of detailed errors was hampered by subjects' inability to develop even basic structured diagrams. Solutions were incomplete rather than erroneous. Subjects were more successful at recognising system goals and inputs (23% and 20.5% of the expert solution respectively), whilst there was poorer recognition of system data stores (11.9%), processes (8.8%) and outputs (5.8%). Inspection of protocols suggested subjects were unable to recognise or infer those processes and data store accesses included in the expert software engineers' solutions.

The semantic content of subjects' solutions was examined to determine the existence of critical facts about the scheduling problem as stated by the meta-schema of knowledge types in chapter 3. Fundamentally, the delivery scheduling system consists of two object allocation, or constraint satisfaction, domains. The first problem must allocate daily deliveries to lorry space, and the second problem must allocate lorries to particular routes to maximise deliveries to customers. Both constraint satisfaction functions have requirements which must be met by resources, so they are analogical to the lift routing domain shown in Appendix A. Subjects' solutions were examined for the inclusion of key domain features as described in Table 4.3.

key facts about the domain	subjects												
	D	E	G	H	I	J	K	M	N	O	P	R	S
fn: allocate daily orders			√				√			√			
fn: allocate orders to lorries													
object: order/delivery				√		√				√			
object: lorry										√			
object: lorry space													
object: route						√							
condition: suffic. lorry space													
condition: selected route is ok.													
delivery type													
lorry space type													
lorry type													
route type													
reqt: meet all deliverables													
reqt: meet all route needs													
function of 1st alloction													
function of 2nd alloction													
label: object allocation													
label: constraint satisfaction													
label: requirement matching													
totals	0	0	1	1	0	2	1	0	0	3	0	0	0

Table 4.3 - verbalised or drawn key domain facts by subject: a tick indicates recognition of the fact by the subject during concurrent protocols. Facts are derived from the logical model of software engineering analogies defined in chapter 3. Generic terms labelling the domain type were also included in this analysis

However, subjects were unable to incorporate these concepts into their specifications, suggesting that they had a poor understanding of critical domain facts. Rather subjects tended to model the current system and focus on high-level system inputs, outputs and processes more related to sales order processing than delivery scheduling, possibly because they had previously analysed and specified the sales order processing system for the same case study organisation. Eight subjects did not include any critical components in their solutions while at best one subject included three components. In addition, subject J recognised the need to sort requirements by priority to maximise subsequent allocation. However, we should not be surprised by these findings, since subjects had little or no previous experience of similar domains. These results have important implications for the specification retrieval mechanism. Subjects' solutions revealed their inability to determine most key domain using diagrammatic representations, although several subjects were able to model key state transitions. Furthermore, many subjects modelled the correct state transitions for the sales order processing domain, indicating that the proposed logical model of domain abstraction may be an effective mechanism for modelling and matching domain descriptions, although extensive tool support appears necessary. In the light of these findings, subjects' analytic strategies were investigated more closely to determine potential reasons for this failure to recognise key domain facts.

4.1.2.5 Analytic strategies

This analysis was completed by examining subjects' planning and reasoning behaviours in combination with the types of knowledge employed. Investigation of analytic strategies was primarily qualitative rather than quantitative. Pilot data suggested that subjects start analysis by investigating the scope of the domain and structuring the problem space. Indeed, most structured methods advise boundary definition as an early exercise in analysis; consequently this study also focussed on whether subjects effectively structured the problem space before moving onto more detailed analysis. The concurrent and retrospective protocols were examined for the following qualities of reasoning strategies:

- was a logical model of the current system constructed before design issues were introduced, following the usual paradigm of separating analysis from design ?
- were physical issues (implementation) brought into the analysis and design, contrary to the advice of most methods ?
- were method heuristics, procedures or steps used explicitly (e.g. top down functional decomposition) ?
- did subjects scope the size of the problem and establish the problem boundaries before proceeding to analytic detail ?

All subjects attempted to scope the problem but eight (D, G, H, M, N, O, R, S) were not effective in doing so. One of the eight poor scopers did not determine the boundaries of the domain, while the other seven became embroiled in detail before determining the problem boundary. Five poor scopers modelled the current system and rigidly employed SSA method heuristics (e.g. 'identify all external agents' and 'model the current system using a data flow diagram') to guide domain scoping. These heuristics appeared to quickly focus subjects' attention on domain details, hence successful domain scoping may have been impeded. A sixth poor scoper (N) only modelled the inputs to the current system before attempting to design a solution. Subjects who had difficulty in scoping the domain also had low completeness scores. Findings suggest that literal reliance upon method heuristics may have impeded proper scoping of the domain.

Nine subjects (E, G, H, I, J, K, N, O, R) explicitly reported using heuristics and method steps of structured analysis (e.g. determining external entities, system inputs outputs, context diagram, etc) but only three (E, H, J) used the method consistently. Ten subjects used a top-down approach without verbalising it, though it was not applied consistently. Subjects tended to "divide and conquer", analysing one area in depth before moving onto another.

Only the least successful subjects exhibited much overt usage of structured analysis methods while developing data flow diagrams. Furthermore, over half of the subjects did not follow the dictates of developing a current logical model before developing a new logical design. Only three subjects (G, E, K) clearly separated analysis from design. On the other hand three subjects (J, M, O) disobeyed the structured approach by mixing physical design details (e.g. files, sorts, implementation hardware) with analysis. Two of these subjects (J, M) had the most programming experience and did not suffer from inclusion of implementation features. It is notable that subject J constructed the most complete data flow diagram using several sort procedures as the hub of his model. Generally subjects did not exhibit a well-ordered approach to systems analysis and mixed design of the problem solution with analysis and problem description.

4.1.3 Conclusions: Qualities of Effective Requirements Engineering Behaviour

Findings indicated that good performance by inexperienced software engineers cannot be predicted by a single factor, although rigorous testing of hypotheses and reasoning with structured diagrams do give a firm indication of performance. Three subjects (J, K, E) who combined opportunistic generate-and-test strategies also appeared to be more effective. Similarly no one factor underlies poor performance. Poor domain scoping and lack of hypothesis testing appear to be important. Performance in any one individual is probably the result of a combination of these factors, as demonstrated by case studies of two successful subjects and one poor subject given in Appendix B. The possible reasons for poor and good performance by subjects are summarised in Table 4.4.

reasons for good performance	subjects												
	D	E	G	H	I	J	K	M	N	O	P	R	S
good problem scoping		√			√	√	√				√		
use of structured diagrams		√	√		√	√	√		√				
critical testing					√	√	√		√		√		

Table 4.4 - summary of possible reasons for good performance by subjects

4.1.4 Discussion

This study investigated inexperienced software engineers during the analysis and specification of a complex software engineering problem. Conclusions from this study were mixed. Several factors linked to effective analytic behaviour were identified, however software engineers were unable to identify many key facts about the delivery scheduling domain defined in chapter 3, and their final specifications were incomplete. One obvious reason for this failure was their lack of relevant domain knowledge due to no exposure to similar scheduling problems. Instead, subjects may have exhibited a recency effect and focused on system features related to a recent problem solving experience with the sales order processing function of the same problem.

Findings suggest several factors linked to poor analytic performance. Changes in totals of behaviour over time during the protocols combined with analysis of the protocol transcripts indicate that analysis may have two distinct phases; an initial scoping of the problem and structuring the problem space, followed by more detailed reasoning about the problem. Information acquisition was concentrated in the first phase, and novice software engineers in this study spent an average 33.5 % of the whole protocol gathering information, which is greater than Vitalari's (1981) figure of 25% information acquisition time for experienced, but poor performing, software engineers. However, these software engineers were unable to scope and structure the problem, suggesting it to be one determinant of poor performance which software tools must help software engineers to overcome.

Poor performance may be attributed to a variety of other factors, from ineffective testing of hypotheses, supporting Jefferies et al.'s (1981) and Adelson & Soloway's (1985) findings, to ineffective modelling. This contrasts with the essentially unitary causalities of poor performance reported by previous studies (e.g. Vitalari & Dickson 1983, Fickas et al. 1988). This study may have uncovered natural human limitations in information processing ascribable to poor mental model formation or motivational factors.

Software engineers' success in this study appeared to be able to reason critically and effectively with structured diagrams. Model-based reasoning may be indicative of mental model formation in the sense of Gentner (1983). As mental models involve holding many linked facts in memory they may be regarded as a large composite knowledge structure. Experts in many domains are distinguished by their ability to retrieve and use large knowledge structures (Schank 1982) and mental model formation has been postulated as an important factor in human mental reasoning ability (Johnson Laird 1983, Gentner & Stevens 1983, Adelson & Soloway 1985). It is therefore not surprising that model based reasoning and performance appear linked. Representation of related information in tractable notations may be one of the more important improvements which structured methods have made in supporting the analytic reasoning process.

Contrary to the advice given in software engineering methods, novice software engineers did not appear to separate analysis from design, and mixed physical implementation detail with logical specification. The former strategy incurred no performance penalty, indicating that mixed analytic and synthetic-design based reasoning may be more natural than the enforced separation advised by methods, a finding which agrees with Lawson's (1980) and Akin's (1986) studies on architectural domains. Likewise, concentration on physical detail did not appear to incur performance penalty, suggesting even the methodological sin of mixing logical and physical development phases may not be disadvantageous. A similar diversity in opportunistic reasoning strategies have been reported in professional domains (e.g. Akin 1986) and everyday problem-solving tasks (Hayes-Roth & Hayes-Roth 1979) as well as software development (Guindon & Curtis 1988, Guindon 1990, Visser & Hoc 1990).

To conclude, this study led to a tentative, empirically derived cognitive task model of requirements engineering with implications for supporting individual software engineers during the investigation and specification of an unfamiliar problem. Of course, much software development occurs between distributed peer groups or during meetings with end-users (e.g. Curtis et al. 1988, Curtis & Walz 1990), so the proposed model has limited implications. However, it does provide important clues about the fact acquisition dialogue needed to feed the specification retrieval mechanism. An outline of this support is described in the next section.

4.1.5 Implications for Support Tools

The most important implication from this study is the need for semi-automated tool support during key fact acquisition. In particular such tools must provide domain knowledge necessary during fact acquisition to facilitate problem understanding, scoping and identification of key domain facts (Curtis et al. 1988). One effective way of providing this domain knowledge may be to present software engineers with key domain abstractions early in the fact acquisition dialogue to assist problem scoping, structuring and evaluation. Understanding these abstractions may be difficult, so concrete examples can be provided to aid comprehension (Gick & Holyoak 1983). A similar, example-based approach was implemented in the CODEFINDER system (Fischer et al. 1991a).

Alternative notations for representing key knowledge types are needed to supplement more traditional structured analytic notations which failed to capture key domain facts in this study. Such notations must represent key state transitions and object structures. Indeed, defining software engineering domains in terms of key state transitions may be cognitively plausible since software engineers exhibited a tendency to model key system functions, inputs and outputs equivalent to these transitions. Tool support will be necessary to assist completion of these domain descriptions because software engineers found this task difficult.

This first study revealed other factors which were linked to good and poor requirements definition. On the positive side, support tools should encourage software engineers to reason effectively with structured and informal diagrams, thus prompting more effective mental model formation. However, the need for alternative notations to support structured diagrams formulated by software engineers should not be underestimated. Software engineers must also scope and test specifications more effectively, and one solution may be to provide explicit, prescriptive guidelines to encourage better scoping and testing. Furthermore, support tools should not encourage use of structured analysis heuristics during individual bouts of analysis but promote a more opportunistic mode (e.g. Guindon 1990) of reasoning during problem exploration and mental model formulation. The implications from this study for the design of support tools are described in more detail at the end of chapter 4 and throughout chapter 5.

4.2 Study 2: An Experimental Study of Specification Reuse

The effectiveness of specification reuse on analytic performance was investigated experimentally in a second study. A controlled experiment examined the effect of providing analogically-matched abstract and concrete specifications on the analytic behaviour of inexperienced software engineers. Minimal support for specification reusers was provided so the experiment investigated valid analogical reuse in a realistic CASE tool scenario. Software engineers received the reusable specification and a brief narrative describing the system's functionality without any explanation of analogical mappings or key domain abstractions.

This study also investigated software engineers' ability to comprehend and transfer analogical specifications to inform design of tool support. Previous empirical studies of programming behaviour suggested that effective specification reuse would be difficult. Program debugging tasks revealed that, in practice, even expert programmers required considerable time and mental effort to understand and modify unfamiliar programs (Pennington 1987), whilst novices often fail to achieve any successful modifications (Holt at al. 1987). Indeed novices tend to adopt strategies which hinder understanding (Nanja & Cook 1987). The inexperienced software engineers in this study lacked experience and knowledge of similar domain types, so specification understanding was likely to be difficult and error-prone. In addition, as Sheil suggested (cited in Sein 1988), inexperienced software engineers are unlikely to have many domain analogies to draw on when constructing new mental models. However, given novices' poor performances in the previous study, it was hypothesised that analogical specifications would still promote reuse and improve specification completeness and accuracy.

Finally, this study investigated the effectiveness of reusing abstract and concrete specifications. As discussed in chapter 1, reuse of generic templates or cliches has received considerable attention in the literature (e.g. Harandi & Young 1985, Fugini et al. 1991). However, software engineering authors rarely evaluate the usability of their products, consequently little evidence exists on how abstract concepts may help systems development. In light of this the reuse of specifications presented in concrete and abstract forms was investigated to determine the relative effectiveness of generic templates and whether they can supplement reuse of concrete analogical specifications.

4.2.1 Method

Complete accounts of this experiment are given in Sutcliffe & Maiden (1990a & 1990b) while the experimental data is presented in Appendix C. Thirty (23 male, 7 female) subjects were full-time MSc students in Business Systems Analysis and Design. They had knowledge of several structured analysis and Jackson (JSD) techniques, and all but 6 had previous systems development experience. The subjects, whose age ranged from 21 to 36 years, volunteered their services, for which they received practice and supplementary tuition on JSD techniques.

A video camera recorded all written work, and verbal protocols were tape recorded. Subjects were asked to develop a JSD process structure diagram for a scheduling function allocating videos to hotels. The problem built upon target domain knowledge already acquired by subjects from a case study.

A between subjects, two conditions experiment was conducted with:

- a control group, where subjects were given the problem narrative alone;
- an abstract analogy group (AA), where subjects were provided with the problem narrative and an unfamiliar abstract JSD template of a general scheduling problem;
- a concrete analogy group (CA), where subjects were given the problem narrative and a JSD specification of a real but unfamiliar analogous production planning application.

Each group of 10 subjects was balanced with respect to subjects' experience.

In concurrent protocols, groups CA and AA subjects were requested to verbalise:

- similarities between the reusable specification and the problem; and
- how these similarities were used to solve the problem.

Subsequent retrospective analysis probed subject's general problem-solving strategies and their understanding of the analogy and the target problem. The main concept was the functional requirement to allocate a resource within certain constraints. This was manifest as scheduling a resource (or video copies in the target domain) within constraints such as time and hotel preference in the target domain. Retrospective protocols and a written questionnaire captured problem-solving and reuse strategies.

Subjects' solutions were scored for completeness and validity. Completeness was used as a measure of success for the same reasons as given in the first study while an error count was found to be effective in this study due to the well-defined nature of the task and the scope of the domain. Completeness was measured against a solution provided by an experienced JSD analyst. Subjects' solutions were scored for the correct number of actions in the diagram and for use of JSD design constructs (e.g. Backtracking). The validity of solutions was measured by the quantity of specific errors, determined by the extent to which the specification was incorrect in terms of domain knowledge and JSD syntax. Solutions were independently cross-marked by two experts, who agreed on scoring in 91% of all cases for completeness scores.

4.2.2 Results

Completeness scores for subjects are shown in Table 4.5. They indicate that the subjects who were given reusable specifications produced more complete solutions than the control group. This effect was significant for the abstract template (AA group) (T-test, using the approximating Z distribution for non-normal populations; $Z = -2.23$, $p \leq 0.05$); however, although the concrete analogy (CA group) showed better scores than the control group, this difference was non-significant. Control subjects made more errors than both experimental groups although these differences were also non-significant.

average % completeness and number of errors	control group	abstract (group AA)	concrete (group CA)
% completeness	24.4	41.1	32.8
average number of errors	11.8	10	10.1

Table 4.5 - average completeness (as % of ideal solution) and error scores for solutions developed by subject groups

Recognition of the analogy was evaluated by asking subjects whether they recognised three key analogical associations derived from the object allocation abstraction described in Appendix A:

- the functional transformation of allocation/scheduling;
- the concept of resources; and
- the requirements needing the resources.

All the AA subjects recognised at least one key association, 8 out of 10 CA subjects also recognised one association but none of the subjects recognised all three.

Three mappings which involved JSD method knowledge as well as domain knowledge were analysed in more detail. Subjects were asked whether they recognised and used three features and their solutions were checked for inclusion of the same. The three features of the allocation function were: integrity of the top-level sequence, an iteration of hotel-to-video allocations, and a backtracking selection for each allocation (see Table 4.6). Most of the control subjects failed to recognise these features whereas the experimental subjects performed significantly better, with the AA group having a higher overall score than the CA group (T-tests approximating Z distribution for non-normal populations; Z = -4.637 & -2.6.76 respectively, p≤0.05). This suggests that the transfer of structural knowledge about the target solution was effective, particularly with the abstract template.

solution structure	control subjects	group AA subjects	group CA subjects
top-level sequence	1	9	5
alloction iteration	4	8	8
backtrack condition	0	6	3
group totals	5	23	16

Table 4.6 - number of subjects who reused key structures in their solutions

Subjects' attitude, recorded by the post-test questionnaire, underlined the effectiveness of the abstract template. Subjects from the AA group rated the abstract analogy to be more helpful in developing a solution than did CA subjects for the concrete analogy.

The reasons for failure to use the analogy appear to be matters of motivation and comprehension of the analogical specification. Five group CA subjects failed to use the analogical material. Two of these 5 subjects retrospectively reported that they rejected the analogical specification since it contained too much information to be absorbed in the time allowed, whilst another 2 totally ignored the analogical material. The other CA subject was unable to reuse the scheduling function, even though the analogy was recognised. Two AA subjects also failed to reuse the abstract template because they misunderstood the functionality in the specification, although they did recognise the potential analogy with target problem. These findings indicate that the concrete analogy may be more difficult to assimilate than the abstract template.

Analytic strategies which appeared to lead to errors were identified in retrospective protocols, backed up by analysis of subjects' solutions. Three strategies were apparent:

- creation of unnecessary components in the target specification, apparently caused by the motivation of mapping all components across from the analogical specification (4 AA and 2 CA subjects);
- making false analogies, apparently caused by trying to link all structures in the specification with a structure in the target domain (1 CA subject);
- choice of the structure to map was based on its general familiarity (2 CA subjects).

Some subjects used more than one of the above erroneous strategies. It was noticeable that weak heuristics were also used to attempt to solve other, less-important aspects of the target problem not associated with the analogy.

4.2.2.1 Effect of Experience

No significant interaction was found between experience and solution completeness or errors with a two-way analysis of variance, see Table 4.7. Inexperienced subjects made proportionally fewer syntax errors, which appears to contradict their lack of experience (see Table 4.8). This result may be caused by a copying strategy from the analogical and template specifications which could also explain their higher rate of domain errors.

completeness or error score	subject experience	control group	abstract (group-AA)	concrete (group-CA)
%age completeness	considerable	27.8	63.9	50
	some	21.3	38.9	20.4
	none	30.6	25	52.8
average errors	considerable	8.5	4	6.5
	some	11.33	14.83	12.16
	none	16.5	1.5	7.5

Table 4.7 - average completeness (as % of possible total) and error scores by subject group andsubject experience

subject experience	percentage completeness	average errors	ratio of domain: syntax errors
considerable	45.3	5.5	1:1
some	26.9	12.5	1.2:1
none	36.1	9.83	2:1

Table 4.8 - average completeness (as %age of possible total), error scores by subject experience including domain: syntax error ratios for subject solutions

4.2.2.2 Analysis of the Use of Analogy

The quality of analogical reuse was rated in four bands according the completeness scores and reuse strategies reported by the subjects. In all cases the strength of assertions made retrospectively about analogical transfer agreed with the quality of subjects' solutions. Five group CA subjects and 8 group AA subjects, who had completeness scores of $\geq 7/18$ components, verbalised a clear model of the analogy and its association to the domain in retrospective protocols. In addition, poor usage was shown by another 2 CA subjects who misunderstood the concrete analogical specification, made false inferences about component details and mapped to an inappropriate JSD process structure. Finally two AA subjects partially used the abstract template and employed some of the analogy's components in their solutions. Results are given in Table 4.9.

quality of application	number of group-AA subjects	number of group-CA subjects
good	6	5
partial	2	0
poor	0	2
none	2	3

Table 4.9 - application of analogy knowledge by group AA & CA subjects

Successful reuse of the abstract template and concrete analogical specification was examined more closely. Good and partial reuse subjects were grouped as *successful reusers* while others were classified as *unsuccessful reusers* - see Table 4.10a. Although the completeness scores, predictably, were better for successful subjects, this effect was not present for errors. It suggested that although reuse may promote a more complete solution, accuracy of the result may not be improved. Furthermore, successful reuse of the concrete analogy resulted in more complete and valid specification than reuse of the abstract template, however, these differences in scores of successful subjects only neared significance - see Table 4.10b.

	no. of subjects	average % completeness scores	average error scores
successful subjects	13	49.2	9.23
unsuccessful subjects	7	14.3	11.57

Table 4.10a - average completeness and error scores for *successful* and *unsuccessful* subjects

	no. of subjects	average % completeness scores	average error scores
group-AA	8	47.2	11.125
group-CA	5	52.2	6.2

Table 4.10b - average completeness and error scores of *successful* subjects in AA and CA groups

The type of errors made by transferring knowledge from the reusable specification was investigated by looking at constraint checking, an important part of the scheduling function. Again successful CA subjects performed better than their AA counterparts. All 5 successful CA subjects modelled individual constraint checks and correctly used components in the concrete analogical specification. However, of the 8 successful AA subjects, four modelled the constraints in non-specific terms, (e.g., 'Check Constraints' component, or '1st', '2nd', 3rd', etc Constraints'), two only modelled 2/4 individual constraints, one subject specified incorrect constraints, and only one subject correctly modelled the constraints as required. This better performance of the CA subjects among the *successful reusers* may be caused by the extra mental effort required to understand the concrete analogy. The number and naming of the check constraints components by the AA subjects were closely related to the reusable specifications from which they were derived, suggesting they may have been copying the material rather than reasoning about it.

Specification copying in the sense of direct transfer and lexical tailoring of specification components, without reasoning, also accounted for many errors. Errors in eleven of the 13 successful subjects' solutions, combined with their retrospective reports, suggested a general failure to understand the analogy. One subject included the sort component within the allocation iteration while the remaining 10 subjects had errors related to the backtracking concept, from duplication of conditions and posits to use of conditional rather than backtrack symbols in the posit\admit components. Retrospective questioning also revealed that no subject understood the reusable specification to their satisfaction, although this did not inhibit reuse. For example, 9 subjects transferred the backtracking concept, although post-test questioning revealed only 3 of these subjects understood its meaning.

Among the *successful reusers*, only 6 developed solutions that supplemented the material derivable from the analogy. One CA and 3 AA subjects added minor components or structural features, while 2 group AA subjects expanded the abstract solution, retrospectively claiming that the abstract constraint checking component was insufficient. However, it was more common for subjects to omit components from the reusable specification, for example all but one subject omitted backtracking quits from their solutions. Retrospective probing suggested that such omissions may have been caused by failure to understand the role of the components in the abstract template and concrete analogical specification.

4.2.2.3 Other Findings

Analysis of the video tapes suggested successful reuse required considerable effort, since subjects who developed the most complete solutions spent 90% of the protocol session time attending to the reusable specification. There was a significant correlation between time spent analysing the reusable specification and subject completeness scores (Spearman Rank Order Coefficient r=0.657, p≤0.05). Most subjects took some time to recognise similarities between the two systems, suggesting that understanding the analogy may have been incremental.

Failure to understand the analogy led to mappings based on surface similarities between the problem and the reusable specification. Thirteen of the 20 subjects provided with the reusable specification were unable to construct mappings where no surface similarities existed (e.g. Video-Copy to Resource entities). An analysis of false mappings made by both groups emphasised dependence on surface similarities.

4.2.3 Conclusions from Study 2

The two aims of this second study were to investigate the hypothesis that analogical reuse can enhance the analytic performance of inexperienced software engineers. Results indicated that specification reuse did improve analytic performance although problems were encountered, indicating the need for tool-based support during specification understanding and transfer.

Reuse of specifications appears to improve the completeness but not the validity of solutions produced by software engineers. Reusable material presented in an abstract form appears to enhance performance more than presentation of concrete analogies, probably because similarities with the abstract template were more easily recognised than with the concrete specification. Abstract concepts in software engineering are thought to reflect expert performance and require considerable learning (e.g. Gilmore & Green 1988), so a stronger effect may have been expected from the concrete analogy. However, abstraction does not appear to help creation of more accurate specifications. A possible explanation is that the skill level of the software engineers was insufficient for them to be familiar with abstraction, even though no significant interaction between experience and the abstract/concrete condition was found.

Design of tool support can be informed by errors encountered in this study. Although analogical recognition was effective, understanding the analogies was not. Even successful reusers made mistakes many of which could be attributed to lack of detailed reasoning about the specification. Software engineers appeared to exhibit a mental laziness which was manifest in copying rather than reasoning while reusing specification components. While this effect may be ascribed as a lack of motivation among software engineers, this was not the given impression as most expressed keen interest in the experiment and its outcome. A more probable explanation consistent with other findings is that reuse offers developers a mentally-easy cognitive strategy for problem solving. Novick (1988) observed that subjects invoked cognitively-easy strategies when exploiting analogies. Chi et al. (1989, 1982) also identified mental laziness in students who solved physics problems by copying example textbook solutions which had similar surface properties to the current problem. Similarly a frequent mistake made by the software engineers in this study was to focus on surface, lexical properties of the reusable specification. The software engineers' poor understanding of the analogy was probably caused by a lack of reasoning.

Understanding a problem domain requires construction of a mental model, based on analysis and knowledge of similar domains held in memory (e.g. Gentner & Stevens 1983, Pennington 1987). However, the software engineers did not have past experience of solving similar problem types. Furthermore, the first study suggested that inexperienced software engineers tend to follow weak problem solving strategies and have difficulty in initial scoping of the domain. It is therefore not surprising that when presented with a reusable specification they take it as a potential ready-made solution.

4.2.4 Implications for Support Tools

This study validated the hypothesised reuse scenario, although it indicated that analogical specification reuse may be problematic and in need of tutorial support to encourage analogical understanding and avoid mental laziness. Support tools must ensure effective reuse by teaching analogical specifications to software engineers then emphasising the need for extensive reuse based on good analogical understanding. In short, software engineers must be grabbed '*by the scruff of their necks*' and made to understand and transfer the analogy effectively.

Reuse of the abstract template and analogical specification revealed different characteristics which can influence the design of support tools. Software engineers recognised similarities with the abstract template more easily, indicating that abstraction may be effective for analogical recognition. However they tended to copy the abstraction rather than reason analogically with them, suggesting it was less effective for analogical comprehension and transfer. On the other hand, the concrete specification encouraged more critical analogical reasoning which led to better analogical transfer and fewer errors. These differences would suggest that design of effective tool support should incorporate the benefits of both abstract and concrete analogical reuse. In particular, recognition of analogical specifications will be supported by presentation of domain abstractions to maximise this matching process. Once the analogical match is recognised, reuse of the concrete specification promotes both better analogical reasoning and understanding. These strategies will be elaborated at the end of this chapter.

Intelligent tutoring techniques are one means of assisting software engineers to understand specifications correctly. Previously, intelligent tutoring systems have been developed in well-understood domains such as algebra (e.g. Wenger 1987) and LISP programming (e.g. Polson & Richardson 1988) to help learning non-complex skills such as basic arithmetic. They generally consist of three components. The expert component has expertise of the tutor's domain, the didactic component instructs students by imparting knowledge using teaching strategies, and the diagnostic component attempts to infer students' understanding of the domain so that the most appropriate instruction can be given. Support tools also require capabilities to diagnose and explain software engineering analogies. The analogy engine will provide the analogical expertise while the diagnostic and explanatory capabilities will be incorporated into the specification advisor, see Figure 1.6. For instance, mapped components which share syntactic similarities may indicate incorrect analogical mappings, although findings from this study alone are insufficient to inform the full range of complete diagnostic capabilities.

Design of a complete intelligent reuse advisor must be informed by stronger models of how software engineers should and do reuse analogical specifications. Cognitive task and reasoning models of analogical comprehension and transfer by inexperienced software engineers are needed to determine where and how software engineers make errors during analogical reuse. Furthermore a mental model of their analogical understanding is needed, to be contrasted with theoretically-derived computational models of analogy defined in chapter 3. The need for these stronger models led to a second study of inexperienced software reusers.

4.3 Study 3- Detailed Study of Analogical Specification Reuse by Inexperienced Software Engineers

This study informed design of tool support by investigating how inexperienced software engineers understand and transfer retrieved specifications. This design was based on cognitive task and reasoning models of the analogical reuse process and a mental model of analogical understanding exhibited by software engineers. In particular, these models informed design of the specification advisor, the tool component supporting comprehension and transfer of specifications. This study used concurrent and retrospective protocol analysis to investigate several hypotheses suggested by software engineering behaviour in the previous study, namely mental laziness during specification reuse, errors which occur during reuse and how these errors may be overcome.

4.3.1 Method

Protocol analysis was used to investigate analytic and reuse behaviour of 5 novice software engineers (3M & 2F MSc students in Business Systems Analysis) with a maximum of 3 years programming experience obtained from commercial and academic backgrounds and one year of university tuition in systems analysis. They were asked to use SSA techniques to develop a specification for an air traffic control (ATC) system by reusing the flexible manufacturing system (FMS) specification (see Appendix D & chapter 2). All subjects had background method knowledge and were provided with the domain knowledge necessary to develop a specification.

4.3.1.1 Experimental Material

All subjects were given a narrative describing the air traffic control system and a specification describing the analogical FMS. The 820-word problem narrative described the domain of air traffic control and functional requirements for the computerised system. The analogical FMS specification was represented using DFD notation supplemented by short narratives describing the objectives and main processes of the system. It is shown in Appendix D. Subjects had access to both documents at all times during the protocols, but were not allowed access to any other material.

4.3.1.2 Experimental Design

Subjects were requested to think aloud and their verbal protocols were captured by a video camera which also recorded drawing and reading behaviour. During the protocols subjects were advised to take their time, and not to be afraid of verbalising too much, following the practice of Ericsson & Simon (1984). Instructions for subjects were read by the experimenter. Each subject was strongly recommended to reuse the analogical specification to develop two new data flow diagrams, otherwise the problem was to difficult to complete in the time allowed.

Subjects were given 75 minutes to develop two data flow diagrams (context and level-0). All subjects were informed of this time limit before beginning the task, and were expected to complete a solution by the end of it. While each subject performed the task a concurrent protocol was recorded. Upon completion of this task the experimenter retrospectively elicited further details of subjects' analytic strategies and mental and non-mental behaviour. First, a 10 minute written questionnaire elicited their understanding of the analogy, then the experimenter verbally questioned the subject for 15 minutes to elicit further details of subject's analytic and reasoning strategies and to investigate specific hypotheses and errors. Retrospective questioning was controlled by a checklist of different events and issues which were expected to occur during the task (e.g. "Why did you stop analysing the problem and start to develop your solution ?"). Care was taken not to prejudice the retrospective protocol, so the experimenter only asked open-ended questions, following Ericsson & Simon's practice. Finally all subjects completed a questionnaire identifying details of all previous analysis and programming experience.

4.3.1.3 The Analogy

The analogy between the ATC and FMS domains was carefully constructed to allow considerable reuse of the FMS specification, although several similar features of the reusable specification were 'red herrings' included to identify mental laziness during analogical comprehension and transfer. Otherwise, the analogy is similar to that in chapter 2.

4.3.1.4 Analysis

Protocol transcripts were analysed twice: (i) by categorising the mental behaviours represented in speech segments, usually sentences and incomplete utterances; (ii) identification of analytic strategies using a taxonomy based on criteria of mental and non-mental (physical) activity. Protocol utterances were categorised using the following definitions of mental and non-mental behaviour. Mental behaviours were:

Assertions -	verbalisation of a belief or statement of facts about the target or source domains directly attributable to the problem narrative or reusable specification (e.g. "Aircraft fly along unidirectional air corridors at predetermined heights");
Reasoning -	verbalisation of the creation, development and testing of hypotheses about the problem, its proposed solution or the source domain (e.g. "the aircraft risk colliding, hence the warning process must be automatic, and it must inform the air traffic controller with warning messages displayed on the radar screen"). Each reasoning utterance was further categorised to identify subjects' topic focus: (i) reasoning about the target (ATC) domain, (ii) reasoning about the source (FMS) domain, (iii) reasoning about analogical concepts between the source and target domains, and (iv) reasoning about general concepts which do not describe the target or source domains, or the analogical links between them;
Planning -	meta-level control over the analytic process. Two types of plan were distinguished by their content: method knowledge and SSA heuristics, or general heuristics. A method plan is "I'll list the inputs, outputs and sources, then draw a logical new DFD", while general plans include "I'll read the problem once, then look at the analogical specification";
Diagram-based testing -	generation of multiple tests to evaluate a solution DFD. Testing was guided by the flow of data through the model, and evaluated the role of each component linked to the data flow (e.g. "so the radar sends in data, which is stored, then the radar data is passed to this process....");
Other -	utterances which cannot be allocated to any other mental behaviour.

Reasoning utterances were distinguished from assertions by the degree of inference applied, concurrent non-mental behaviour (e.g. reading behaviour suggested assertions) and the tone and vocal inclination of the verbalised utterance. Non-mental behaviours were:

Information acquisition -	searching for and retrieval of data in the problem text or the reusable specification;
Model recording -	physical construction of the system specification, recorded as a

	data flow diagram;
Notetaking -	physical notetaking and highlighting not related to the construction of the data flow diagram.

Non-mental behaviour was categorised as occurring concurrently with mental behaviour. Both mental and non-mental behavioural categories were similar to those employed in the first study, although they represented some improvements in the experimental design learnt from the first study.

Analytic strategies were based on mental and non-mental behaviour and classified using eight strategies developed in part from analytic strategies identified in the first study (section 4.1). The use of video cameras in this study assisted categorisation through evidence of physical behaviours:

Gather information -	read the target document or the reusable specification;
Summarise data -	summarise the contents of the target document or the reusable specification;
Reuse -	reuse the FMS specification to develop a structured diagram representing a solution to the ATC problem;
Construct -	develop a structured diagram representing a solution without reusing the FMS specification;
Revise -	redraw the solution;
Evaluate against the target -	test the subject's solution against the target requirements in the problem document;
Evaluate against the analogy -	test the subject's solution against the reusable specification;
Summarise solution -	test the subject's solution without accessing the problem document or the reusable specification.

Gather information and summarise data strategies represented problem scoping while solution building was achieved by reusing, constructing and revising the solution. Testing occurred either by evaluating the solution against functional requirements, the reusable specification and the subject's own mental model. Illustrations of these analytic strategies and mental behaviours are given in the example protocol transcript in Appendix D and in Figure 4.5.

Reads Reusable Specification	
Gen Hyp Analogy	*For example, here we have production controller, now this could be um, our air traffic controller,*
Scribbles on Reusable Specification	
Ext Hyp Analogy	*With the same form of feedbacks, reports and warnings, changes.*
Reads Reusable Specification	
Gen Hyp Analogy	*Flexible manufacturing system can be, um, a process of moving between, um, corridors.*
Scribbles on Reusable Specification	
Gen Hyp Analogy	*Production operators are assumed to be, or could be compared to pilots,*
Ext Hyp Analogy	*and infrared sensors could be the radar.*

Read Reusable Specification	
Gen Hyp Analogy	*Product being manufactured, this is new products entering the system,*
Ext Hyp Analogy	*this is instructions, this product being manufactured*
Ext Hyp Target	*change in direction*
Draws Data Flow (Instructions), Data Store (Change in Direction)	
Mod Hyp Target	*or this could be new flight plan*
Change Data Store (New Flight Plan)	

Figure 4.5 - example protocols from subject N3, the first from the transcript while gathering information from the reusable specification, and the second while reusing the analogical specification

4.3.1.5 Protocol Categorisation

Protocol categorisation was validated through cross-marking by two independent observers with experience of protocol analysis. Each observer allocated a behavioural category to each utterance in 3 randomly-selected protocols. Inter-observer agreement was 83% of all categorised protocol utterances, and differences between observer categorisation were reconciled. Analytic strategies in three different protocols were also independently categorised, and the observers reconciled differences between categorisation of strategies to develop a common definition of analytic strategies used to categorise all protocols.

4.3.1.6 Solution Completeness and Errors

Completeness and error scores were allocated to each subject's solution to evaluate their success or otherwise in solving the problem, using a similar scheme to that reported in the first two studies. In order to construct a marking scheme a solution was developed by two expert software engineers who had considerable knowledge of both domains and the analogy between them. The marking scheme for solution completeness contained a list of components to be included in a specification, and focused on semantic features of subjects' solutions rather than on the syntax of the data flow diagramming notation. Solution components included the processes, system inputs and outputs, external entities and data store accesses in the expert solution. Subjects received a score if a component was included in the resulting data flow diagram, and each subject's completeness score represented the number of required components included in their solution. The expert solution to the ATC problem is given in Appendix D.

Subjects' solutions were also analysed to determine their validity through an error analysis. Specifications were examined for their inclusion of 5 types of syntactic error and 7 types of semantic error. Subjects received a score for each type of error included in their solution, and their score represented the total errors of different types made by each subject.

4.3.2 Results

All subjects developed a solution to the ATC problem, although subject N2 only developed the level-0 DFD. Completeness and error scores are shown in Table 4.11. Subjects N3 and N5 developed more complete solutions than other subjects. Control of their analytic processes was manifest in planning behaviour. Subjects did not appear to rely on SSA method knowledge to structure the analytic process (average 25.2 instances of general planning behaviour, 2.8 instances of method planning behaviour). Indeed, successful subjects (N3 & N5) ignored method plans (1 and 0 planning utterances respectively) and relied on the reusable specification to guide their behaviour, suggesting an important role for reuse in subjects' behaviour.

sub-ject	completeness scores (as %age of expert score)						total no of error types
	total score	re-use	con-strct	sum. soltn	eval. trget	eval. anlgy	
N1	61.4	*25*	*31.9*	*4.5*			2
N2	50	*45.4*		*2.3*	*2.3*		3
N3	72.7	72.7					3
N4	54.5	*52.3*				*2.3*	2
N5	72.7	*68.1*	*2.3*		*2.3*		0

Table 4.11 - completeness & error score totals and completeness scores by strategy for all subjects

4.3.2.1 Analytic Strategies

Analytic strategies were examined to investigate analytic behaviour in more detail. The occurrences of analytic strategies and mental behaviours were counted within each 5 minute time period (see Figures 4.6 & 4.7). The trends in frequencies of strategies suggested an initial period of problem scoping before the solution was developed by reuse, then tested by summarising the solution and evaluating it against the target. Testing strategies occurred more frequently at the end of the protocols.

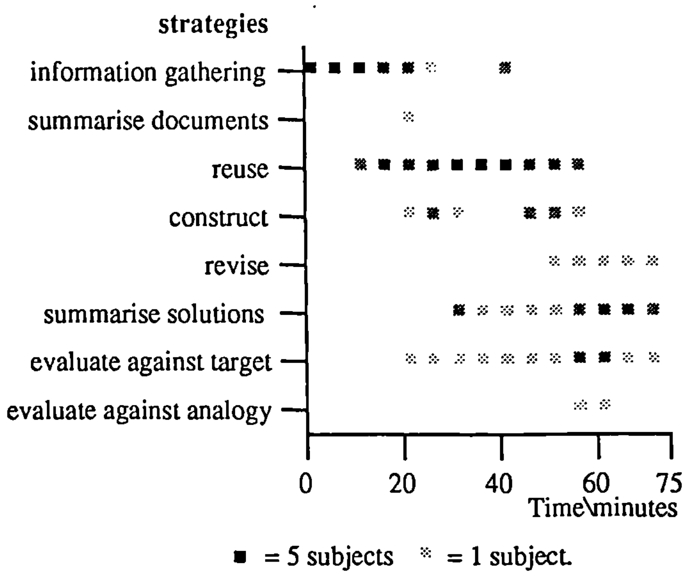


Figure 4.6 - number of subjects using a strategy within 5-minute periods

The use of analytic strategies by subject is also shown in Table 4.11. Four strategies were employed by at least 4/5 subjects however 3 strategies (summarise data, revise and evaluate against the analogy) were only used by 2 or less subjects, and construct was only used by one subject for more than 2 minutes. This suggested that subjects' analytic behaviour appeared to be based on four analytic strategies. Strategies were examined more closely by determining the average length of time spent by all subjects on each analytic strategy (see Figure 4.8). Information gathering and reuse were the most frequently-used strategies while subjects spent least time constructing and revising their solutions and evaluating it against the analogy, suggesting that the reusable specification appeared to be useful for building a solution but not for testing it. Subject N1 was unique in that she constructed rather than reused most of her solution. The payoff of using each strategy was investigated by examination of the improvement in the completeness of subjects' solutions. This payoff was measured by the components added during each strategy. Reuse resulted in the most effective development of subjects' solutions while one subject (N1) constructed some of their solution directly from the problem narrative, and testing strategies led to few improvements (see Table 4.11). As expected subjects developed most of their solutions during the building phase of analysis.

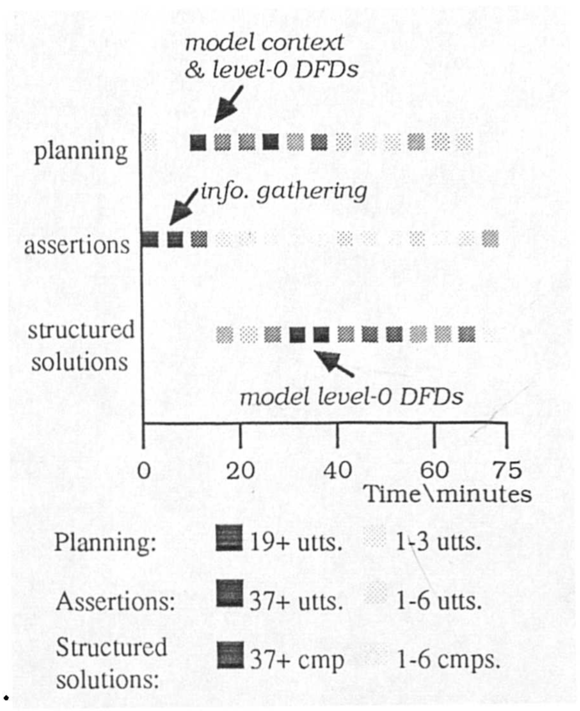


Figure 4.7 - average number of utterances of mental behaviours verbalised during 5-minute periods for all subjects, and average number of components added to structured solutions during 5-minute periods

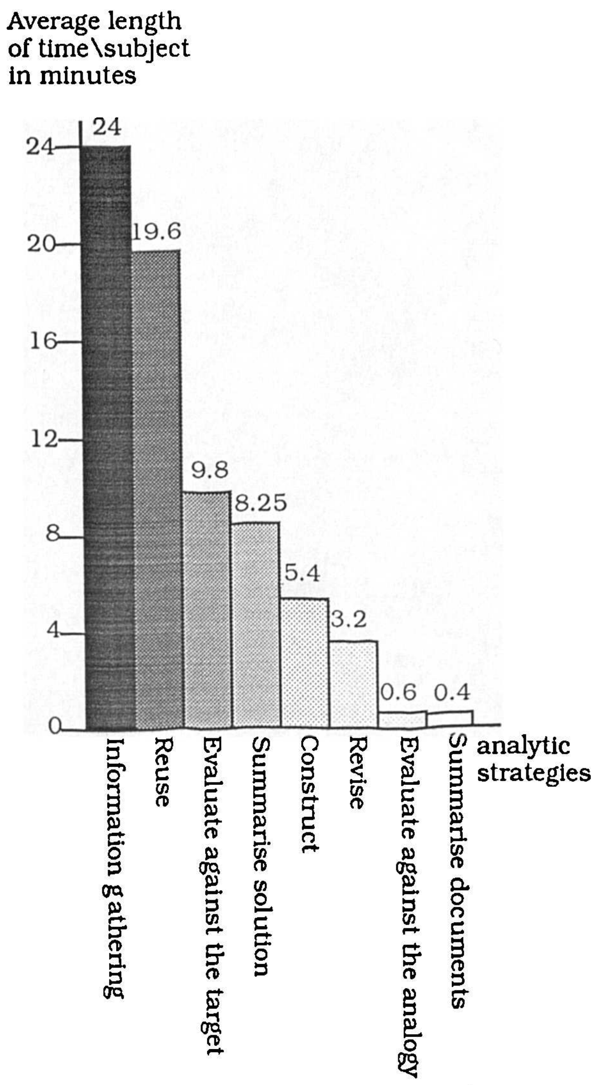


Figure 4.8 -average duration of strategies for each subject

4.3.2.2 Sequential Patterns of Analytic Behaviour

Sequential dependencies between analytic strategies for all subjects were analysed by casting the strategies in a transition matrix (occurrences of A following B and vice versa) and then constructing a network model of the temporal relationships between strategies. Frequencies greater than 5% of the total are shown in Figure 4.9. Reuse had a pivotal role in analytic strategies. Initial bouts of information gathering led directly to specification reuse and some construction of the specification. Testing involved many interactions between reuse, evaluate against the target and summarise solution strategies, hence testing and building behaviour were interleaved. Evaluation of the specification against the analogy was of little importance, suggesting that testing was achieved by summarising the solution and evaluating it against the requirements.

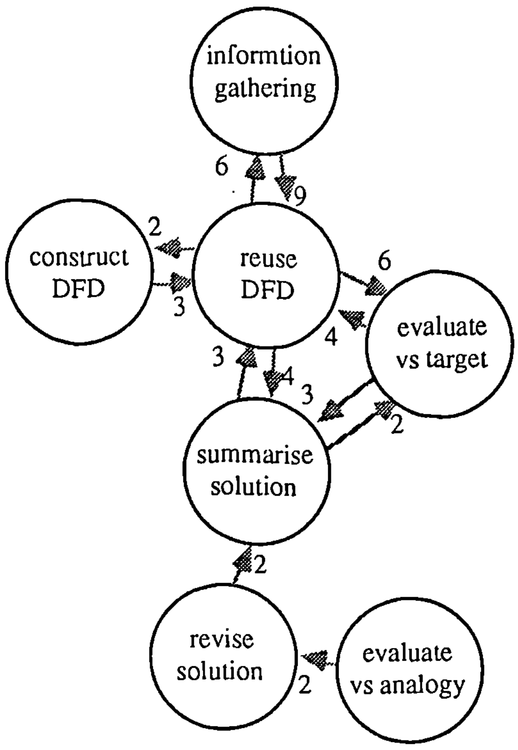


Figure 4.9 - sequential dependencies between strategies for all subjects, showing all dependencies ≥ 5% total number of dependencies

4.3.2.3 Detailed Analytic Strategies

Reuse and information gathering strategies, plus construct solution, were examined in more detail to suggest reasons for good analytic performance and to identify individual differences in analytic behaviour.

Reuse

Reuse strategies were examined for specification copying by investigating reuse of the level-0 DFD. Reuse strategies were categorised as top-down (reuse all processes first) or incremental (reuse each process and its inputs\outputs in turn), then each occurrence of reuse was analysed to determine the number of components reused and added to the solution. Reuse strategies were also examined for the number of missed reusable opportunities. The number of candidate reusable components which each subject reasoned about were counted, see Table 4.12. Results revealed that incremental reuse was only employed by the two successful subjects (N3 & N5) who also reasoned about more candidate reusable components, hence good analytic performance may require careful reuse of the analogical specification to ensure that it was fully exploited.

Concurrent and retrospective protocols revealed that 4/5 subjects (including N3 and N5) admitted to word substitution and copying during reuse, although two of these subjects admitted that copying was not an ideal approach but the only option available. This indicated that the analytic success of N3 and N5 may be attributed to detailed incremental copying of the reusable specification to exploit the close 'analogical fit' between the ATC and FMS domains. In addition, retrospective questioning revealed that N3 and N5 depended more on the reusable specification than other subjects to guide their analytic behaviour.

sub-ject	number of components reused incrementally during reuse of L0 DFD	number of reusable components omitted from L0 DFD during reuse
N1	3	8
N2	0	7
N3	14	3
N4	0	12
N5	14	0

Table 4.12 - number of components transferred incrementally or omitted during reuse of the L0 DFD

Reuse was examined to identify which features of the reusable specification were successfully transferred. All subjects reused at least 4/6 FMS processes but were less successful at transferring other reusable components. N3 and N5 transferred 65.5% of all reusable data store accesses, external entities, inputs and outputs while transfer by the other subjects (N1, N2 & N4) was poor, transferring 13.8% of all reusable data store accesses, external entities, and inputs and outputs). One reason for N1, N2 and N4's lack of success was that they did not follow top-down reuse through to its conclusion and used construct and testing strategies to add many of the system inputs, outputs and data store accesses to their solutions. Reasons for abandoning the reuse strategy varied by individual: N2 adopted an iterative testing cycle while N4 redrew his solution in order to better understand it.

To conclude, successful analytic behaviour appeared to be determined by *both* painstaking copying and incremental reuse of the analogical specification. Subjects achieved this by rigidly copying the specification at the expense of more cognitively-demanding analytic stràtegies. Other subjects adopted top-down reuse tactics which did not lead to full exploitation of the reusable specification.

Information Gathering from the Reusable Specification

Information gathering occurred with either the requirements document or the reusable specification, so these two sub-strategies were investigated. Information gathering from the reusable specification focused either on reading the DFDs or the supporting narrative. N1 and N4 gathered information equally from both sources while N2 found the DFDs too confusing and read the supporting narrative. However, N3 and N5 were so keen to begin reuse of the DFDs that they failed to notice the third page of the specification containing the supporting narrative. Indeed, N3 retrospectively claimed that the reusable DFD was the solution to all her problems, so it was unnecessary to read any further. However, reading this narrative was expected to have enhanced subjects' understanding of the analogy since it described important analogical features of the FMS domain which are unlikely to have been inferred from examination of DFDs. This indicated that subjects who extensively copied the specification made no attempt to understand it by gathering extra knowledge about the source domain.

Information Gathering from the Requirements Document

Only two subjects employed method knowledge during information gathering. N2 used different-coloured pens to underline recognised documents, external entities and data stores in the problem narrative while N5 initially identified key problem entities, although he admitted this was due to a recency effect from course work on entity-modeling techniques. In short, method knowledge did not appear to assist subjects to analyse and scope new problems.

Identification of the system boundaries was critical to scoping the ATC problem. Four of the five subjects ignored problem boundaries stated in the narrative and included the pilot in their solutions, although some subjects read and verbalised reasoning about problem boundaries during information gathering. No subject scoped the problem in detail and no subject read the problem document more than once before building a solution. Subjects read sequentially and only once cross-referenced different pages of the two documents. However, they appeared to be satisfied with general levels of information gathering and did not express fears that the problem contained too much data to remember.

Construct

One subject (N1) constructed rather than reused the context-level and the remainder of the level-0 DFD once the reuse strategy had been abandoned. However, this failed to improve on the analytic performance achieved by poorer reusers (N2 & N4). Some solution construction exhibited by N2 also failed to improve the specification completeness because the added components were either incorrect or beyond the stated problem boundaries. Therefore, constructing a specification without reuse proved to be less effective than rigidly copying it from the analogy.

Reasoning Behaviour

The first study reported in this chapter revealed the importance of reasoning in software engineers' behaviour, so their reasoning was examined in more detail. All subjects reasoned more about the target domain than about the analogy or the source domain (see Table 4.13). Totals of reasoning about the source domain was poor (average 9.8 reasoning utterances per subject) so subjects did not verbalise much understanding of the reusable domain.

subject	no. of reasoning utterances		
	target	analogy	source
N1	119	15	0
N2	192	46	28
N3	107	50	3
N4	162	78	5
N5	147	71	13

Table 4.13 - totals of reasoning utterances by subject

The occurrence of reasoning behaviours was counted within each 5 minute time period (see Figure 4.10). Most analogical reasoning occurred early in the protocol while subjects gathered information and accounted for 70.3% of all reasoning utterances during information gathering from the reusable specification. Recognition involved generating many different analogical mappings, and the subjects who transferred the specification most effectively (N3 and N5) verbalised more analogical reasoning utterances than other subjects during this recognition phase.

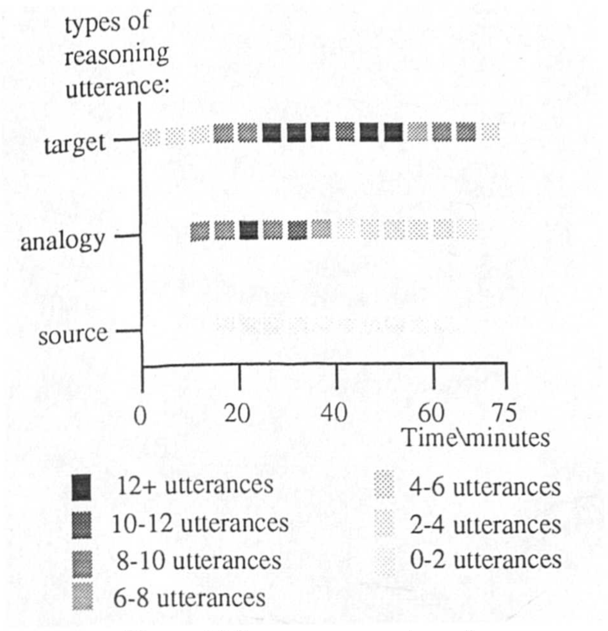


Figure 4.10 - average number of reasoning utterances verbalised during 5-minute periods for all subjects

Hypotheses' life histories were examined using the same approach as in the first study. Frequencies greater than 1% of the total are shown in Figure 4.11. More transitions occurred within target utterances and within analogy utterances than occurred between the target and the analogy, suggesting that subjects tended to separate reasoning about the target domain from reasoning about the analogy. Subjects reasoned more from the analogy to the target than vice versa, indicating the importance of reuse in specification development. Sequential dependencies between reasoning behaviours during reuse (see Figure 4.12) revealed a similar network pattern, so the analogical specification appeared to trigger hypotheses which were developed and evaluated in terms of the target domain. Indeed, closer examination of reasoning behaviour during reuse indicated that N3 and N5 exhibited the largest number of hypotheses involving reasoning about both the analogy and the target, so reinforcing the critical role that specification reuse appeared to play in their reasoning behaviour and analytic success.

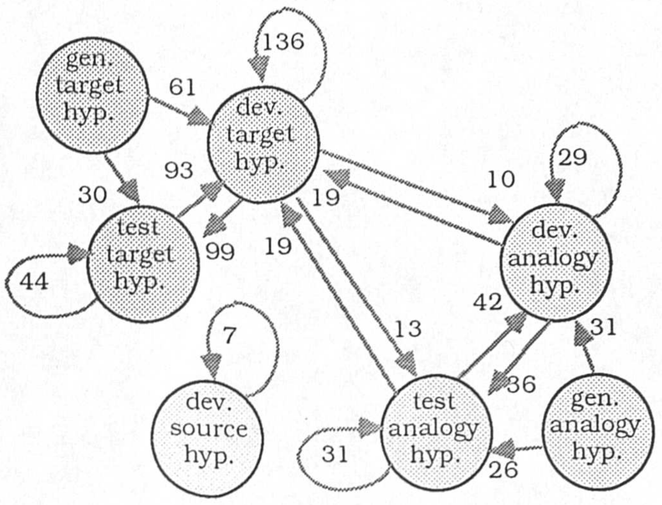


Figure 4.11 sequential dependencies
between hypothesis categories for all subjects

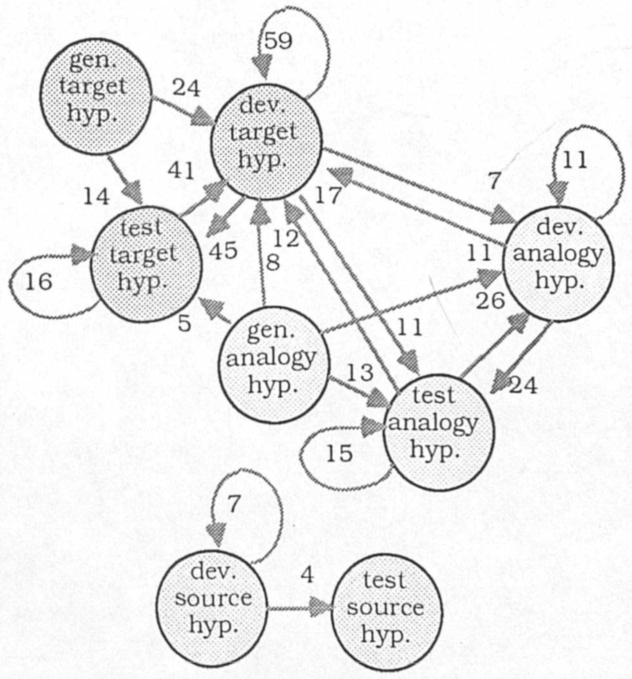


Figure 4.12 - sequential dependencies
between hypothesis categories for reuse
strategy only by all subjects

Summary of Analytic Strategies

Subjects developed much of their solutions by reuse and successful transfer was achieved by copying and incremental reuse of the analogical specification, ignoring domain knowledge described in the supporting narrative. Investigations into subjects' reasoning behaviour reinforced the critical role of the analogy during specification development. On the other hand, development of specifications without reusing the analogy appeared to be less effective. Subjects tested their specifications against the system requirements and by summarising them, however both strategies appeared to be ineffective for identifying errors and omissions. Subjects only gathered partial information about the target domain and did not use method heuristics to structure the analytic process. The dependence of subjects' reasoning on the analogy was examined further by investigating subject's qualitative understanding of the analogy.

4.3.2.4 Errors and Misconceptions

Malrules were investigated by examining specific source domain and analogical errors verbalised by subjects. Subjects exhibited six misconceptions about the source domain. Each error resulted in failure to transfer the associated source concept to the solution specification. Subjects averaged an error for every 8 source reasoning utterances (see Table 4.15), suggesting that they made a proportionally high number of source domain errors. Incorrect reasoning about source concepts could be attributed to the following malrules:

- **IF** fail to acquire source domain knowledge **THEN** abandon hypothesis (subject N3);
- **IF** unable to distinguish between different source domain concepts which share some features **THEN** reject analogical mapping (N5);
- **IF** unable to distinguish between similar reusable components which share some similarities **THEN** reject analogical mapping (N2 & N4);
- **IF** postponed reasoning about complex, detailed reusable components **THEN** do not return to hypothesis (N1 & N2);
- **IF** similar structural position in the analogical specification **THEN** match incorrect reusable component to the target using this structural match (N2 & N4).

These five heuristics warrant some explanation. The first heuristic appeared to be a consequence of poor fact acquisition from the reusable specification (N3 did not read the specification narrative). On the other hand, postponed reasoning in the fourth heuristic may have been due to the complexity of the reasoning necessary to understand the source domain concept. The remaining three heuristics indicate the importance of syntactic similarity to incorrect analogical mapping first identified in the previous study. These heuristics are demonstrated further in Table 4.15. Most surprisingly, these five malrules were derived from a total of only 48 reasoning utterances about the source domain out of a total of over 1000 such utterances. Successful analogical understanding will probably require more reasoning about the source domain, so software engineers are likely to infer many misconceptions about the source domain before developing a thorough analogical understanding.

subject	error	result	malrule
N1	fail to understand process 4 ('update the production plan')	fail to transfer the process	transfer easy mappings, fail to return to postponed reasoning about complex reusable components
N2	fail to understand flow 'data describing product's devt'	fail to transfer the flow	transfer easy mappings, fail to return to postponed reasoning about complex reusable components
N2	fail to distinguish process 4 ('update the production plan') & process 5 ('change and identify product details')	fail to transfer process 5	unable to distinguish two reuse components, fit target hypotheses to structure of specification
N3	fail to understand flow 'track section of machine'	fail to transfer the flow	abandon hypothesis due to lack of source knowledge
N4	fail to distinguish process 4 ('update the production plan') & process 5 ('change and identify product details'), then confuse with process 3 (' monitor to ensure the production plan is met')	fail to transfer process 5	unable to distinguish two source components, fit target hypotheses to structure of specification
N5	fail to distinguish between 'product' and 'machine'	fail to transfer all relevant components	unable to distinguish two source concepts due to domain-dependent interference, so fit target hypothesis to structure of specification

Table 4.15 - malrules exhibited by subjects while reasoning about the source domain

Subjects also exhibited five false analogical mappings (incorrect mappings not based on syntactic similarities between components), see Table 4.16. These malrules were:

- **IF** some syntactic similarities between mapped source and target concepts **THEN** incorrectly map other related components based on these similarities (N1, N3);
- **IF** two objects share equivalent positions in the two specifications **THEN** incorrectly match target hypotheses to reusable components (N4);
- **IF** synomynous source concepts exist in specification **THEN** interchange components during reuse, leading to false mappings between concepts (N5).

subject	error	result	malrule
N1	'new products entering system' maps to 'aircraft takeoffs'	aircraft takeoffs included (wrongly) in the scope of the system	false mapping caused by common syntactic features ('new' & 'input')
N3	'product being manufactured' maps to 'changes in direction'	inclusion of ATC instructions in an inappropriate area of the solution	false mapping caused by common syntactic features ('new' & 'input')
N3	'new products entering system' maps to 'instructions'	---||---	---||---
N4	'aircraft' maps to 'production operators'	no effect on solution	failure to fit poor reasoning about target concepts into the analogical structure
N5	'flight' maps to 'product'	inconsistency in use of 'flight' and 'aircraft' terms in the solution specification	synomonous use of different target terms contradicting corresponding mappings

Table 4.16 - malrules exhibited by subjects while reasoning about analogical mappings

To conclude, this study suggested nine distinct malrules leading to reuse-related errors by four of the five subjects. Errors varied and were thinly spread across subjects so that there were no clues to links between error occurrence and subjects' success. Although further work is required to elaborate and evaluate these malrules, findings from the study can provide a starting point for a diagnostic module incorporated into support tools.

4.3.2.5 Analogical Understanding

Subjects' final understanding of the analogy was examined by retrospectively requesting target mappings for each object in the source domain. Subjects only partially understood the analogy and recognised on average 51.5% candidate analogical mappings. In general subjects recognised mappings with *product, two products in the same section, misdirected product manufacturing, production plan, production controller, production operator and infrared sensor* but showed a poor understanding of the physical domain structure and only recognised a total of three correct mappings with *machine, track section and production floor layout*, see Table 4.17. When retrospectively prompted for critical analogical mappings no subjects identified *machine* and *track section* as key analogical concepts and only two subjects recognised mappings with the *production floor layout* as important. Poor understanding of analogical physical structures may have been due to the inexact fit between the continuous three-dimensional ATC domain and the segmented two-dimensional FMS production paths, indicating that subjects may only understand close analogical matches.

analogical mapping		correctly mapped	incorr-ectly mapped
source domain	target domain		
production controller	air traffic controller	5	0
infra-red sensors	radar	5	0
production plan	flight plan	5	0
two products in same space	two aircraft in same space	5	0
product	aircraft	4	1
misdirected product	aircraft off course	3	0
production floor layout	airways	3	1
production operator	pilot	3	2
production track	air corridor	2	3
delayed product manufacture	delayed flight	2	1
manufacture of a product	flight	2	1
production track section	air space	1	2
lost product manufacture	missing flight	0	1
machine	air space	0	2
job	flight step	0	4

Table 4.17 - retrospective understanding of analogical mappings, prompted from candidate source domain mappings

Interestingly N3 and N5 mapped the largest number of wrong source objects while N2 and N4 mapped the largest proportion of correct analogical mappings (see Table 4.18) based on the ratio of correct to incorrect mappings. Therefore subjects who copied their solutions and recognised analogical mappings from the reusable DFD seemed to have a poorer understanding of the analogy. Good analogical understanding may have been due to their willingness to read the supporting narrative during information gathering. Subjects N1, N2 and N4 all read the supporting narrative and exhibited less incorrect analogical mappings than N3 and N5. The reusable DFDs represented solution knowledge, however analogies were critically determined by similarities between the underlying ATC and FMS domains, so reading domain descriptions may have led to improved analogical understanding, although this understanding was not complete.

subject	total correct	total incorrect
N1	6	4
N2	9	2
N3	8	8
N4	9	0
N5	11	5

Table 4.18 - totals of retrospectively-recognised correct and incorrect analogical mappings

This link between analogical understanding and domain understanding was investigated further by allocating reasoning utterances to one of three categories. Domain utterances represented inferences about the underlying domain and were independent of the required computer system. Reasoning about the target and reusable computer systems were categorised as solution utterances. Utterances describing the required high-level functionality of the target computer system were categorised as requirements utterances. Results revealed that subjects reasoned mostly about the computer solution. Subject N2 exhibited most reasoning about the domain, so the supporting narrative during information gathering may have encouraged more reasoning about the domain underlying the reusable specification.

Subjects' analogical understanding was also investigated by reexamining their concurrent protocols. Retrospective questioning elicited key analogical mappings between objects, then reasoning behaviour during concurrent protocols was examined to identify uttered relations between critically-mapped pairs of objects which may be compared to the domain terms in chapter 3, see Figure 4.13. Findings revealed that subjects reasoned in terms of functional relations between target objects rather than the domain's physical structure. They support the logical model of software engineering analogies at least in part, since subjects did not reason about physical domain structures in isolation, but considered the logical domain structure in relation to key state transitions and functions (*e.g. aircraft must not deviate from aircraft*).

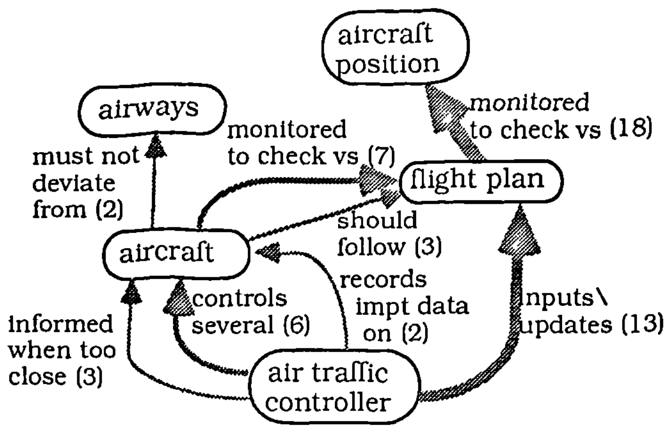


Figure 4.13 - composite model of subjects' problem space

To summarise, subjects exhibited only a partial understanding of the analogy. Those who gathered domain knowledge from the supporting narrative understood more of the analogy than subjects who directly copied the solution specification, possibly because the narrative represented domain features which were critical to understanding the analogy. Subjects well-understood six analogical concepts but exhibited a poor understanding of analogical mappings between physical concepts in both domains, possibly due to the inexact match between the physical ATC and FMS domains.

4.3.3 Conclusions from Study 3

This third study investigated problems encountered by inexperienced software engineers when understanding and reusing an unfamiliar but analogically matched specification. In particular, it examined problems of mental laziness manifest as copying during reuse. Findings revealed that problems occurred during both analogical understanding and transfer, and a better analogical understanding did not imply improved transfer, a result also reported by Novick & Holyoak (1991). Indeed, software engineers who copied the specification most effectively exhibited the poorest analogical understanding. As a result software engineers could be divided into two groups: (i) those who preferred rigid copying, and: (ii) those who reused the specification more opportunistically.

Possible reasons for poor analogical understanding were three-fold. First, the inexperienced software engineers lacked the relevant domain knowledge, hence key analogical constructs were difficult to recognise and elaborate (e.g. Gick & Holyoak 1983). Second, analogical comprehension can be improved through notetaking (e.g. Gick & Holyoak 1983), however software engineers made little use of sketches and notes. Third, comparison with the logical model of software engineering analogies suggests that the software engineers in this study failed to recognise many key facts about the ATC/FMS analogy, that is two or more aircraft or products in the same air space or track section. Subjects did not recognise the *air space* or *track section* concepts as important analogical components. One reason may have been the lack of syntactic similarities for analogical recognition or the inconspicuous nature of the *track section* component in the reusable specification. Component prominence was dependent on the specification's notation, and reuse of entity-relationship diagrams may have led to a different result. On the other hand well-understood mappings were both syntactically similar and prominent, suggesting the need for explicit analogical triggers in the reusable specification. Of course, an alternative reason for this poor understanding is the inexact fit between the physical three-dimensional ATC and two-dimensional FMS domains.

Possible reasons for poor analogical transfer were two-fold. First, software engineers' failure to infer complex analogical mappings between data stores representing domain objects (e.g. *track section* and *airways*) may have limited transfer of components without syntactic similarities between them. A second, complementary reason is that assimilating data from three different pages of the reusable specification may have led to less-structured and ineffective transfer of the specification. As a result, successful specification reuse may require more integrated techniques to support analogical understanding and transfer, including explanation of reusable components and analogical mappings during transfer.

4.3.4 Implications for Support Tools

Findings have implications for the design of tools supporting analogical specification reuse. These tools must infer software engineers' analogical understanding, identify symptoms of mental laziness and diagnose analogical errors so that the best assistance can be provided. The nine malrules of analogical misuse provide an empirical basis for error diagnosis during analogical reuse, with implications for explanation of analogical mappings discussed further in chapter 5. However, diagnosis from just nine malrules is unlikely to determine analogical misconceptions, so a dialogue (Self 1988) which questions software engineers about their analogical understanding will be needed to support diagnosis. A second major implication for tool support is the need to discourage mental laziness by encouraging the software engineer to understand the specification prior to its reuse.

The analogical specification proved an effective trigger for hypotheses about the target domain, however software engineers' analogical reasoning was poor, so help is also needed to promote analogical reasoning in the form of external memory aids and explanation of analogical mappings. Similarly, help is needed when scoping and structuring the analogical specification, so guidelines to this effect may assist specification reuse.

Most findings from this study identify the problems which tool support must overcome, i.e. what tool support must do, rather than how support will proceed. One solution to specifying how support will proceed was to examine how successful, expert reusers undertake reuse tasks, to determine how they work and why they are successful. Therefore, a fourth study examined experienced software engineers' strategies to determine how they understood and transferred analogically matched specifications.

4.4 Study 4: Expert Analogical Specification Reuse

The aim of this study was to examine the analogical comprehension and transfer strategies of successful reusers in the same scenario as the third study. In particular it investigated cognitive task and reasoning models of effective analogical specification reuse to inform design of tool support. Expert software engineers, some with over 20 years commercial analytic experience, analysed and reused an unfamiliar problem domain so that they were required to develop understanding of the domain before analogical transfer took place, thus providing clues to both comprehension and transfer strategies of experts. Unfortunately, analogical specification reuse is a novel concept, so no expert specification reusers were available. Rather this study investigated expert software developers with considerable exposure to many different classes of software engineering domain.

4.4.1 Method

Protocol analysis was used to investigate analytic and reuse behaviour of 12 expert software engineers. They (9M,3F) had a minimum of 6 years programming and between 6 months and 20 years analysis experience obtained in commercial environments. During the experiment 2 software engineers (1M,1F) failed to verbalise sufficiently and their protocols were discarded. Data from the remaining 10 experts provided the basis for results presented in this study.

The software engineers were drawn from four different sources. Five (4M,1F) worked in local government and had knowledge of structured analysis techniques, although only one had employed these techniques on a daily basis. Four software engineers (3M,1F) employed by 2 consultancy firms had regularly used structured analysis techniques. The 10th expert (1F) lectured in computing at an academic institution and had experience in applying and teaching structured analysis techniques.

Subjects were asked to use Structured Systems Analysis (SSA) techniques (De Marco 1978) to develop a specification for the air traffic control (ATC) system from a specification describing a flexible manufacturing system (FMS). Experimental material, design and analysis were the same as those given to inexperienced software engineers described in study 3, so see chapter 4.3.1 for method details. Cross-categorisation of analytic strategies and mental and non-mental behaviours was also the same as in study 3.

4.4.2 Results

All subjects developed a specification to the ATC problem. Subject E4 initially misunderstood the experimental instructions and developed part of his specification without accessing the reusable specification. Completeness and error scores are shown in Table 4.19. There was a negative correlation between completeness and error scores (Spearman Rank Order Coefficient r= -0.722, p=0.018), hence stronger subjects developed more complete and valid specifications. There were notable differences between subjects' success, for instance E11 developed a specification which was only half as complete as that of E3, so there appeared to be differences in the quality of subjects' reuse and analytic behaviour. Indeed, overall performance of the experienced subjects was not significantly improved over their inexperienced counterparts from study 3, so reasons for the success of the most effective subjects were examined.

subjt	completeness scores (as %age of expert score)						total no. of error types
	total score	re-use	con-strct	sum. soltn	eval. trget	eval. anlgy	
E1	59.1	*20.5*	*20.5*		*15.9*	*2.3*	2
E3	79.5	*70.5*	*4.5*		*4.5*		1
E4	75		*72.8*		*2.3*		1
E5	70.5	*47.8*	*15.9*		*6.8*		1
E7	63.6	*50*	*6.8*		*2.3*	*4.5*	1
E8	70.5	*47.8*	*11.4*		*4.5*	*4.5*	1
E9	68.2	*61.4*			*9.1*		2
E10	61.4	*61.4*					2
E11	38.6	*29.6*	*4.5*		*4.5*		3
E12	61.4	*27.3*	*27.3*	*4.5*	*2.3*		1
Avge	64.8	*41.9*	*16.4*	*0.45*	*5.22*	*1.12*	1.5

Table 4.19 - completeness & error score totals and completeness scores by strategy for all subjects

Initially analytic strategies were investigated to identify patterns of subjects' behaviour. The occurrence of analytic strategies was counted within each 5 minute time period (see Figure 4.14) and occurrences of mental and non-mental behaviours were also counted (see Figure 4.15). Subjects began with a period of problem scoping followed by building then testing the solution. Closer examination of each subjects' analytic strategies revealed individual differences in their approaches to reusing and testing specifications. Analytic behaviour was also investigated by retrospectively asking subjects to describe their strategies if they were required to repeat a similar reuse problem. Three subjects (E5, E7, E8) proposed a two-step approach: produce a first-draft solution from the reusable specification, then test and improve that solution. Two subjects (E1, E10) also claimed to employ the two-step approach during their protocol sessions while E3 spent the last 20 minutes of the protocol 'filling in the gaps in the solution'.

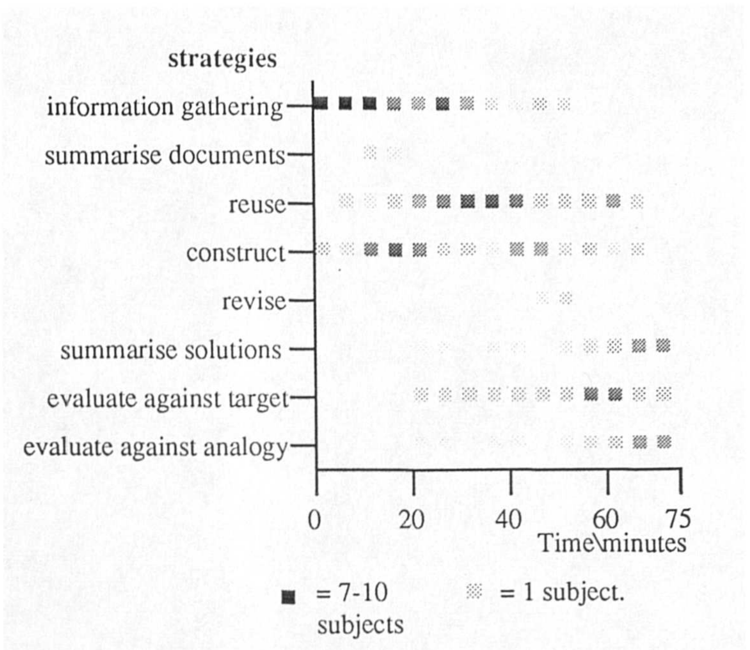


Figure 4.14 - number of subjects using a strategy within 5-minute periods

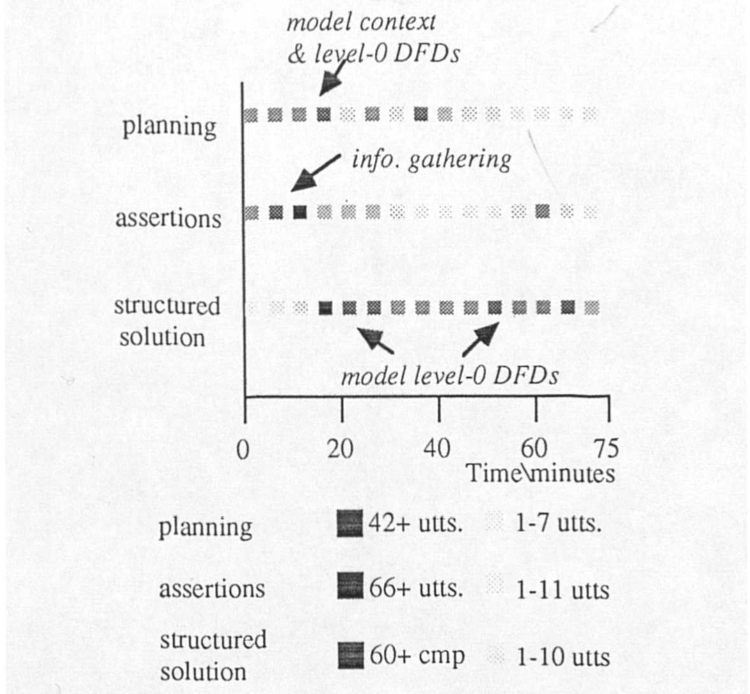


Figure 4.15 - average number of utterances of mental behaviours verbalised during 5-minute periods for all subjects, and average number of components added to structured solutions during 5-minute periods

4.4.2.1 Planning Behaviour and Analytic Heuristics

Subjects' analytic behaviour was suggested by control of their analytic processes. High-level control of this process was manifest in planning behaviour. As in the previous study subjects did not appear to rely on method knowledge to structure the analytic process, as on average 23.1 instances of general planning behaviour were observed, compared with 5.6 instances of method planning behaviour. Two subjects (E8, E9) from a consultancy background exhibited the greatest use of method knowledge, and the four consultants accounted for 67.8% of all method planning behaviour. Protocol transcripts of the four consultants were examined more closely. All consultants considered developing entity-relationship models to understand the ATC problem while only one other subject (E1) employed SSA techniques during information gathering. Of the consultants E11 used method heuristics to constrain the scope of the problem space and develop an incomplete solution. However, overall there was little difference between the analytic behaviour of consultants and other subjects, hence method knowledge appeared to have little influence on analytic performance.

To sum, method knowledge had little influence on analytic behaviour. Indeed, some subjects preferred to contradict SSA advice and considered implementation issues during analysis. Subjects also tended to adopt a 'try it and see' approach to specifying the problem, emphasising the importance of structured diagrams for developing as well as recording specifications. However, these behaviours did not indicate any reasons for success, so analytic strategies were examined more closely.

4.4.2.2 Analytic Strategies

Six of the eight strategies were employed by more than 7/10 subjects and two strategies (summarise data and revise) were only used by 3 or less subjects for short lengths of time, hence expert analytic behaviour appeared to be primarily based on 6 analytic strategies.

The average length of time spent by all subjects on each analytic strategy is given in Figure 4.16. There was a correlation between solution completeness and the amount of time spent reusing the analogical specification (Spearman Rank Order Coefficient r=0.647, p=0.047), indicating that reuse may be an important determinant of good analytic performance. One exception (E4) was discarded from the statistic because he misunderstood the experimental instructions and failed to read the reusable specification during the first 26 minutes of the protocol. The payoff from strategies, measured by the components added and reasoned about during each strategy, revealed that reuse and construct strategies resulted in the most effective development of subject's solutions while summarise data and revise strategies were ineffective (see Table 4.19) and evaluation against the target was the most effective testing strategy.

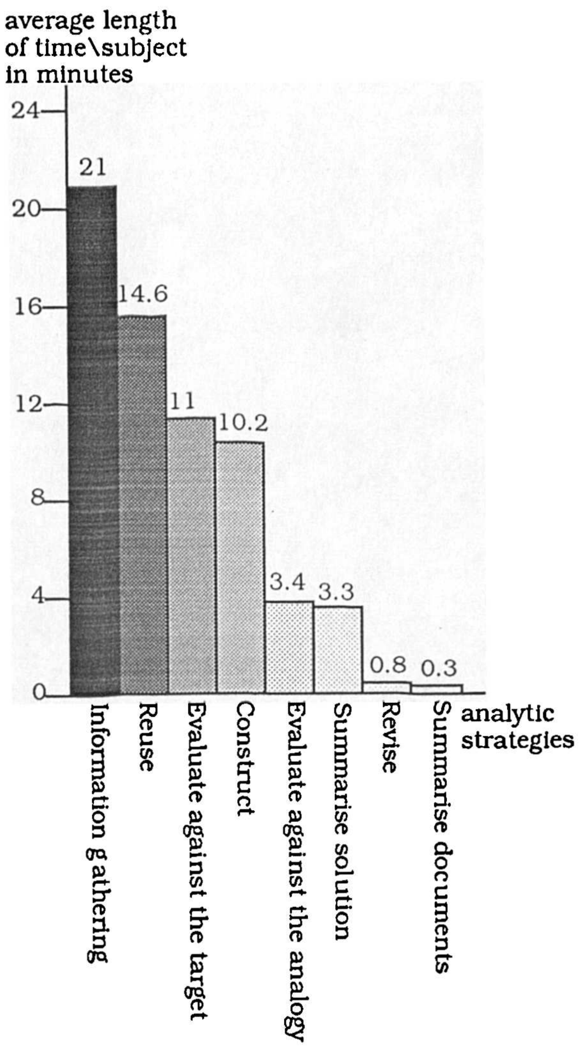


Figure 4.16 - average duration of strategies for each subject

4.4.2.3 Sequential Patterns of Analytic Behaviour

Sequential dependencies between analytic strategies for all subjects were analysed by casting the strategies in a transition matrix (occurrences of A following B and vice versa) then constructing a network model of temporal relationships between strategies. Frequencies greater than 5% of the total are shown in Figure 4.17. Reuse had a pivotal role in the analytic strategies, often involving interaction with construct and evaluate against the target strategies, while the link from evaluate against the target to evaluate against the analogy shows the involvement of the reusable specification during testing and validation. This considerable interaction between reuse and testing strategies suggested that specification reuse played an important role in subjects' analytic behaviour.

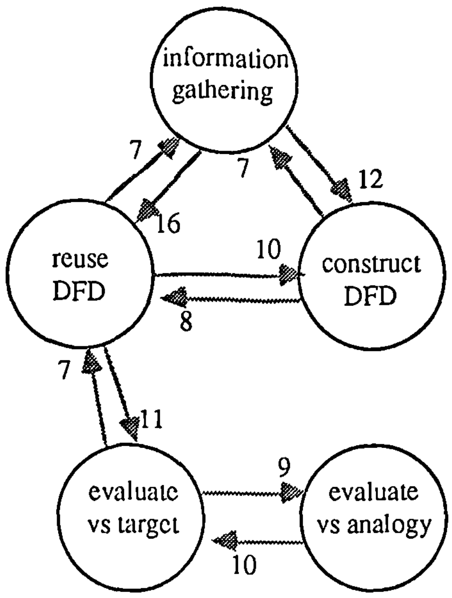


Figure 4.17 - sequential dependencies between strategies for all subjects, showing dependencies ≥ 5% of total number of dependencies

4.4.2.4 Detailed Analytic Strategies

Initial examination of analytic strategies revealed that specification reuse was related to subjects' success. Reuse and other strategies were examined in more detail to suggest reasons for good analytic performance and identify individual differences in behaviour.

Reuse

Reuse proved to be the most effective strategy, so reuse of the level-0 DFD was investigated more closely. This study hypothesised that expert software engineers would attempt to understand rather than copy an analogical specification. Concurrent and retrospective protocols revealed that 7/10 subjects were wary of the analogical specification and refused to take its similarities for granted. Three subjects (E5, E7, E11) did admit to some copying of reusable processes, although this copying was limited and these three subjects also exhibited behaviour consistent with analogical reasoning and specification understanding.

Further evidence of analogical understanding was suggested by seven reuse-related heuristics verbalised by subjects. Five of these heuristics were concerned with knowledge required to achieve reuse and suggested that subjects attempted to understand the analogy before reusing the analogical specification:

- be concerned with the details during reuse;
- only model the reusable specification if it is understood;
- do not be constrained by knowledge of previous systems;
- be wary of differences between the solution and reusable DFDs;
- acquire more data on the reusable data stores.

Two other types of heuristic suggested that the process of reuse could be constrained and ordered by the structure of the analogical specification:

- reuse the structure of the DFD;
- fit the reusable specification into the new specification.

This indicates that transfer of the reusable specification may best be achieved by using the analogy as a template to control analogical reuse. Furthermore, heuristics suggested that specification understanding was important to expert software engineers. Bouts of reuse were examined for the number of exploited reusable opportunities. The number of candidate reusable components which each subject reasoned about during reuse were counted, see Table 4.20. A correlation existed between subjects' completeness scores and the number of considered reusable components (Spearman Rank Order Coefficient $r=0.637$, $p=0.048$), indicating that successful subjects reasoned about more reusable components and that good analytic performance may have required careful reuse of the analogical specification to ensure that it was fully exploited. E9 was the only subject to reason about all candidate reusable components. She was also the only subject to reuse the analogical specification by directly modifying it and changing or deleting the names of reusable components based on analogical mappings, hence the specification acted as a development template to control analogical transfer. This thorough exploitation of the analogy by successful subjects contrasts markedly with copying demonstrated by novice software engineers in the previous study.

subject	number of reusable components which subjects did not reason about during reuse of level-0 DFD
E1	9
E3	8
E4	-
E5	8
E7	6
E8	8
E9	0
E10	9
E11	10
E12	22

Table 4.20 - number of reusable components not reasoned about during reuse of level-0 DFD, by subject

Further studies revealed that effective reusers who transferred most reusable components correctly (E3, E9, E10) exhibited long, uninterrupted bouts of reuse. The importance of these long bouts was investigated in more detail by measuring the length of the longest reuse bout exhibited by each subject. There was a strong correlation between the completeness scores of subjects' final solutions and the length of the longest bout of reuse in minutes (Spearman Rank Order Coefficient $r=0.911$, $p\leq 0.001$), hence effective analogical transfer appeared to be related to long and uninterrupted bouts of specification reuse. On the other hand, less successful subjects abandoned reuse and employed alternative strategies to complete their solutions. They did not appear to adopt reuse to guide their reasoning once alternative strategies, such as intermittent testing of the solution and frequent access to the problem requirements document, had been used. Thus, mixing development and testing strategies may not be the most effective way of reusing specifications.

To summarise investigation of subjects' behaviour during reuse suggested that they attempted to understand rather than copy the reusable specification and the analogy. Successful reuse was achieved by controlled transfer of the structure of the reusable specification as a template and by detailed reasoning about all of the candidate reusable components during an uninterrupted bout of reuse. Subjects who mixed reuse with other strategies failed to fully exploit the analogy.

Construct

Six subjects employed the construct strategy (develop a solution without reuse) for a total of more than 10 minutes. E1 and E5 developed the least complete and the least valid context DFDs respectively while E12 constructed the second-least complete level-0 DFD and E8 only constructed three processes before developing the remainder of his solution by reuse. In addition E9 modelled the existing ATC system. These findings indicate that subjects who constructed parts of their solutions without extensive reuse also developed poor solutions, thus supporting the conclusion that subjects' reuse strategies were one major reason for their success.

Information gathering

Analogical recognition and comprehension occurred while gathering information from the reusable specification. Subjects tended to favour the supporting narrative for this purpose because 6/10 subjects read this narrative only while 3/10 subjects read both the supporting narrative and the reusable structured diagram. Two subjects found the DFDs difficult to understand, although this was not the case for other subjects. After all, DFDs are supposed to be simple to understand. A more likely reason was that structured diagrams did not contain the key domain knowledge found in the narrative, so subjects may have considered the importance of assimilating key domain knowledge when understanding the analogy.

Evaluate against the analogy

Analogical transfer was also investigated during evaluation of the solution against the analogy. However, this strategy only resulted in an average increase per solution of 0.5 (1.12%) completeness points (see Table 4.19) from the transfer of analogical components which had been omitted when building the solution, suggesting that little analogical transfer occurred. Possible reasons for ineffective testing against the analogy were two-fold. First, subjects appeared to test their solutions superficially against the analogy. They only evaluated parts rather than the whole of the DFDs and did not reason about the analogy in the necessary detail. Second, those subjects (E4, E8) who did reason about the analogy in detail were still unable to recognise analogical mappings or correctly transfer reusable components, suggesting that successful transfer was inhibited by ineffective analogical comprehension. Subjects' difficulty in transferring detailed reusable components may be due to the cognitive distance within the analogy. Mapping equivalent source and target concepts required considerable abstraction to identify and apply key similarities to the ATC specification.

Evaluate against the target

Evaluation against the problem document resulted in an average increase per solution of 2.3 (7.6%) completeness points (see Table 4.19), hence it proved a more effective testing strategy than evaluating specifications against the analogy. Improvements to solutions were additional requirements which had been omitted during solution building. There was a strong correlation between length of time spent evaluating and improvements made to a solution (Spearman Rank Order Coefficient $r=0.953$, $p\leq0.001$) hence successful evaluation proved to be a time-consuming process, although no subjects successfully identified all omissions from their solutions.

Summarise solutions

Finally, summarisation of solutions only resulted in an average increase per solution of 0.2 (0.45%) completeness points (see Table 4.19). Improvements to solutions resulted from inferred omissions from subjects' solutions. Subjects did not reason about solution concepts in detail, and were unable to identify omissions and errors in their solutions. Only one subject (E7) exhibited much diagram-based testing during solution summarisation, but this did not result in many improvements to his solution.

Reasoning Behaviour

There was no correlation between success, subjects' experience and totals of reasoning behaviour. The content of reasoning was examined by categorising each utterance as reasoning about the target or source domains or the analogical mappings between them, and 9/10 subjects reasoned least about the source domain, see Table 4.21. The occurrence of types of reasoning behaviour was counted within each 5-minute time period. Information gathering from the reusable specification coincided with peaks in reasoning about the analogy and the source domain after 15 and 20 minutes respectively, suggesting that subjects reasoned most about the reusable specification during problem scoping.

subject	no. of reasoning utterances		
	target	analogy	source
E1	182	4	31
E3	185	42	0
E4	143	48	16
E5	287	120	13
E7	302	78	59
E8	324	93	35
E9	332	112	27
E10	219	119	60
E11	116	48	1
E12	145	45	0

Table 4.21 - totals of reasoning utterances by subject

The life history of each hypothesis was traced by its thematic content until eventual rejection or resolution. Sequential dependencies between reasoning behaviour for all subjects were analysed to construct a network model of the temporal relationships between categories. Frequencies greater than 1% of the total are shown in Figure 4.18. Subjects exhibited more transitions among target utterances and among analogy utterances, suggesting that they tended to separate reasoning about the target from reasoning about the analogy. Sequential dependencies from reasoning about the analogy to reasoning about the target supported the importance of reuse in subjects' behaviour. The pattern and frequencies of transitions suggested that many hypotheses were triggered from the analogy but evolution and evaluation of hypotheses occurred in the target domain. Sequential dependencies were also used to investigate reasoning behaviour during strategies. Reasoning behaviour during reuse and evaluate against the analogy revealed a similar network pattern which was not observed during other analytic strategies, thus reinforcing the critical role that specification reuse appeared to play in subject's reasoning behaviour. It suggested that subjects reasoned analogically with the reusable specification, however, it was the quality rather than the quantity of this reasoning which appeared to be critical in this study.

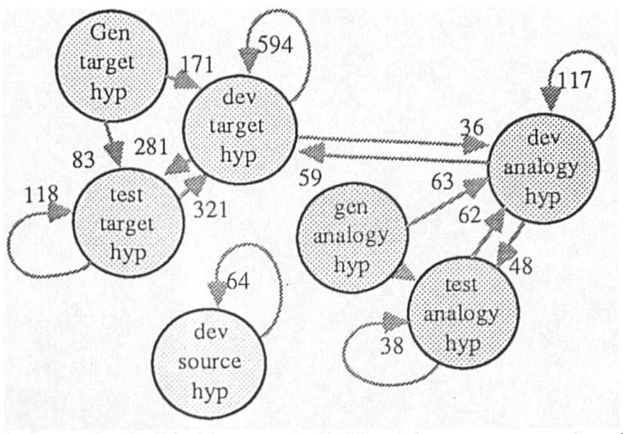


Figure 4.18 - hypothesis life history of all 10 subjects, expressed as frequencies of transitions between reasoning behaviours, showing major reasoning behaviour about the target, source and analogy. Only transition ≥ 1% of total number of transitions are shown

Summary of Analytic Strategies

Specification reuse appeared to be important to subjects' analytic performance. Reuse appeared to be effective for transferring the structure and major functions of the specification but not for reusing detailed analogical components. Successful subjects transferred the structure of the reusable specification as a basis for guiding reuse of detailed analogical components. This success was linked to their motivation for analogical transfer rather than any sophistication of their thought processes during analogical reuse. Subjects also correctly transferred more reusable components as a result of long, uninterrupted bouts of reuse behaviour, suggesting that subjects' concentration may also have been important. On the other hand, short periods of reuse interspersed with problem scoping, solution testing and construction resulted in ineffective transfer of reusable components. Analogical transfer proved effective for specification building but not for testing, so subjects evaluated their specifications against target requirements rather than the analogical match. Finally, in contrast to inexperienced software engineers, most experts attempted to understand the analogy rather than copy the specification, and analogical understanding was based on key domain knowledge represented in the narrative document. Thus, these findings may provide the basis for expert analogical comprehension and transfer strategies. However, quantities of analogical reasoning behaviour did not correlate with analytic success, so subjects' qualitative analogical reasoning was examined more closely during analogical recognition, comprehension and transfer.

4.4.2.5 Recognition of the Analogy

Analogical recognition was investigated by examining subjects' concurrent protocols to identify their first analogical mappings when gathering information about the reusable specification. Not surprisingly, subjects appeared to be influenced by the sequential order of the narrative in the reusable specification and recognised analogical mapping with objects described in the first paragraphs of the document. The concepts which prompted analogical recognition were investigated further during retrospective questioning to reveal three triggers to analogical recognition. Five subjects recognised the analogy through similarities between the radar and the infrared sensors, three subjects recognised that the two systems had similar objectives (*e.g. collision detection and avoidance*) and one subject recognised that both problems involved objects which moved in a space. In short, the analogy was recognised from a variety of viewpoints (e.g. object functionality), suggesting individual differences in subjects' understanding.

4.4.2.6 Comprehension of the Analogy

Analogical comprehension was investigated by examining concurrent verbalised misconceptions and retrospective reports of understood analogical mappings.

Error Analysis

Concurrently verbalised false analogical mappings and source domain misconceptions were investigated to identify possible misunderstandings about the analogy. Ten false mappings were made by 5 different subjects. Five of these false mappings suggested breakdowns in analogical reasoning when a subject was unable to infer analogical mappings and developed alternative, incorrect solutions to specification requirements. Other errors varied and could be ascribed to several possible causes (see Table 4.22), including syntactic similarities which indicated that subjects may have copied some analogical components. However, these components were neither prominent or central in the reusable specification, and copying was not on a scale of that exhibited by inexperienced software engineers. Four subjects made a total of 6 erroneous inferences during reasoning about the source domain, although no error pattern could be discerned, see Table 4.23. Interference from subjects' knowledge of other applications may also have inhibited analogical mappings.

error type	number of subjects
breakdown in analogical reasoning, adoption of alternative solutions	5
incorrect analogical reasoning	2
cascade effect on erroneous analogical mappings	1
erroneous mappings based on surface similarities	3

Table 4.22 - types of analogical error concurrently verbalised by subjects

error type	number of subjects
poor understanding of source goals and domain structure	2
interference from other applications	2
interference from programming knowledge	1
confusion between similarities between processes	1

Table 4.23 - types of source domain error concurrently verbalised by subjects

To sum, error analyses revealed that most misconceptions occurred while reasoning about minor functions and data stores while subjects generally showed a good analogical understanding of important functions and problem boundaries. There was some evidence of mental laziness while reasoning analogically to transfer minor specification components, suggesting that subjects were able to understand the broad outline of the analogical match but were unable to fill in the analogical details. The major difference between these errors and those exhibited by novices may have been the degree of reasoning involved. Novice errors could be explained by mental laziness while experts exhibited more incorrect reasoning leading to false analogical mappings and incorrect understanding of source concepts.

Retrospective Understanding

Subjects' final understanding of the analogy was examined by retrospectively requesting target mappings for 15 concepts in the source domain. Analogical understanding was only partial and on average experts only correctly recognised 43.8% of candidate analogical mappings, which was actually less than the inexperienced software engineers in study 3 (51.5%). This result was surprising since novices copied specifications while experts attempted to understand the analogy, so it may indicate that the analogy may have been more difficult to understand than originally assumed.

Like novices, experts also tended to understand key analogical concepts (see Table 4.24), suggesting that their analogical understanding appeared to be based on six central analogical mappings. They also believed that these mappings were important to the analogy. When retrospectively prompted for critical analogical mappings subjects identified five of these six central mappings as very important for supporting reuse of the FMS specification (see Table 4.25). Retrospective misconceptions about the analogy were investigated by examining erroneous mappings recognised for each object in the source domain (see Table 4.25). Subjects had a good understanding of the six central analogical mappings (no errors with 4/6 mappings) however they generally did not understand other mappings. Most errors occurred while mapping the physical structure of the two domains, and incorrect analogical mappings involving 'production floor layout', 'product track' and 'track section' accounted for half (11/23) of all incorrect mappings. Six of these eleven errors involved incorrect mappings, so despite a clear and accurate specification, subjects seemed to have a confused picture of the physical structure of FMS and ATC domain, possibly due to the inexact mapping between the three-dimensional and two-dimensional domains. In addition 4 subjects were unable to differentiate between the physical structure of the domains and the logical structure of the flight and production plans, since their erroneous mappings involved a link between the physical components of the FMS domain with 'flight plan' or 'flight step'. In sum, these errors suggested that subjects had a poor analogical understanding of concepts beyond the 6 critical analogical mappings, in particular with the physical domain structures.

analogical mapping	subjects										total mapp-ings
	E1	E3	E4	E5	E7	E8	E9	E10	E11	E12	
aircraft/product	√		√	√		√	√		√	√	7
radar/infra-red sensors			√	√	√		√	√		√	6
air traffic controller\ production controller			√	√	√		√	√			5
air corridor/track			√	√		√	√				4
flight plan/product plan	√					√			√		3
similar processes									√	√	2
warnings in two systems				√							1
air space\machine				√							1
similar collision proces.					√						1
similar guiding process.					√						1
aircraft pos\product pos									√		1
both tracking systems		√									1

Table 4.24 - analogical mappings which were retrospectively recognised as important by subjects

analogical mapping		correctly mapped	incor rectly mapped
prompt in source domain	correct choice in target domain		
production controller	air traffic controller	9	0
infra-red sensors	radar	9	0
production plan	flight plan	8	0
production track	air corridor	7	2
product	aircraft	6	2
two products in same track section	two aircraft in same air space	6	0
manufacture of a product	flight	3	2
production floor layout	airways	2	4
misdirected product	aircraft off course	2	0
production track section	air space	1	5
job	flight step	1	2
production operator	pilot	1	3
machine	air space	1	1
delayed product manufacture	delayed flight	0	2
lost product manufacture	missing flight	0	0

Table 4.25 - retrospective understanding of analogical mappings, prompted from source domain mappings

Several subjects showed a better understanding of the analogy, so reasons for these improvements were investigated. The total number of correct analogical mappings identified retrospectively correlated with the total of reasoning utterances about the analogy (Spearman Rank Order Coefficient $r=0.831$, $p\leq0.005$), revealing that more analogical reasoning may have lead to greater analogical understanding. Analogical reasoning occurred primarily while assimilating and gathering information about the reusable specification. The behaviour of one subject (E10) also suggested that careful scoping and notetaking during assimilation and analogical understanding may have improved comprehension of the analogy. She made extensive notes to record analogical mappings (12 analogical mappings were recorded in narrative form) before reusing the data flow diagrams and exhibited the best retrospective understanding of analogical mappings.

Abstraction of key analogical concepts is necessary for analogical understanding, see chapter 3, Gick & Holyoak (1983) and Gick (1989). Subjects' understanding of these abstractions was investigated by retrospectively prompting for generic descriptions of each key analogical mapping identified by subjects. Abstractions are given in Table 4.26.

Subjects viewed both domains as objects moving in a space and following a spatial position plan, during which they were tracked by a remote monitoring device. A human monitor or controller supervised object movement. These retrospectively verbalised abstractions bear some resemblance to the logical domain abstractions defined in chapter 3 which underlie the ATC/FMS analogy, hence these findings lend some cognitive plausibility to the defined model of domain abstraction.

analogical mapping	subjects							
	E1	E4	E5	E8	E9	E10	E11	E12
aircraft/ product	main details	moving	fundamentls	tracked	contrlled objt	things moved		fundmtal entities
flight/ production plan	subsid. details			plan	permitted progress		monitored aganst	
production track/air corridor		fixed		geographical				
infra-red sensor/radar		monitoring	input		remote sensor			tracking & detcctn
air traffic production cntllr		human	monitor	controllr	progress monitor	controls		
product position/ac position							position info.	
processes							elements of control	

Table 4.26 - abstraction of perceived important analogical mappings retrospectively recognised by subjects

Finally, subjects' partial understanding of the analogy was investigated more closely using the same analysis as in study 3. Concurrent and retrospective findings were integrated to develop a composite model of the analogy. Reasoning behaviour during concurrent protocols was examined to identify relations between the mapped pairs of objects, similar to Gentner's structure-mapping theory (Gentner 1983). The resulting model is represented as an informal semantic network in Figure 4.19. The analogical mapping between aircraft and product was central to subjects' understanding of the analogy. As in study 3, subjects tended not to reason about the domain in terms of its physical structure. Rather, they verbalised utterances about the domain structure in the context of required functionality. The most commonly-verbalised functional relation was *Air Traffic Controller updates/changes the Flight Plan*, while only two purely structural relations were identified (*Aircraft flies in Air Corridor* and *Aircraft passes along Air*

Corridor).

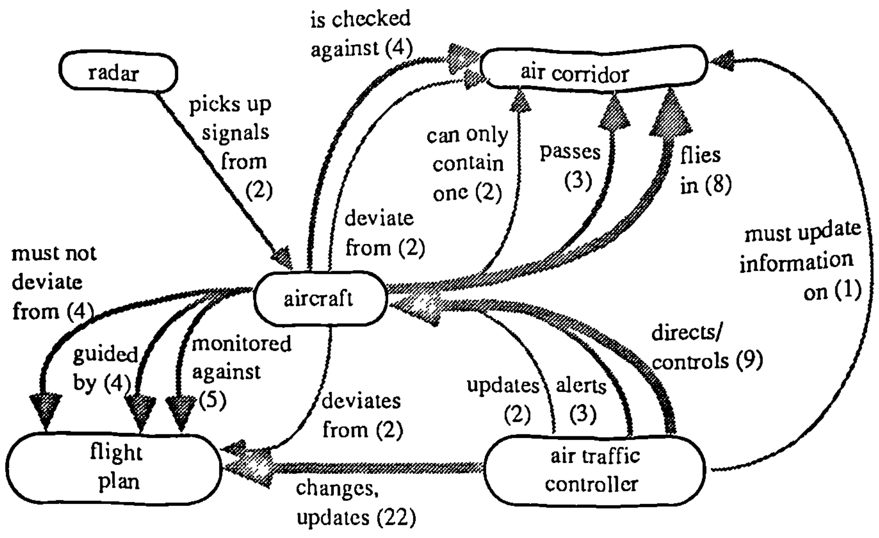


Figure 4.19 - composite expert mental model

To summarise, subjects only partially understood the analogy from 6 out of a possible 15 analogical mappings. They exhibited good understanding of the aircraft/product, radar/infrared sensor, flight plan/production plan, air traffic controller/production controller, air corridor/production track and aircraft collision/product collision mappings but showed poor understanding of other analogical concepts. Analogical abstractions revealed that subjects viewed the analogy as collision avoidance between objects which followed predetermined plans in a space. Differences between inexperienced and expert levels of understanding yielded surprising results. The novices' greater analogical comprehension may have been linked to guessing and less precision during retrospective questioning: they averaged over twice as many incorrect false mappings as experts during retrospective questioning (4.8 to 2), possibly indicating a greater willingness to complete the questionnaire.

4.4.2.7 Analogical Transfer

Transfer of the analogical specification occurred primarily during reuse, and solutions developed by 9 subjects were similar in size (average completeness score of 9 subjects was 63.67%) and layout to the reusable specification, so large scale transfer took place despite the analogy only being partially understood. The overview of analogical reuse given in the first chapter distinguished analogical comprehension from transfer. Concurrent protocols were reexamined to identify how quickly subjects understood the 6 central analogical mappings. Subjects only verbally recognised on average 2.58/6 mappings during information gathering compared with recognition of 4.5/6 mappings at the end of the reuse session. This suggests that subjects developed their understanding of important analogical mappings while both scoping and transferring the reusable specification.

It is intuitively appealing to link good analogical comprehension to effective analogical transfer, so transfer of the six central analogical mappings understood by most subjects was examined. Twelve components in the final solution were linked directly to the target concepts in the six mappings. However, subjects only included 65% of these components in their solutions, so good understanding of analogical concepts did not necessarily lead to their successful transfer. Similarly, successful transfer did not imply that the analogical concept had been learnt. For example, three subjects exhibited effective analogical reasoning to reuse the data flow from 'track section' to the monitoring processes. However, these subjects did not exhibit any retrospective analogical understanding of 'track section', suggesting that effective analogical transfer may have occurred without learning the underlying analogical concepts.

Finally, analogical transfer appeared to have an important influence on solution development. Each component included in seven or more of subjects' solutions could be matched analogically to a component in the reusable specification. On the other hand, the required solution contained several components without analogical matches to the reusable specification, and no more than three subjects recognised each of these unmapped components. Therefore, many subjects relied heavily on the analogical specification throughout specification development.

To summarise, successful expert reusers appeared to compensate for a partial analogical understanding by using the structure and contents of the reusable specification to guide analogical transfer of the components which were reused. Thus transfer proved to be important to solution development since many transferable components were included in most subjects' solutions. Finally subjects also appeared to develop an understanding of important analogical concepts during reuse, suggesting that analogical comprehension and transfer occurred in parallel.

4.4.2.8 Summary of Subjects' Behaviour

Most subjects exhibited an initial period of information gathering followed by reuse of the analogical specification, then solution testing against the target requirements or the reusable specification. During reuse they reasoned analogically to transfer reusable components using the structure of the reusable specification. Subject's behaviour was individually variable and they used different strategies to achieve analytic success. Reuse, construct and evaluate against the target strategies were all used to develop subjects' specifications, although individual differences existed in the use of these strategies. Subjects only partially understood the analogy from six analogical mappings and exhibited a poor understanding of mappings between the physical domain structures. Effective analogical transfer was achieved by following the structure of the reusable specification rather than by developing a detailed analogical understanding. Mappings triggered hypotheses which were developed and evaluated in the target domain. Such an approach was effective for developing a specification given the closeness of analogical fit in this study, however more effective analogical understanding may be necessary for a less exact fit.

Several determinants of good and expert analytic behaviour were identified:

• careful and painstaking reuse to fully exploit the reusable specification;
• long, uninterrupted bouts of reuse;
• reusing a specification so that the new solution is structurally similar to the reusable specification where appropriate;
• directly writing over the analogical specification during reuse so that the specification acts as a template for analogical transfer;.
• reasoning analogically during reuse of the specification;
• assimilating and understanding the analogy from a narrative describing critical concepts in the underlying problem domain; and
• thorough and detailed testing against the functional requirements of the target domain.

Furthermore better analogical understanding was related to more extensive reasoning about the analogy. Poor analytic performance in this study was linked to:

• mixing reuse with other analytic strategies to develop a specification;
• constructing solutions independently from the reusable specification;
• testing by summarising solutions and evaluating them against the reusable specification;
• redrawing solutions or summarising documents;
• failing to reason about analogical mappings.

Other results suggested that use of method knowledge to structure the problem space and the analytic process was not effective.

Subject behaviour was complex and individually variable, and no subject exhibited all determinants of good analytic performance. Individual differences in subject's behaviour were investigated in more detail by examining the protocols of two good quality subjects. These case studies are summarised in Appendix D.

4.4.3 Conclusions from Study 4

The three aims of this study were to elicit cognitive, reasoning and task models of effective analogical reuse and mental models of analogical understanding possessed by expert software engineers. The study was successful on all three counts.

Expert software engineers effectively recognised and transferred the ATC/FMS analogy. Analogical recognition was triggered by three well-understood analogical concepts while reuse strategies maximised the transfer and minimised omission of analogical components. There was no link between effective analogical comprehension and transfer, supporting findings reported by Novick & Holyoak (1991). However, unlike the inexperienced software engineers in the previous study, experts recognised problems associated with copying and attempted to understand the analogy prior to reuse. Surprisingly, complete analogical understanding appeared to be difficult even for experienced software engineers, despite their greater analogical reasoning and motivation for the task.

4.4.3.1 Effective Analogical Transfer

The cognitive task model of expert specification reuse revealed effective strategies for analogical transfer. Most expert software engineers adopted a 'try it and see' approach and quickly drew specifications to structure and scope the problem space using the analogical specification's layout and boundaries. Their reasoning processes during reuse were also guided by the analogical specification, which is not surprising given that method procedures and heuristics were inappropriate for supporting specification reuse. As such, analogical specifications may be one way of providing templates which guide software engineers' reasoning focus during the early stages of software design (e.g. Reubenstein & Waters 1991, Fugini et al. 1991).

The analogical specification structure proved effective for transferring major functional components although experts occasionally exhibited incorrect transfer of less-important reusable components, thus supporting Novick & Holyoak's (1991) finding that analogical transfer is difficult. Ineffective reusers reasoned more opportunistically, mixing reuse with testing, problem scoping and construction strategies whilst reasoning about specific topics. This contrasts with studies of planning and system design by Hayes-Roth & Hayes-Roth (1979) and Guindon (1989, 1990) which revealed externally-cued, opportunistic reasoning to be typical of successful, expert performance. However, ill-structured reuse in this study was linked to reasoning about more candidate reusable components. This may indicate that opportunistic reasoning is ineffective without well-formed mental models of the domain. Software engineers appear in need of guidance during specification reuse, thus the syntactic structure of analogical specifications can be exploited to guide their reasoning strategies.

4.4.3.2 Partial Analogical Comprehension

In contrast to their effective transfer strategies, software engineers failed to understand the entire analogy, and their mental models were built upon key object mappings which justified reuse of some but not all analogical components. This may indicate cognitive limitations when understanding complex analogies. Possible reasons for partial analogical comprehension are three-fold:

- software engineers filtered out many useful facts while reading the reusable specification and tended not to use external memory aids to record domain facts or analogical mappings, although such notetaking appeared necessary given the complexity of the analogy;
- software engineers also failed to determine key analogical mappings defined in chapter 3 and had a poor understanding of the key *air space/track section* mapping, which in turn may be due to syntactic differences between the *air space* and *track section* concepts, the obscurity of *track section* in the reusable specification, or both;
- the narrative domain description failed to provide software engineers with a good analogical understanding, a result also found by Gick & Holyoak (1983, 1989) who reported that spatial diagrams prompted analogical comprehension more effectively.

Results indicate that even experts were unable to assimilate and recognise all analogical cues or hold models of the source, target and abstraction concurrently in working memory, thus reducing the cognitive processing capacity available for schema acquisition (Sweller 1988). As such, analogical comprehension prior to specification transfer is naturally difficult and in need of tool support to overcome cognitive limitations.

4.4.3.3 A Cognitive Model of Analogical Specification Reuse

The cognitive model of analogical specification reuse shown in Figure 4.20 indicates how analogical reasoning occurs during specification reuse. An iterative cycle of hypothesis generation, development and testing occurs throughout the model. Analogical reasoning is divided into two parts. First, an analogical understanding is developed whilst assimilated source domain data from documentation, then this analogical understanding is validated during transfer of the specification to identify the extent of the analogical match. During the initial phase, information gathering about the specification leads to recognition of analogical mappings and development of a mental model of analogical understanding. Subsequent validation and transfer of the analogy is driven by the reusable specification. Analogical cues generate target hypotheses which are developed and evaluated in the target domain. As such, initial analogical mappings activate further mappings when reasoning about other components linked to the mapped component. This analogical validation and transfer continues until each analogical specification component has been transferred or rejected.

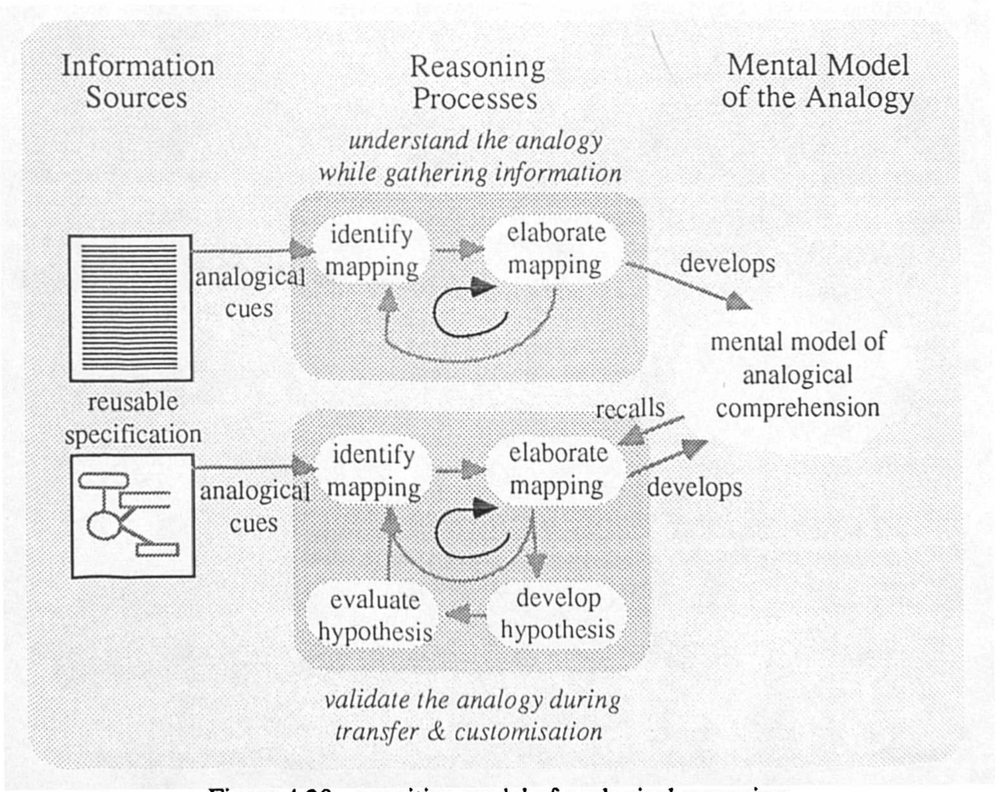


Figure 4.20 - cognitive model of analogical reasoning during specification reuse

4.4.3.4 Cognitive Theories of Analogical Reasoning

Existing theories of analogical reasoning differ according to the types of knowledge mapped across domains. Our findings support several of these models, however they also indicate that analogical reasoning may be more complex. For instance, retrospective analyses revealed that many software engineers developed an abstract model of the analogy, supporting Greiner's (1988a, 1988b) theory that analogical domains must belong to the same domain class. Within this abstraction, aircraft and products were identified as system objects while flight plan and production plan were both plans. Software engineers also mapped several analogical system goals (e.g *monitor collision*), suggesting that purpose is important for the recognition and comprehension of analogies (Gick & Holyoak 1983, Kedar-Cabelli 1988). They also exhibited evidence of Gentner's (1983) structure-mapping theory by transferring interconnected object-relation structures. Indeed, the syntactic structure of the reusable specification also provided an important basis for analogical transfer, indicating that two types of structural transfer occurred. This greater analogical complexity may have been due in part to the small-scale problem analogies investigated previously. On the other hand, Russell (1989) suggests that analogical reasoning is domain specific rather than being determined by simple measures of similarity between source and target analogs. These results indicate that effective analogical comprehension is dependent upon understanding key domain abstractions. As such the empirical findings from this study support the logical model of software engineering analogies defined in chapter 3.

4.4.3.5 Contrast with Analogical Specification Reuse by Novices

Studies 3 and 4 were not intended to be an experimental study of expert/novice differences, however some comparisons are worth making. First, notable differences existed in software engineers' attitudes towards specification copying, although both groups exhibited similar degrees of analogical understanding. Second, expert software engineers were better information gatherers while novices tended to exploit reusable solutions and ignore other knowledge sources which were not directly transferable. Third, experts reasoned more extensively about both source domain and the analogy, suggesting a greater desire to understand the FMS domain despite the difficulties encountered. Indeed, many of the differences identified between expert and novice software engineers indicate that the experts were more motivated to understand the analogy before reuse. In spite of these differences, the difficulties encountered during large-scale analogical understanding by both inexperienced and expert software engineers may be considerable, indicating the need for similar forms of extensive tool support for expert, intermediate and experienced software engineers alike.

4.4.4 Implications for Support Tools

An integrated view of the cognitive reasoning, task and mental models is shown in Figure 4.21. Implications for support tools are three-fold. First, the cognitive task model determines effective strategies for analogical reuse. Second, the cognitive reasoning model has implications for controlling hypothesis level interaction within analytic strategies such as information gathering and reuse. Finally, the mental model of analogical comprehension indicates the extent of analogical understanding, with implications for the reasoning focus during analogical comprehension and transfer. Design implications from each of these three models are examined more closely.

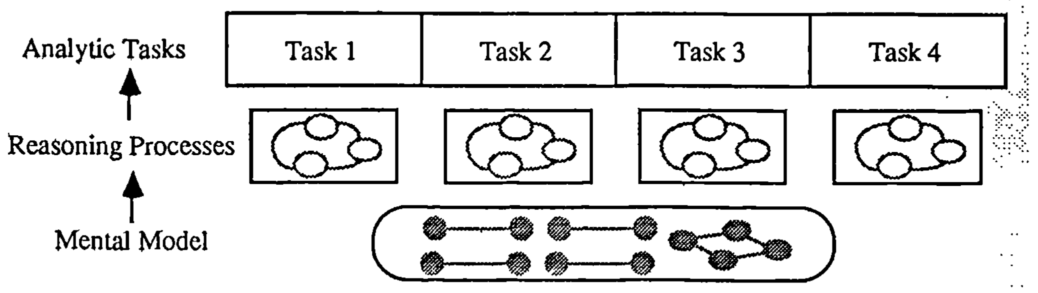


Figure 4.21 - links between the cognitive reasoning, task and mental models of analogical understanding

4.4.4.1 Design Implications from the Cognitive Task & Mental Models

The cognitive task and mental models of successful, expert analogical reuse influenced design of support tools in two ways. First, the tool must encourage inexperienced software engineers to follow successful expert reuse strategies, thus simulating their performance. Second, tool support must incorporate additional strategies to assist software engineers to overcome the comprehension difficulties encountered even by experts. The functional requirements include the following observed strategies for analogical comprehension and transfer:

- encourage some analogical comprehension prior to transfer;
- understand the analogy from the reusable specification and descriptions of key facts about the source domain;
- prompt additional analogical reasoning about mappings with reusable components which share notable syntactic similarities with the target domain;

- strong guidance to ensure full exploitation of the analogy;
- transfer each component and all its neighbours in turn;
- long, uninterrupted bouts of transfer to maximise analogical reuse;
- use the analogical specification as a template to guide the reuse process, directly modifying the reusable specification when required and ensuring that both solutions have a similar structure. As such the reusable specification provides a stencil for spreading activation and hypothesis triggering during analogical transfer;
- evaluate the transferred specification against functional requirements of the target system, to identify omissions from transfer;
- evaluate the transferred specification against the analogical specification by promoting further analogical reasoning between the old and new solutions.

Inferred strategies to augment observed behaviour in this study are:

- provide spatial diagrams to encourage visualisation of the analogy and aid mental induction of key domain abstractions (Gick & Holyoak 1983, Gick 1989);
- provide familiar analogies of common, everyday objects or problems to also aid mental induction of key domain abstractions;
- promote greater analogical reasoning by explicitly defining analogical mappings using the advisor. Salient syntactic triggers are likely to provide the most effective prompts for analogical mapping;
- hide specification components not relevant to the current reuse task, to concentrate software engineer's reasoning and discourage copying and opportunistic reasoning. Partial exposure to the full functionality has been found to be effective elsewhere (e.g. Carroll et al. 1988).

The mental model of analogical understanding also has implications for guiding the software engineer's reasoning focus. Software engineers failed to understand all key domain facts, so domain abstractions must be explained early during reuse to focus reasoning on critical analogical determinants. Software engineers must also be guided to reason analogically about all other candidate mappings. Assistance will be needed since the software engineers in this study exhibited difficulties when reasoning analogically about physical structures which are not perfectly matched.

4.4.4.2 Design Implications from the Cognitive Reasoning Model

The cognitive reasoning model has implications for controlling detailed reasoning by the software engineer within analytic strategies such as information gathering and reuse. The proposed reasoning planner shown in Figure 4.22 must encourage iterative and incremental analogical reasoning and integrate explanation of different knowledge types into the proposed iterative reasoning cycle. Indeed, the meta-schema of knowledge types defined in chapter 3 indicates that software engineers are required to learn multi-layered facts about an analogical match. Analogical reasoning while gathering information from the specification is driven by key types of knowledge to be explained. State transitions with respect to a structure are central to the proposed model, so mappings with state transitions and object structures should be explained first, followed by other knowledge types such as preconditions on state transitions and object types. Explanation of each knowledge type can be layered by depth, degree of causality and reliance on the domain abstraction to provide complex and variable explanation of the analogical match. These explanations will be coupled with gradual exposure to key domain abstractions and, if necessary, simple examples of these abstractions. Additional explanations will also be presented to correct analogical misconceptions and erroneous mappings. Unlike analogical comprehension, the topic focus during transfer will be driven by candidate components for reuse. The reasoning planner will explain each reusable component topic at several levels of complexity. These increasing levels of explanation include presentation of the underlying analogical mapping, explanation of mappings between linked domain objects, explanation of the mapping in the context of the domain abstraction and presentation of simple examples of the domain abstraction.

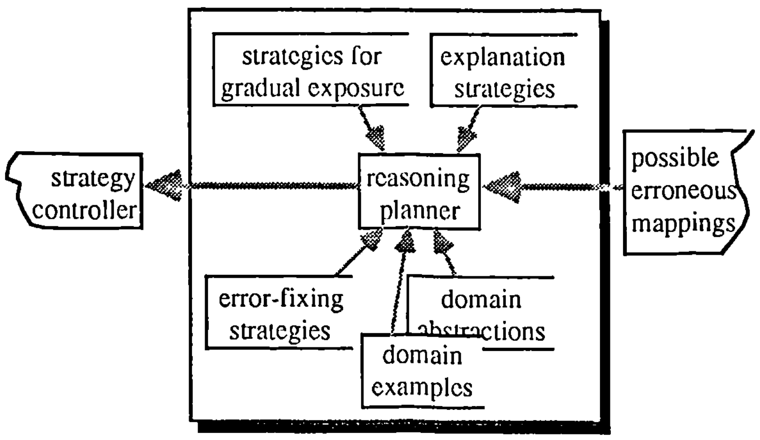


Figure 4.22 - empirically derived design of the reasoning planner

To sum, the analogical reasoning, comprehension and transfer strategies derived from this study demonstrate the potential benefits from empirically-derived requirements for software tools. These findings are integrated into an empirically-derived design for support tools outlined in the remainder of this chapter and expanded on in chapter 5.

4.5 An Empirically Derived Design for the Intelligent Advisor

Implications for design of the problem identifier and specification advisor are examined in the remainder of this chapter. The overview of the intelligent reuse advisor incorporating them is shown in Figure 4.23.

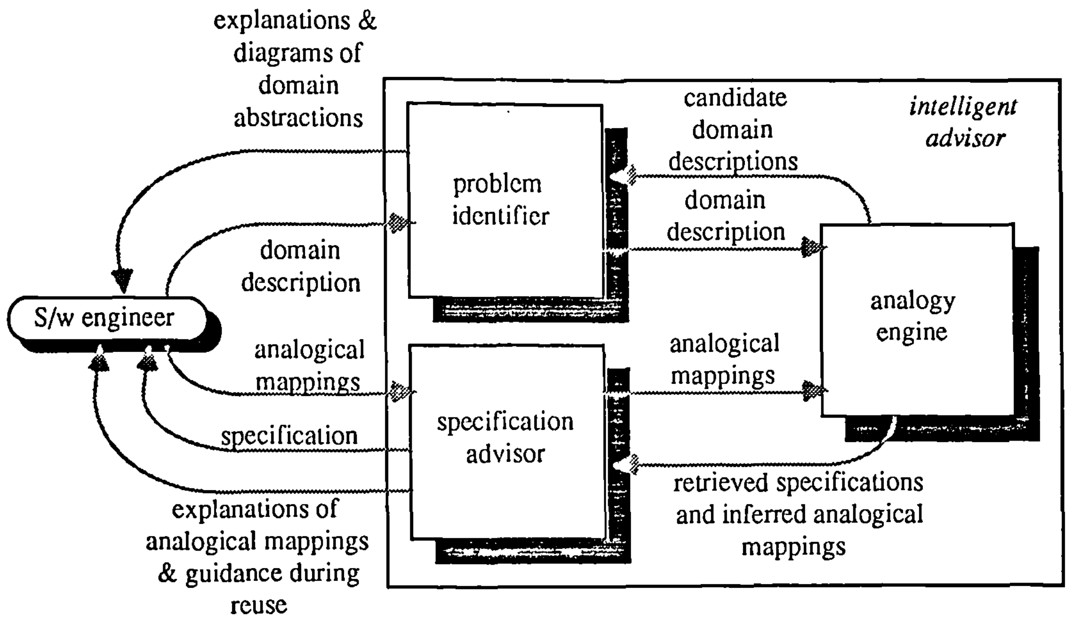

explanations & diagrams of domain abstractions
S/w engineer
domain description
analogical mappings
specification
explanations of analogical mappings & guidance during reuse
problem identifier
specification advisor
candidate domain descriptions
domain description
analogical mappings
retrieved specifications and inferred analogical mappings
analogy engine
intelligent advisor

Figure 4.23 - overview of the intelligent reuse advisor

4.5.1 Implications for Design of the Problem Identifier

Inexperienced software engineers in the first study lacked the necessary domain knowledge to develop a specification and were unable to structure and scope new problems, specify detailed functional requirements or evaluate the resulting specifications. Therefore, the problem identifier must interact with the software engineer to acquire and explain key target domain facts prior to their input to the specification retrieval mechanism. This is achieved by providing the domain knowledge necessary during fact acquisition to facilitate problem understanding, scoping and identification of key domain facts (Curtis et al. 1988). Presenting software engineers with appropriate domain abstractions and real world instances of these abstractions early in the fact acquisition dialogue can assist domain scoping, structuring and evaluation, similar to the approach implemented in the CODEFINDER system (Fischer et al. 1991a). Languages and notations are also needed to model the new domain in terms of these abstractions and examples, thus encouraging the model-based reasoning which appeared to be one determinant of better analytic performance. The failure of structured method notations to represent key fact types defined in the meta-schema emphasises this need for alternative representations.

More generally, software engineers reasoned effectively with diagrams, indicating that they may have led to better mental model formation. Indeed, fact recording with diagrams was linked to more effective analytic performance while software engineers made little effective use of procedural heuristics. Although method knowledge was used by inexperienced software engineers in the first three studies it appeared to hinder rather than improve their analytic performance. Rather, inexperienced software engineers' reasoning focus appeared to be triggered by the narrative, so domain knowledge had greater influence on their analytic approaches.

The need to retrieve domain abstractions, combined with software engineers' failure in the first study to specify the domain correctly, indicates the need for semi-automated fact acquisition strategies. Dialogues can also be structured to capture facts in an order which helps mental model formation. Controlled fact gathering differs from other intelligent requirements engineering toolkits (e.g. Reubenstein & Waters 1991, Harandi & Lee 1991) which permit free form entry of facts. Indeed, a more structured approach to fact acquisition and modelling may be critical for effective description, retrieval and explanation of domain abstractions. An overview of the problem identifier's architecture is shown in Figure 4.24 to demonstrate how the reported empirical findings can inform its design.

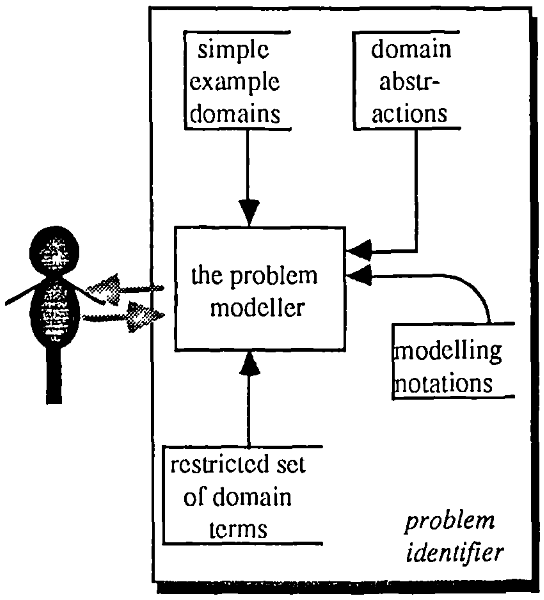


Figure 4.24 - empirically based architecture of the problem identifier

4.5.2 Implications for Design of the Specification Advisor

The specification advisor is intended to assist software engineers to understand and transfer analogical specifications based on empirical findings reported in this chapter. Findings from the second and third studies identified problems encountered by inexperienced software engineers while the fourth study identified successful comprehension and transfer strategies. Most implications for the specification advisor's design are reported in previous sections and not repeated here. On the other hand, the second and third studies indicated the need to overcome mental laziness during reuse. Malrules representing misconceptions during analogical reuse can diagnose software engineers' analogical understanding, thus permitting more accurate support. Integrated designer's notepads (e.g. Haddley & Sommerville 1990) can extend working memory to assist software engineers' reasoning about the target, source and abstraction domains and develop a better analogical understanding. This external memory could also provide the advisor with important knowledge about the software engineer's analogical understanding so that errors can be corrected more easily. An overview of the specification advisor's architecture is shown in Figure 4.25. It incorporates the impact of the empirical studies on the design of its two major components during support of analogical comprehension and transfer.

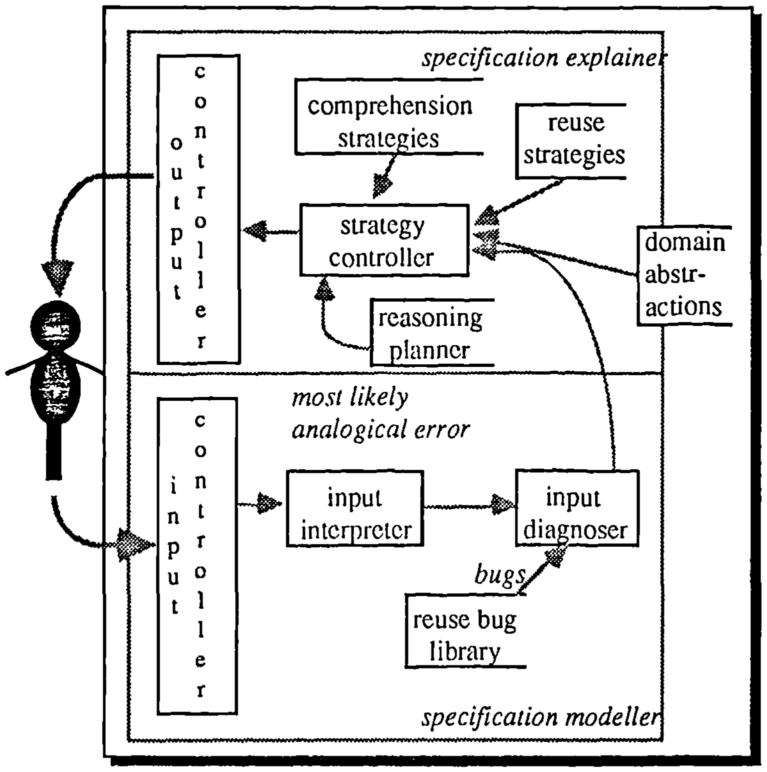


Figure 4.25 - an overview of the architecture of the specification advisor

4.6 Chapter 4: A Summary

This lengthy chapter reported empirical findings from four studies of analytic and analogical reuse behaviour. The cognitive task, reasoning and mental models inform the intelligent advisor's design so that it meets the true needs of its target users. These empirical findings lay the foundations for the advisor's architecture. Surprisingly, expert software engineers in study 4 were also unable to understand the analogical match effectively, so the proposed tool design may also aid analogical specification by experienced software developers. The advisor's architecture will be extended and developed throughout chapter 5 to specify how support tools interact and reason with software engineers during the retrieval, understanding and customisation of analogical specifications.

Chapter 5

5: The Intelligent Reuse Advisor (Ira)

This chapter specifies the intelligent reuse advisor (Ira) which assists inexperienced software engineers during analogical specification reuse. The advisor's analogical expertise is derived from a computational implementation of the logical model of software engineering analogies and its interactive components are founded on empirically-derived findings of software engineering and reuse behaviour. Its design also borrows concepts from computational models of analogical reasoning (e.g. Falkenhainer et al. 1989, Hall 1989) and analogical problem solving (e.g. Gick & Holyoak 1983).

Cooperative problem solving by humans and machines has been proposed elsewhere (e.g. Roth et al. 1988), for instance Kolodner's (1991) model of case-based reasoning proposed tool-based retrieval of analogical cases for application by the user. Similarly, effective analogical reuse requires cooperation between the software engineer and the advisor (Cumming & Self 1989), as opposed to traditional intelligent tutoring systems which teach subjects using their expertise of well-understood domains (e.g. Sleeman & Brown 1982). The division of work built into the advisor's design is intended to make the most of the domain knowledge and analogical reasoning capabilities possessed by the software engineer. Support for software engineers is based on the cognitive task, reasoning and mental models of analogical specification reuse derived empirically in chapter 4. The advisor undertakes the following roles during specification retrieval, understanding and transfer:

- assisting the acquisition of key domain facts prior to retrieval of specifications;
- computationally matching and retrieving candidate analogical specifications for reuse;
- explaining key analogical constructs to the software engineer to ensure effective analogical understanding. Explanations exploit a computational implementation of the logical model of software engineering analogies defined in chapter 3;
- diagnosing the software engineer's incorrect analogical beliefs to inform explanation of the specification. Diagnosis is founded on the model of analogical errors derived from the third empirical study in chapter 4;
- assisting the software engineer to understand and transfer specifications effectively. Strategies are derived from the expert understanding and reuse strategies derived empirically from the final study in chapter 4.

Chapter 5 is divided into four main sections. First, a brief overview of the current state of the art in intelligent tutoring system (ITS) research is given to situate the advisor. The remaining sections describe how Ira supports analogical recognition, comprehension and customisation during specification reuse. A partial prototype of this intelligent advisor incorporating two of these components has also been implemented and is described in chapter 6.

5.1 Previous Research of Intelligent Tutoring Systems

Traditionally, intelligent tutoring systems (ITS) consist of three modules: a student module, a didactic module and an expert module. The current state of ITS research is briefly examined in terms of these three models, then implications for design of the advisor are outlined.

5.1.1 Student Models

Development of student models has bourn the brunt of ITS research (e.g. Johnson 1990, Escott & McCalla 1988, Rizzo et al. 1988). Until recently it was proposed that intelligent, adaptive tutoring requires some dynamic representation of the current knowledge state of the individual student, known as the *student model* (VanLehn 1988, Payne 1988, Nwana 1991). However, difficulties inherent in student modelling compound problems encountered during error diagnosis. Most diagnosis involves inference of unobservable behaviour, so application of thorough diagnostic techniques such as model-tracing (Skwarecki 1988, Anderson 1988) and reconstruction (Johnson 1990) in analogical specification reuse is unrealistic. However, more recent research by Self (1988) suggests that it is not essential that ITSs possess precise student models, for several reasons. It may be better to develop interactions which unobtrusively tell the tutor what it needs to know, and only diagnose what the tutor can treat, to avoid unnecessary effort. Furthermore, do not feign omniscience, but adopt a 'fallible collaborator' role to develop a working partnership between tool and user. Ira has incomplete domain knowledge, so it cannot have complete expertise in the domain and must cooperate with the software engineer. Indeed, this expertise must be constrained to key analogical mappings rather than complete knowledge of complex domains.

5.1.2 The Didactic Model

Didactic models implement pedagogical activities intended to have a direct effect on the student, i.e. they represent teaching styles and strategies (see Halff 1988 for an overview). Unfortunately, computational models of didactics have been incomplete (Wenger 1987, Derry et al. 1988) while empirical studies of expert teachers have been few and far between. One alternative approach is to specialise more general theories to analogical specification reuse, for example the Minimalist approach (Carroll et al. 1988, Rosson et al. 1990) has successfully been applied to state-of-the-art training manuals and novice users of the smalltalk object-oriented programming language, with implications for comprehension and reuse strategies incorporated into the advisor. However, most support for the software engineer will be founded on the cognitive task and reasoning models defined in chapter 4.

5.1.3 The Expert Module

Anderson (1988) claims that there are two places for intelligence in an ITS. The first is in the principles by which it tutors and the method by which it applies these principles. The second is in its knowledge of the subject domain. Anderson describes the expert module as the backbone of an ITS, and a powerful expert must have an abundance of knowledge, for example STEAMER (Hollan et al. 1984) and the LISP tutor (Anderson et al. 1990). Representation of complex knowledge structures in ITSs has received research interest in its own right (Wenger 1987, Woolf 1988, Woolf et al. 1988, McCalla & Greer 1988, White & Frederiksen 1990). However, as Anderson and others admit (Cumming & Self 1989, Nathan 1990), empowering ITSs with sufficient knowledge of non-simple domains is a near-impossible task. Thus, Ira's expertise and assistance is limited to knowledge of: (i) generic domain classes represented by known abstractions; (ii) source domain descriptions, and (iii) mappings between analogical source, target and domain abstractions.

5.1.4 Intelligent Tutoring Systems: A Summary

ITSs have mainly been developed for teaching novices about small, well-defined domains such as algebra or LISP programming. Applying ITS principles to a complex, poorly-understood task is a challenging use of technology which breaks with some ITS conventions. Some ITS concepts will be applied to the advisor as needed (e.g. Anderson's cognitive principles in the design of computer tutors), however, many features incorporated into the advisor are based on research described in the thesis and findings reported in other fields.

5.2 Ira's Architecture

Ira has three main components which support the retrieval, selection and customisation of an analogical specification belonging to the same domain class (see Figure 1.6). Each are examined in this chapter:

- the problem identifier obtains a description of a new target domain and explains domain abstractions retrieved by the analogy engine so that the software engineer can classify the new domain;
- the specification advisor controls interaction with the software engineer during selection and customisation of retrieved analogical specifications by explaining analogies to the software engineer and guiding them to ensure that the analogical specification is fully exploited;
- the analogy engine reasons with key facts acquired by the problem identifier to retrieve and match specifications. Its reasoning capabilities also allow the specification advisor to reason alongside the software engineer to support explanation of the analogy during customisation.

5.3 The Problem Identifier

The problem identifier has two aims: (i) to acquire key facts about the target domain which can be matched successfully by the analogy engine, and; (ii) to explain retrieved domain abstractions to software engineers, see Figure 5.1. An incremental, example-based domain definition and retrieval paradigm is proposed to achieve these aims, similar to that proposed by Fischer et al. (1991b) and Fischer & Reeves (1991). During an interactive session, domain description and explanation of retrieved abstractions occurs iteratively, thus leading to a gradual refinement of the domain description.

The first study in chapter 4 suggests that software engineers encountered difficulties when identifying key domain facts which may be overcome by tool-based, semi-automated assistance. In the proposed scenario, key facts must be specified using the set of domain terms which define the meta-schema of knowledge types in chapter 3. However, people find it difficult to abstract unfamiliar problems, so unsupported use of these domain terms may be problematic. For instance, it has proven difficult to get an agreed set of terms to describe computer artifacts (Furnas et al. 1987). Therefore, one of the problem identifier's main aims is to ensure effective domain description using abstract terminology which can also be matched to retrieve domain abstractions. An alternative, more user-friendly option could involve application-dependent lexicons of terms derived using extensive domain analysis. However, initially deriving this domain knowledge is difficult and time consuming for reasons outlined in chapter 2.

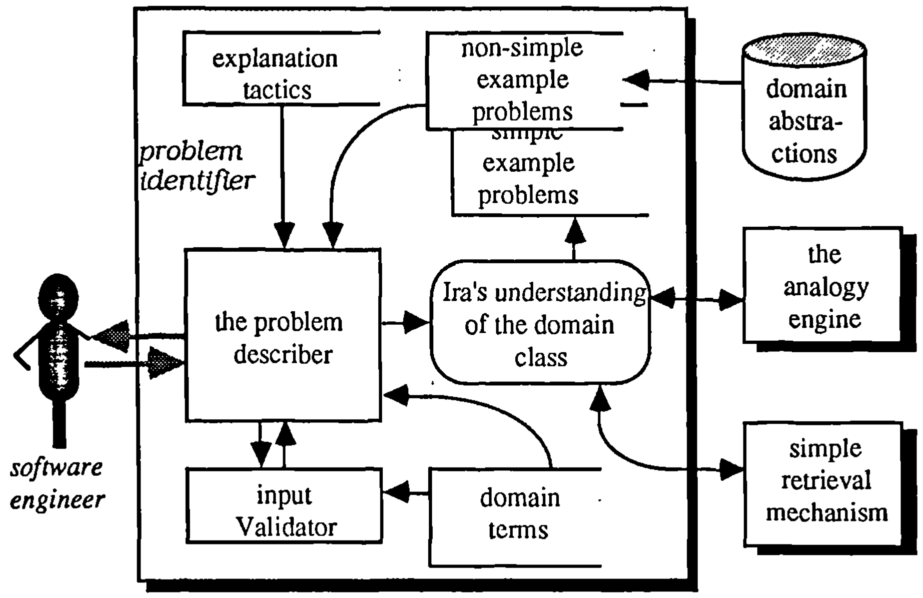


Figure 5.1 - architecture of the problem identifier

Understanding and representing the new domain is assisted by the early retrieval of domain abstractions. However, tool-based explanation of retrieved domain abstractions is also needed since software engineers reported in chapter 4 were unable to understand generic templates while previous empirical studies (e.g. Gick & Holyoak 1983, Gick 1989, Gilmore & Green 1988) also revealed that understanding abstract concepts is difficult. Thus, explanation of unfamiliar domain abstractions during the early phases of requirements engineering appears necessary to assist effective fact capture prior to specification retrieval.

5.3.1 Tactics Employed by the Problem Identifier

The problem identifier incorporates empirically- and theoretically-derived tactics to overcome the difficulties reported in chapter 4 and capture a sufficiently complete and correct domain description for retrieving analogical specifications. The main aim of these tactics is to encourage induction of relevant domain abstractions by software engineers, thus assisting them to scope and structure the domain and identify its key facts, see Figure 5.2. Tactics include example-based explanation, visualisation of abstractions and information hiding. Each is examined more closely.

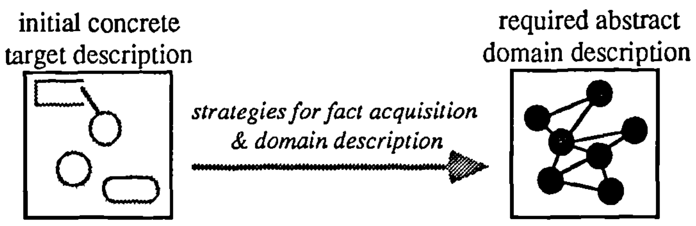

initial concrete target description
strategies for fact acquisition & domain description
required abstract domain description

Figure 5.2 - strategies as a transformation between an incomplete target domain description and required complete abstract domain description

5.3.1.1 Example-Based Explanation

Example-based explanation of domain abstractions (e.g. Breuker 1988) can lead to schema induction, for instance Gick & Holyoak (1983, 1989) revealed that schema induction only occurred when two analogical problem instances were available during problem solution. Example-based categorisation has also been shown to be an effective retrieval mechanism. Fischer et al. (1991a) studied information access mechanisms for poorly understood concepts which are continuously elaborated upon and evaluated until appropriate cues are constructed. People thought about categories of things in terms of prototypical examples as opposed to formal or abstract attributes, a result also identified by Rosch et al. (1976). Elsewhere example-based information access has been applied to create a cooperative relationship giving users the ability to incrementally improve a query by critiquing the results of previous queries (Fischer & Reeves 1991). In the problem identifier, analogical examples of retrieved domain abstractions permit similar cooperative retrieval, incrementally improving the software engineer's understanding of the target domain and narrowing the analogical search space, see Figure 5.2. This paradigm blurs the distinction between domain description and retrieval, suggesting that incremental retrieval and explanation of domain abstractions may be an effective means of retrieving analogical specifications.

5.3.1.2 Visualisation of Analogical Examples

Visualising domain abstractions and analogical examples using spatial diagrams is another empirically demonstrated tactic for analogical comprehension and schema induction. Gick & Holyoak (1983, 1989) reported that spatial diagrams representing key abstractions aided people's understanding of analogical problems while text descriptions did not lead to effective analogical comprehension. Spatial diagrams are relevant to analogical reuse since they can be used to represent key state transitions and object structures in the proposed meta-schema (see Appendix A). Visualisation of abstractions also permits them to be understood in terms of simpler well-known analogies, for instance many domains may be instantiated as simple blockworlds (e.g. Gupta & Nau 1991) which involve the movement of objects or blocks from one position (or state) to another. Domain visualisation can also assist during the acquisition of key facts. Software engineers can define key state transitions and object structures in diagrammatic form, thus encouraging a representation of the new domain which is similar to the retrieved domain abstractions and analogical examples.

5.3.1.3 Explanation of Analogical Examples

Unfortunately, visualising concrete domain examples alone is unlikely to ensure that domain abstractions are understood. Therefore, description and justification of retrieved domain abstractions and knowledge types defined in the meta-schema are proposed (e.g. Breuker 1988, Wenger 1987), supported by evaluative tactics such as elicitation and comment. For instance, the domain terms *resource* and *resource-container* cannot be explained fully in terms of larger analogical examples, so detailed explanation of each domain term is required.

5.3.1.4 Additional Tactics

Analogical copying may occur when attempting to understand analogical examples, so additional tactics are incorporated into the problem identifier to avoid mental laziness. First, information hiding combined with controlled access to windows for defining new domain terms can discourage copying of analogical examples. Second, a single, non-analogical example is presented to demonstrate how these terms describe domains. Finally, summary descriptions of the domain are presented to encourage its evaluation. For example, when evaluating the theatre domain, the state transition linked to allocating theatre bookings to seats only occurs if both the seats and the booking match, so the following paraphrase may be displayed to the software engineer for agreement or rejection:

```
Ira believes that ALLOCATION of BOOKING to THEATRE SEAT only occurs when:
        BOOKING and THEATRE SEAT have the same properties.
```

Thus, the problem identifier explains some of its analogical reasoning during evaluation of the domain description. In addition, immediate feedback on analogical errors can correct false mappings and incorrect reuse before it has a chance to occur. Immediate feedback on errors would appear to be an important tactic since there was little evidence of self-error correction by inexperienced software engineers during analogical reuse.

5.3.1.5 Simple Retrieval Mechanism

Effective domain abstraction requires early retrieval of analogical examples, thus a second, simpler retrieval mechanism is needed to intuitively guess relevant domain abstractions and retrieve class-inclusive examples. Retrieval is achieved by matching information system functions and domain events to a lexicon of functions and events describing domain abstractions. For example 'allocate' is semantically equivalent to 'assign' and 'place' functions in the theatre reservation domain. Function/event matching was incorporated into the search mechanism because these functions and events were the only key knowledge types recognised effectively by inexperienced software engineers in the first study of chapter 4. Every abstraction is also defined using three application-independent terms which also describe the new domain. For instance, 'constraint satisfaction' correctly defines both the theatre domain and its abstraction. Lexical matching in this retrieval mechanism was felt to be acceptable since both functions and domain terms were based on the logical model of software engineering analogies defined in chapter 3.

5.3.2 Ira's Fact Acquisition Dialogue

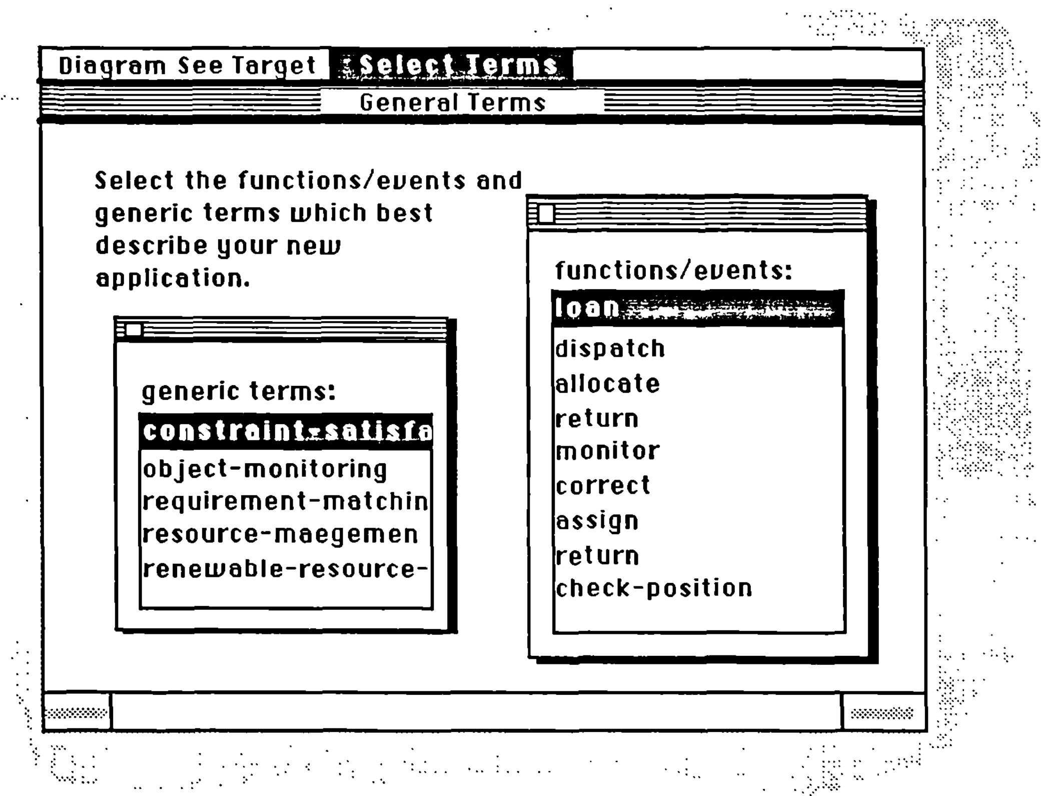
The problem identifier incorporates example-based explanation, domain visualisation, information hiding and summary descriptions into a fact acquisition dialogue which captures all key facts about a domain. The dialogue has five phases:

- acquisition by the advisor of key system functions, domain events and general domain terms as input to the simple retrieval mechanism;
- retrieval and explanation of the two best-fitting domain abstractions to the software engineer using example instances, familiar physical analogies and visualisation of the domain abstraction. These explanations aim to induce a mental model of the domain abstraction;
- definition of key state transitions and object structures in diagrammatic form by the software engineer;
- further definition of the domain using text descriptions. Candidate domain terms are explained by description, justification and non-analogical examples;
- evaluation and modification of the domain description before passing it to the analogy engine.

Furthermore, the first phase of this dialogue may be preceded by problem assimilation and understanding using structured analytic techniques such as functional decomposition and data flow diagramming. The main features of the fact acquisition dialogue are demonstrated during a mocked-up description of the theatre domain described in chapter 1.

5.2.3.1 Acquire Key Facts to Retrieve Candidate Abstractions

The problem identifier provides the software engineer with scroll menus to select functions and general domain terms, see Figure 5.3. For the theatre domain, software engineers may be expected to select the function *allocate* and the general terms *requirements matching* and *constraint satisfaction* during correct specification of the new domain.



Select the functions/events and generic terms which best describe your new application.

Figure 5.3 - screen design of a fact acquisition dialogue, demonstrating the selection of functions and domain descriptors for the theatre domain

5.2.3.2 Present Domain Abstractions and Concrete Examples

The software engineer is encouraged to understand domain abstractions retrieved by the simple matching mechanism. Concrete examples and familiar analogies are represented graphically to encourage the software engineer to visualise the analogical match.

Concrete examples represent well-understood software engineering domains while the analogies include many everyday, non-software engineering objects, domains or situations, a selection of which is shown in Figure 5.4. Throughout this learning process the software engineer can request further examples to assist their analogical understanding. Examples of the retrieved domain abstraction and analogy for the theatre domain are shown in Figure 5.5. As well as encouraging mental schema induction, analogical examples demonstrate the concise range and complexity of the defined domain terms.

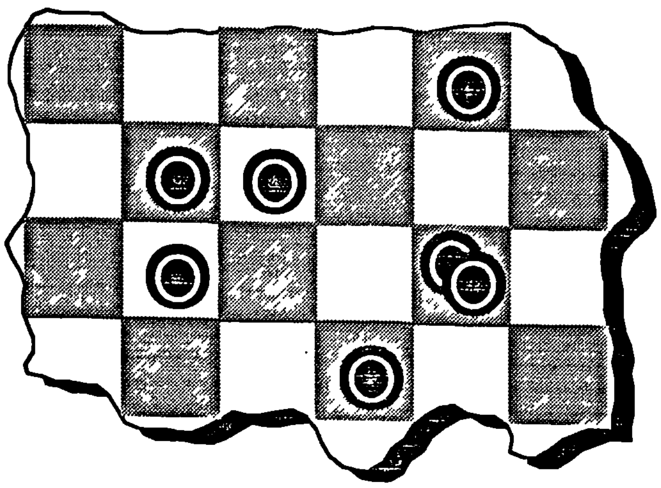


Figure 5.4(a) - example analogy of the object monitoring and object positioning domain abstractions (see Appendix A). A checkers board, representing the concepts of structured space and objects in space, critical to both the ATC/FMS analogy and analogies founded on the object positioning domain abstraction (see Appendix A). In the former domain abstraction two objects should not be positioned in the same space, while rules which explain the object positioning domain include *no two adjacent spaces should be left unoccupied.*

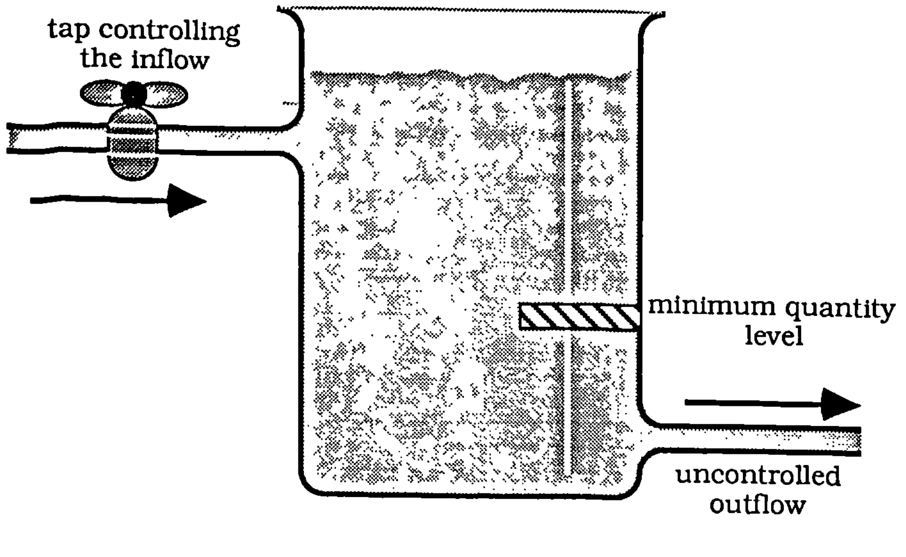


Figure 5.4(b) - example analogy for object containment abstraction (see Appendix A). A water beaker has contents which are controlled in an inflow and an outflow, analogous to an object containment abstraction instantiated to many forms of stock control. An important concept is that of a minimum level of contents in the beaker (store).

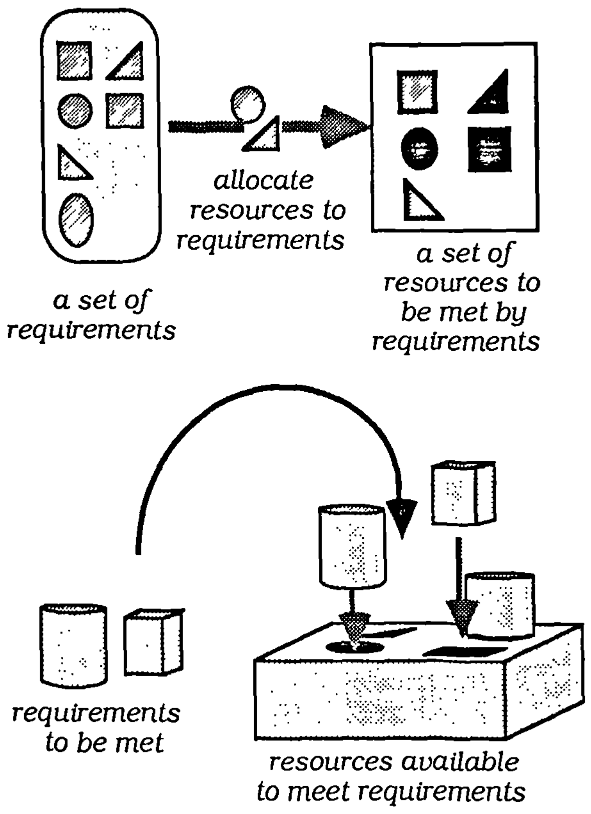
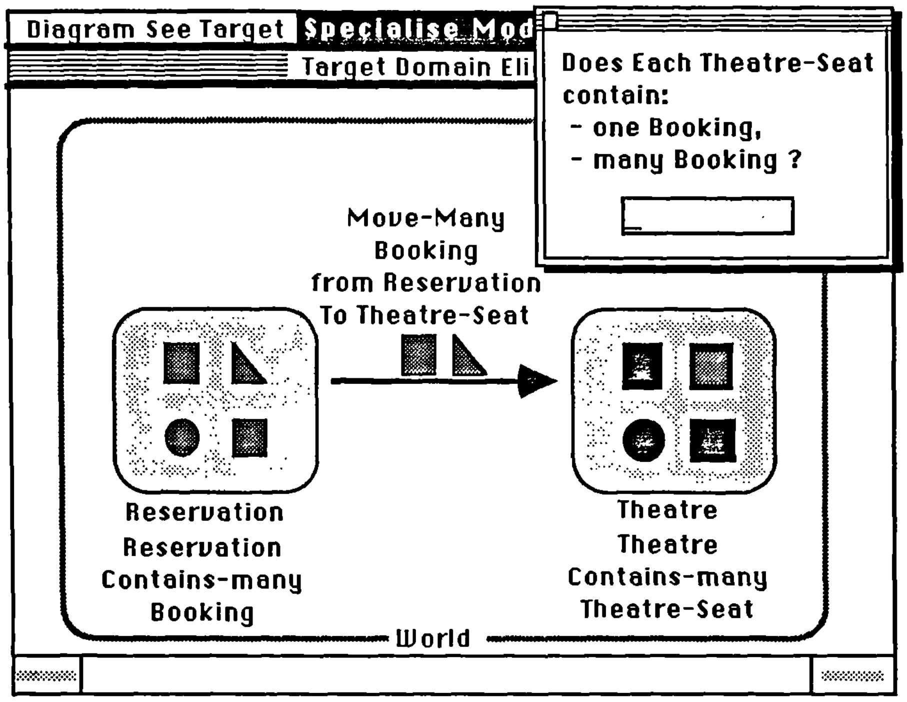

allocate resources to requirements
a set of requirements
a set of resources to be met by requirements
requirements to be met
resources available to meet requirements

Figure 5.5 - abstract and concrete analogical representation of the key features of the theatre and university course administration examples (see Appendix A)

5.2.3.3 Develop a Pictorial Description of the Target Domain

The software engineer enters key state transitions and object structures in diagrammatic form, although some textual entry is also necessary, see Figure 5.6. Direct manipulation palettes can be used to select domain objects, then terms for elaborating these object structures and state transitions can be selected from scroll menus. For instance, text-based menus are needed to determine whether there are one or many theatres in the world, see Figure 5.6.



Does Each Theatre-Seat contain:
- one Booking,
- many Booking ?

Figure 5.6 - screen design of the fact acquisition dialogue, demonstrating the need for text fact entry during diagrammatic definition of the theatre reservation domain

5.2.3.4 Develop a Text Definition of the Target Domain

The remaining knowledge types in the meta-schema are best defined in text form. The software engineer selects appropriate domain terms from menus, supported by explanations of these terms and a single non-analogical example demonstrating their use. Furthermore, the problem identifier uses the result of its initial match with a domain abstraction (see section 5.2.3.2) to present hints about the new domain to the software engineer. The problem identifier also encourages the software engineer to evaluate their entered domain description, so lessening the likelihood of mental laziness during fact acquisition.

5.2.3.5 Evaluation of the Domain Description

Domain descriptions are evaluated by both the advisor and the software engineer. The problem identifier matches the domain description against abstractions retrieved by the simple retrieval mechanism to identify omissions and inconsistencies. The software engineer is also prompted to evaluate their own domain description. Once evaluated this description passes to the analogy engine to be matched analogically against all known domain abstractions.

5.3.3 The Problem Identifier: A Summary

The problem identifier acquires key facts about problem domains using a battery of explanation tactics including domain abstraction, example-based explanation, domain visualisation and information hiding. The fact acquisition dialogue promotes early retrieval of domain abstractions to provide inexperienced software engineers with the domain knowledge necessary to structure and scope the problem space, identify key facts and evaluate their final domain description. As such it is intended to overcome many of the analytic problems identified in the first study in chapter 4. This incremental prototyping approach is similar to that implemented by Fischer and his colleagues at the University of Colorado (e.g. Fischer et al. 1991a). Once acquired, the domain description is passed to the analogy engine to retrieve one or many domain abstractions and analogical specifications.

5.4 The Analogy Engine

The analogy engine is a knowledge-based specification retrieval mechanism representing a computational implementation of the logical model of software engineering analogies defined in chapter 3. Analogical matching has been shown to be a plausible and computationally tractable way of matching a source and target analog (Holyoak & Thagard 1989, Gentner 1989, Falkenhainer et al. 1989). Implementations of Gentner's structure-mapping theory (Gentner 1983) and Holyoak's matching by constraint satisfaction (Holyoak & Thagard 1989) suggest that analogical matching is achievable using interacting structural and semantic constraints which also constrain analogical matching between software engineering domains, see chapter 3. This section presents the algorithms used to match descriptions of target domains, source domains and domain abstractions. First however, a review of existing, general computational models of analogy is presented.

5.4.1 Review of Existing Analogical Matching Mechanisms

Several computational models of analogical matching have been developed. However, three models, the Structure-Mapping Engine, Analogical Constraint-Mapping Engine and the Constrained Semantic Transference model, warrant special attention. Gentner's Structure-Mapping Theory (Gentner 1983) was implemented as the Structure-Mapping

Engine or SME (Falkenhainer et al. 1989). Following this theory, only structural criteria are used to construct and evaluate mappings. The SME algorithm identified syntactic similarities between local matches, then global matching identified their best structural match from structural constraints. The SME was applied to over 40 example analogies drawn from a variety of domains and tasks. It proved effective for simulating human responses for analogical short stories and metaphors as well as serving as a module in a machine learning program. Modified versions of the SME have been found in more recent case-based reasoning engines such as GREBE (Branting 1991).

The Analogical Constraint Matching Engine, or ACME (Holyoak & Thagard 1989) operates in a similar way to the SME, although it also recognises and exploits semantic and pragmatic constraints. Structural consistency is imposed in terms of a morphism between sets of consistent mappings of source and target objects. Semantic similarity loosens the limitations of syntactic similarity between isolated components, so allowing the concept of similarity of meaning to be recognised, although Holyoak and Thagard are unclear as to how such constraints are imposed in ACME. On the other hand, Indurkhya (1987) proposed a formalism based on first order predicate calculus for representing knowledge structures associated with a domain to develop a theory of constrained semantic transference. This formalism allows the terms and structural relationships of the source domain to be transferred coherently across to the target domain by emphasising the coherency of knowledge transfer. It was employed to explain several cognitively identified features of analogy and metaphor. Its algorithms relied heavily on the existence of relevant domain knowledge.

To conclude, SME and ACME represent two domain-independent analogical mechanisms with implications for design of the analogy engine. On the other hand, Indurkhya identifies the need for domain knowledge matched by the algorithm, similar to the inclusion of key domain abstractions in analogical matching.

5.4.2 Ira's Analogy Engine

Ira's analogical retrieval mechanism matches key facts about software engineering analogies defined in the meta-schema. It matches a concrete domain to its abstraction, then this abstraction to key facts about reusable specifications. To recap, analogical matching is successful if a target domain, source domain and domain abstraction share a coherent structure of semantically equivalent facts, and the extent of analogical match is determined by the degree of overlap between these mapped structured facts.

The analogy engine consists of four components. First, the analogical matcher identifies candidate analogical matches with one or many domain abstractions. The abstraction selector then reasons heuristically about key differences between these abstractions to select the best match. Third, the analogy determiner combines quantitative measures of similarity from the analogy matcher and selector to determine the degree of overall analogical match. The analogy engine also includes the simple retrieval mechanism described earlier. The three stages of the matching process are shown in Figure 5.7 and the architecture of the analogy engine is shown in Figure 5.8. Each phase is examined in turn.

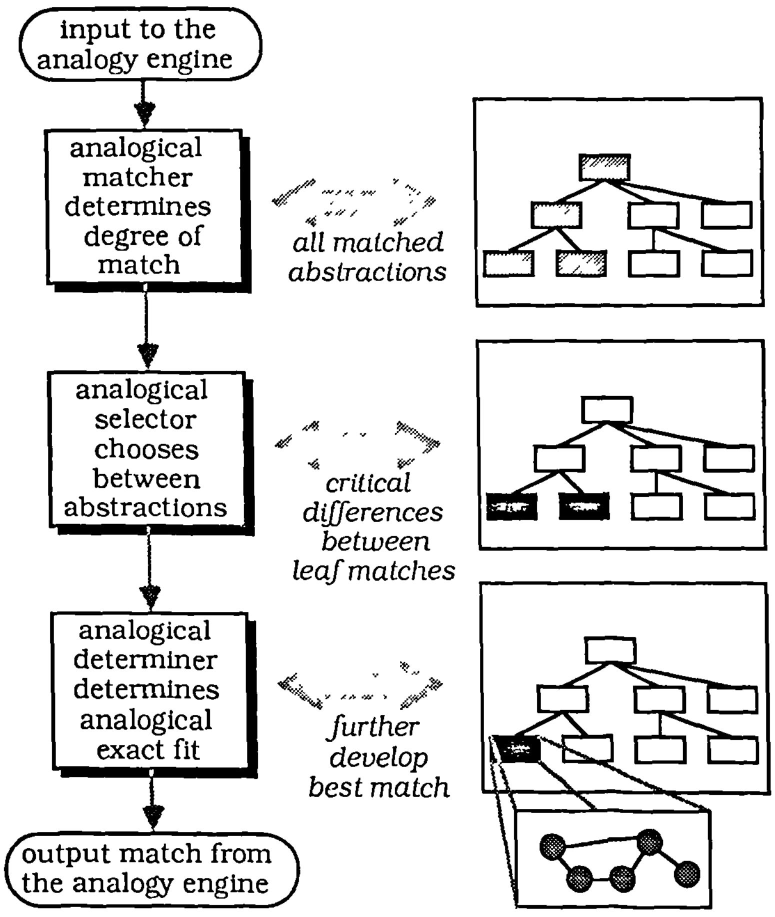

input to the analogy engine
analogical matcher determines degree of match
all matched abstractions
analogical selector chooses between abstractions
critical differences between leaf matches
analogical determiner determines analogical exact fit
further develop best match
output match from the analogy engine

Figure 5.7 - three main stages of the analogical matching process by the analogy engine

5.4.2.1 The Analogical Matcher

The analogical matcher determines the extent of a match between key domain facts and each domain abstraction. Structural coherence ensures that mappings occur between interrelated knowledge structures, similar to the SME (Falkenhainer et al. 1989). Mappings occur between the seven types of knowledge defined in the meta-schema:

object structure:	**< object, object, structural-relation >**
domain requirement:	**< object, object, structural-relation, value >**
state transition:	**< object, source, destination, transition >**
object type:	**< object, object-type >**
conditions on state transition:	**< precondition, object, source, destination, transition>**
external transition events:	**< state transition >**
function achieving transition:	**< function >**

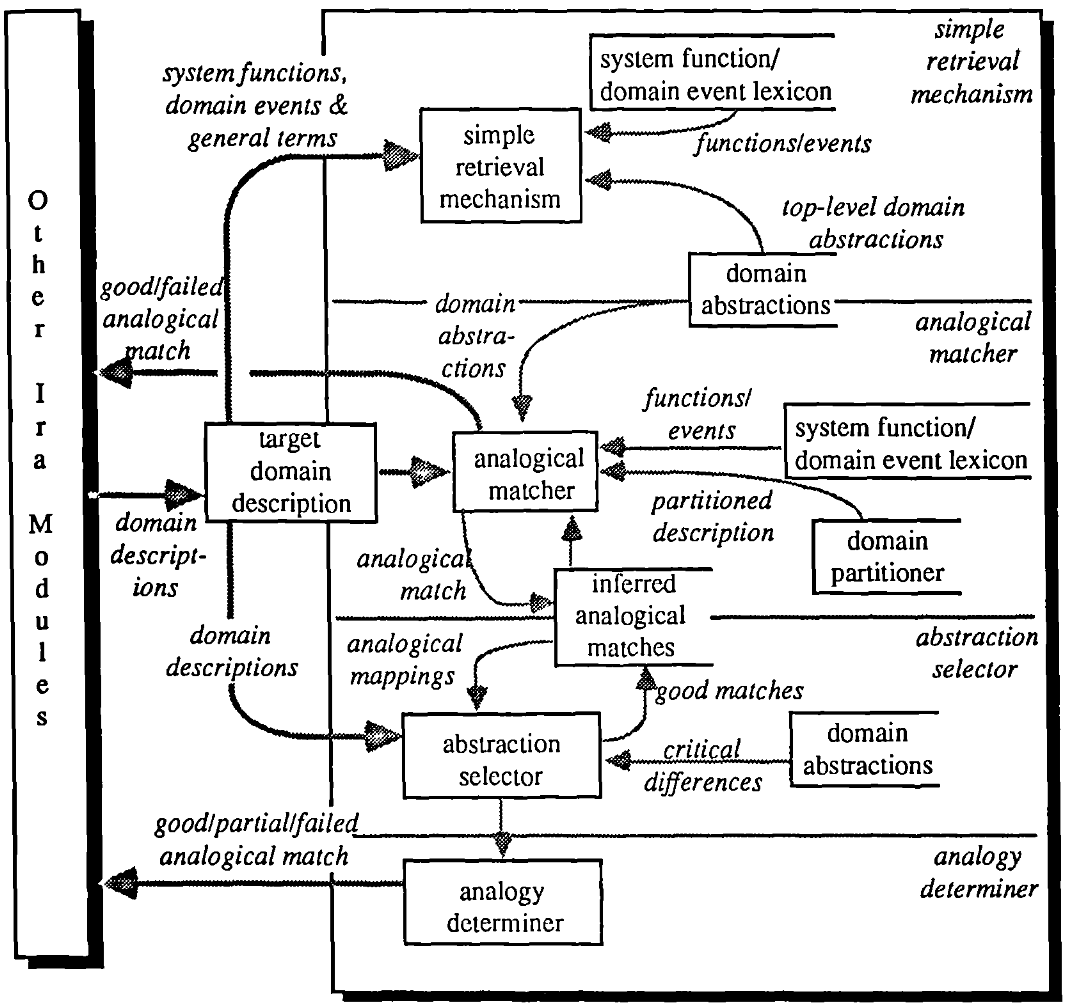


Figure 5.8 - architecture of the analogy engine. The analogical matcher identifies candidate analogical matches which are passed to the abstraction selector which identifies critical differences between candidates as a basis for their selection. Selected matches are passed through the analogy determiner before being presented to the software engineer using the specification advisor

The Structural Coherence Algorithm

The structural coherence algorithm maps pairs of facts which belong to a coherent knowledge structure in preference to those which do not. Previous analogical matching mechanisms discussed in section 5.4.1 had excessive runtime and were computationally complex (Holyoak & Thagard 1989), so the structural coherence algorithm attempts to overcome these problems in three ways. First, definitions of domain abstractions and descriptions are kept small and do not exceed 30 key facts, thus reducing runtime. Second, the analogical matcher's search space is limited by only investigating mappings which are possible given the current state of the analogical match. For example, the analogical matcher does not investigate mappings between information system functions or domain events until their linked state transitions have been matched. Finally, and most importantly, the analogical matcher does not attempt exhaustively to map all possible analogical combinations. Rather it approximates a coherent match between a domain description and abstraction by identifying a topology of related facts. Each fact about a domain abstraction is mapped to the domain fact which fits best with analogical mappings between its linked domain abstraction facts. Thus, the structural algorithm uses a complex and computationally-intensive but pragmatic search strategy. The algorithms which define structural coherence are specified in Appendix I.

The structural coherence algorithm is best demonstrated by a simple example. Figure 5.9(a) shows a fact describing the object structure of a domain abstraction and two facts describing the object structure of a new domain. According to the algorithm, only one of these two domain facts can be matched to the abstract fact, so the structural coherence algorithm must determine which domain fact fits best with the overall analogical match. First, candidate mappings in the structural match must be semantically equivalent, so if only one semantically-matched fact exists in both descriptions they are mapped. In the example, two semantically-matched domain facts exist, so the best mapping is selected by a quantitative degree of fit for each mapping to its neighbouring facts. Second, to this end, each domain fact receives a score for every neighbouring candidate mapping which it is connected to, a candidate mapping also occurring if the neighbouring facts in the domain and abstraction are semantically equivalent. For instance in Figure 5.9(a), object A1 is also defined in four other facts, hence they are neighbours. These neighbouring facts are defined by the domain terms r1, r2, r3 and r4. For the first domain fact there are two abstract neighbours which are semantically equivalent (similar r1, r3) but for the second domain fact there are four candidate mappings, suggesting that the mapping with the second fact may fit best into the analogical match. Mappings with object A2 are also counted to give a score of overall fit for each target fact. As expected, the second domain fact is the best match to the abstract fact because it is connected to 7 potential neighbouring matches while the first fact only has 5 neighbouring matches. State transitions, object types and preconditions on state transitions are mapped using the same structural coherence algorithm. This matching process is repeated for each of the above domain terms in the target description until a degree of match between it and each candidate abstraction has been calculated.

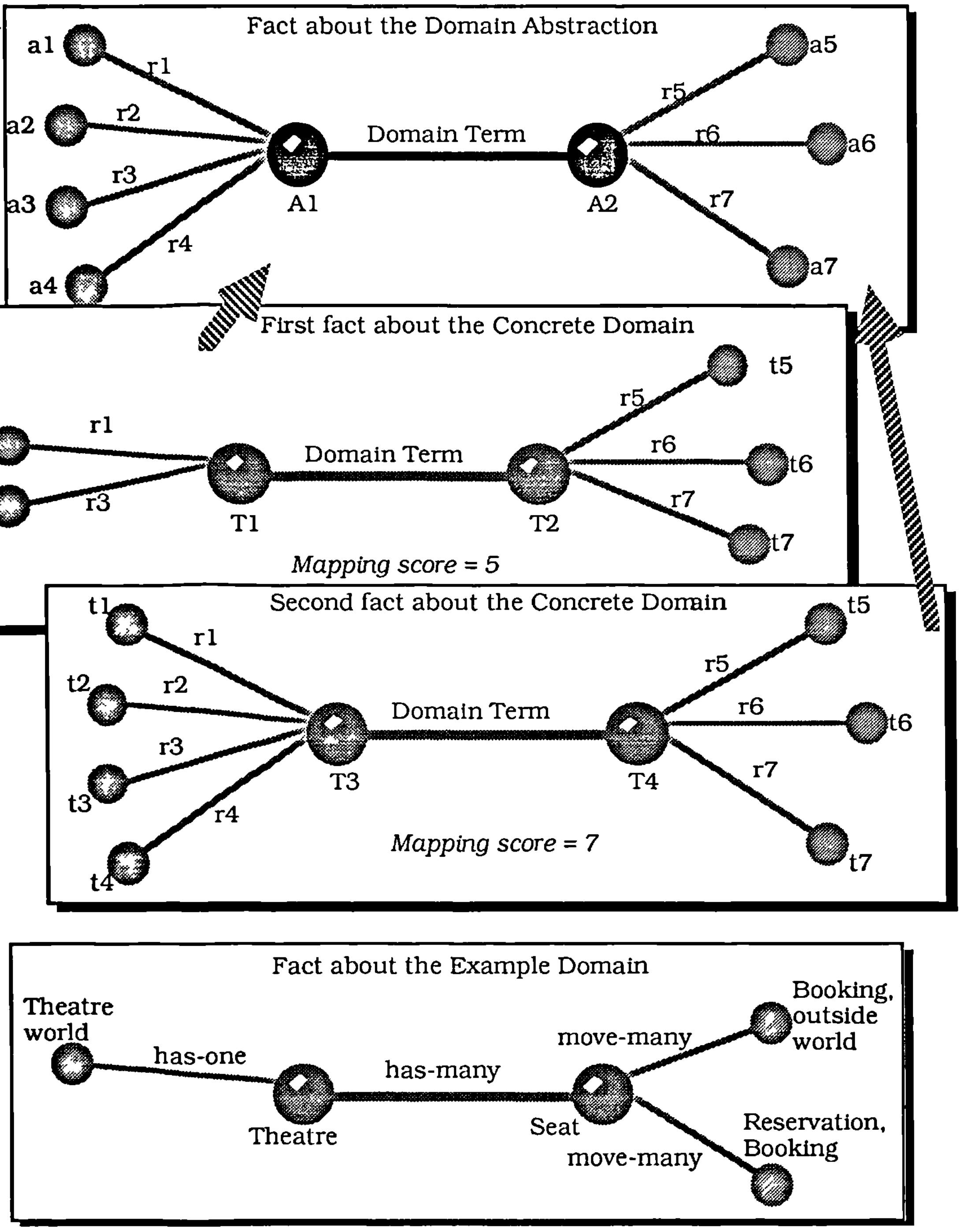


Figure 5.9(a) - example demonstrating the structural coherence algorithm to select the most appropriate domain fact for each candidate fact about the domain abstraction. Resultant object mappings are:
A1 <-> T3,
A2 <-> T4.

Once a structurally coherent match has been established, the analogical matcher attempts to map other knowledge types such as information system functions, domain events and domain requirements. This matching respects the structural coherence of existing analogical mappings, as shown by the algorithms in Appendix I. It also includes similarities between general domain terms defined in section 5.2.3. An overview of a structural match between the theatre reservation and university course administration domain is shown in Figure 5.9(b).

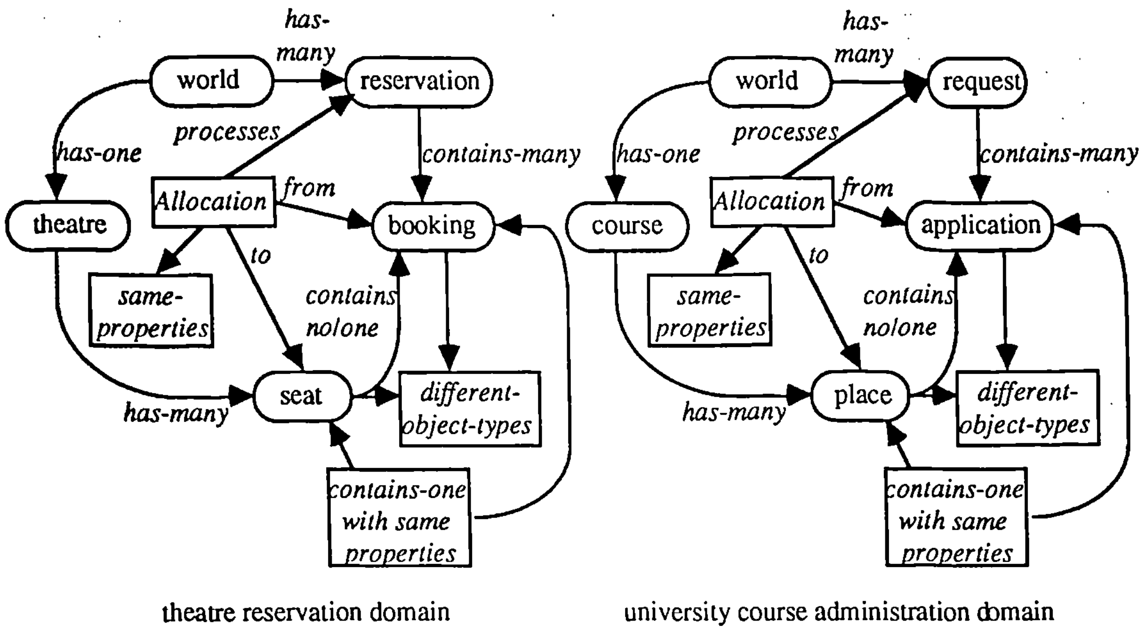
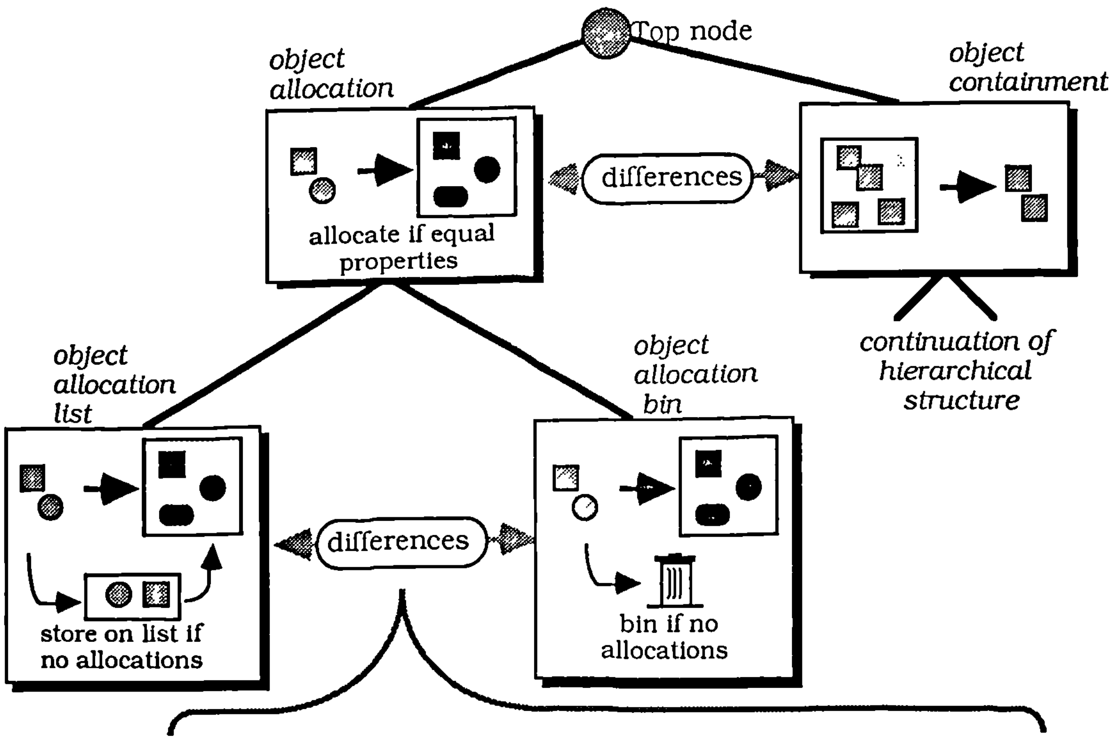


Figure 5.9(b) - network demonstrating the structural isomorphism of the analogical match between the theatre reservation and university course administration domains (lozenges represent domain objects, rectangles and lines show domain terms)

To sum, the structural coherence algorithm approximates structure matching between domain terms describing a concrete domain and its abstraction. Subsequent object mappings can inferred from their equivalent positions in this structural match, see Figure 5.9(b). Approximation of structural matching is due to the small size of the domains matched by the analogy engine, so it may not be appropriate for mapping larger analogs evaluated by the SME (Falkenhainer et al. 1989) and ACME (Thagard & Holyoak 1989). Unfortunately, paper-based evaluation of the effectiveness of the analogical matcher indicated that it alone is insufficient for analogical retrieval of specifications. The next section discusses the abstraction selector, a component which reasons about critical differences between analogically-matched domain abstractions to assist selection of the best analogical fit.

5.4.2.2 The Abstraction Selector

The abstraction selector reasons heuristically about critical differences between domain abstractions at each sub-level in the abstraction hierarchy. For example, Figure 5.10 shows the retrieval path to access an abstraction for object allocation. Analogical matching between large and flat domain descriptions (e.g. Kline's baseball analogy, cited in Falkenhainer et al. 1989) can require several hours to complete, so structuring the search space appears to be necessary to reduce search times. Figure 5.10 also reveals the close similarity between domain abstractions at lower levels in the hierarchy, indicating the need for an alternative reasoning mechanism to identify critical differences between these abstractions.



differences identifying the list domain:
1- there is a mapped object type LIST;
2- there is a mapped state transition from REQT SET to LIST;
3- there is a mapped state transition from LIST to RESOURCE SET.

differences identifying the bin domain:
1- there is a mapped object type BIN;
2- there is only one mapped state transition into RESOURCE SET.

Figure 5.10 - example of part of inheritance hierarchy describing the position of a domain abstraction for the theatre domain, and critical differences with neighbouring abstractions such as that for train reservations

The analogy engine reasons heuristically about key differences between candidate abstractions and analogical mappings between key components. In the example shown in Figure 5.10 the structural coherence algorithm has identified an analogical match with two domain abstractions, so six heuristics reason about key differences between the abstractions then calculate this difference as a percentage of the total possible differences between the abstractions. Currently a critical difference occurs if this score exceeds a predetermined percentage of difference between the abstractions. The algorithms for determining critical differences are given in Appendix I.

5.4.2.3 The Analogical Determiner

This component determines the overall analogical match from results obtained from the analogical matcher and selector, see Figure 5.7. Three levels of analogical match can occur:

• a *good-match* indicates a successful match with a domain abstraction;
• a *partial-match* signifies a possible analogical match, although more domain facts are needed to confirm or reject the match;
• a *failed-match* indicates no analogical match with any abstraction.

These degrees of analogical match are determined by quantitative measures of structural coherence, key differences between abstractions and simple measures of similarity between domain requirements, functions achieving transitions and general domain terms. A quantitative measure of structural coherence is calculated by dividing the predetermined total number of possible mappings with each domain abstraction by the number which occur. Similarly the degree of difference between abstractions is calculated as a percentage of the total possible differences. The algorithms for determining good- and partial-matches are shown in Appendix I. As such the analogical determiner acts as the top-level controller of the analogy engine, see Figure 5.7.

5.4.3 Additional Matching Techniques

Partitioning the search space of domain abstractions requires some modification of the structural coherence algorithm to overcome additional problems which arise. The first problem is that domain abstractions become more specialised with each level of abstraction, so each domain fact must only be matched to facts describing one abstraction in the hierarchy. Additional constraints are needed to avoid repetition and redundancy of analogical mappings and redundant analogical matching, see Figure 5.11. The structural coherence algorithm is modified to ensure that an analogical mapping with a domain fact may only occur with one abstraction in the hierarchy. Subsequent analogical matching with abstractions lower in the hierarchy must be consistent with mappings made to higher-level abstractions. In Figure 5.11, the analogical mapping from the target object *waiting list* to the abstract object *list* is dependent on the existence of analogical mappings inferred between the target description and a higher level domain abstraction.

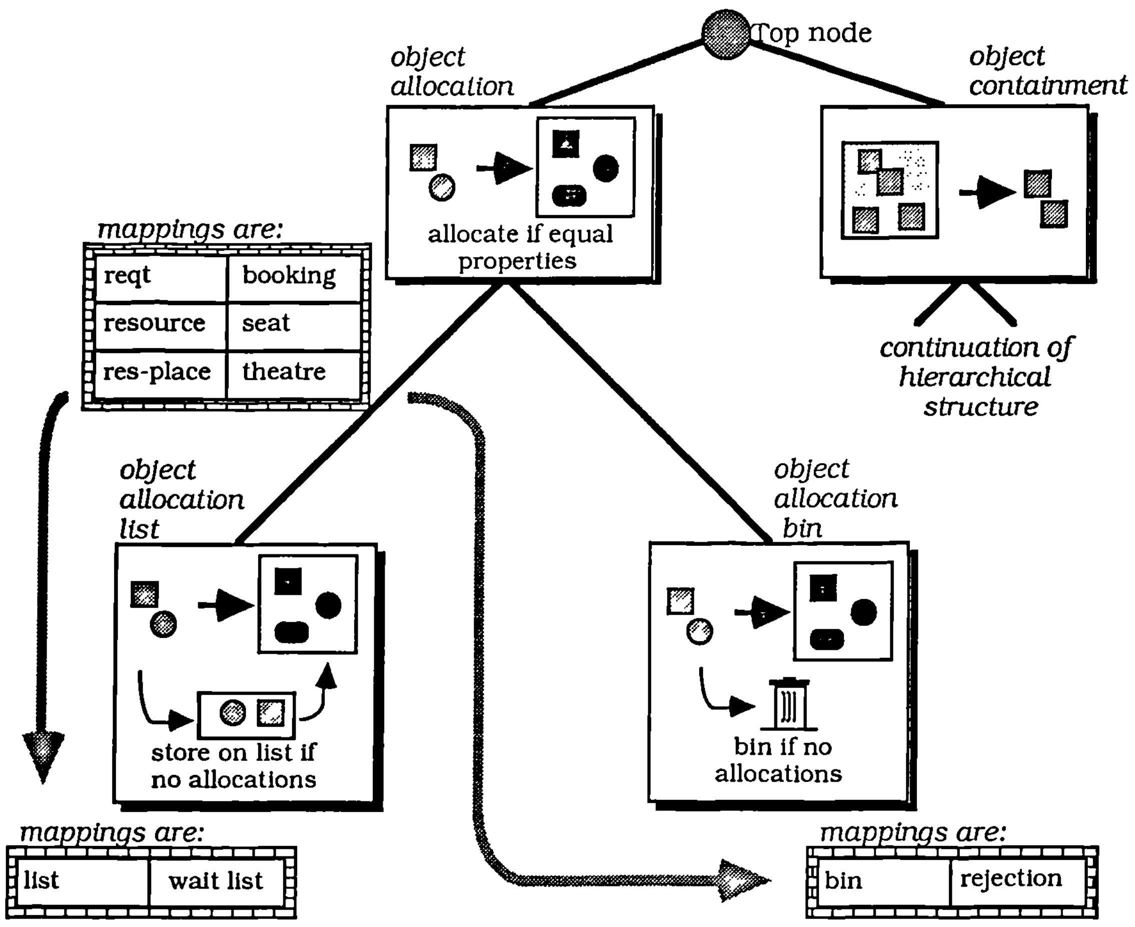
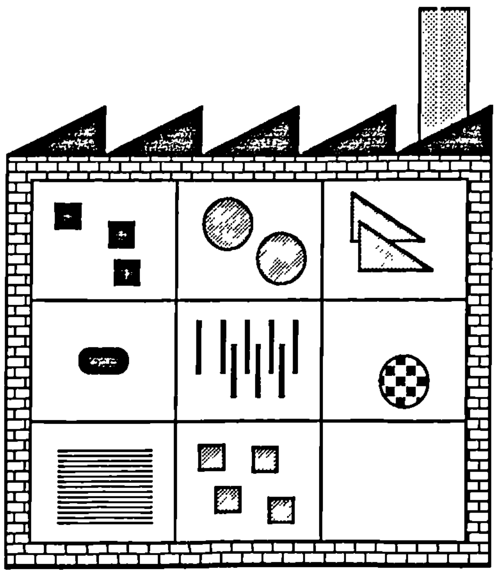


Figure 5.11 - example of part of the inheritance hierarchy demonstrating the inheritance of high-level analogical mappings on lower-level mappings, for instance mapping *list* to *waiting list* cannot be achieved without the higher-level mappings

A second problem which may arise is that matching must be constrained to ensure that high-level domain facts are mapped to high-level domain abstractions and lower-level facts are mapped to low-level domain abstractions. For instance, a complex stock control domain instantiates the low-level domain abstraction shown in Appendix J. To reach this abstraction the stock control domain must be matched to domain abstractions at three different levels in the domain hierarchy, as described in Figure 5.12. This is achieved by partitioning the target domain description into high and low-level facts about the domain, decided by an algorithm given in Appendix L which layers objects by their links with the highest-level *world* entity. For instance, high-level objects are linked by object structures to the *world* entity (e.g. *world has-one warehouse*) whereas low-level objects are not (e.g. *bin contains-many stock-items*). Such partitioning is achieved by the domain partitioner shown in Figure 5.8.



Consider a warehouse with many stock bins, each with many stock items held within. This knowledge structure can be represented in two ways, only one of which may be entered during fact acquisition:
i) WAREHOUSE contains-many ITEMS
ii) WAREHOUSE contains-many BINS, &
BINS contains-many ITEMS.

Figure 5.12(a) - effective matching by the analogy engine requires the ability to recognise equivalence between domain facts. Analogical matching occurs at different levels of domain abstraction, so high-level facts must be mapped to high-level abstractions early in the matching process, and low-level abstractions must be mapped to low-level abstractions later in this process

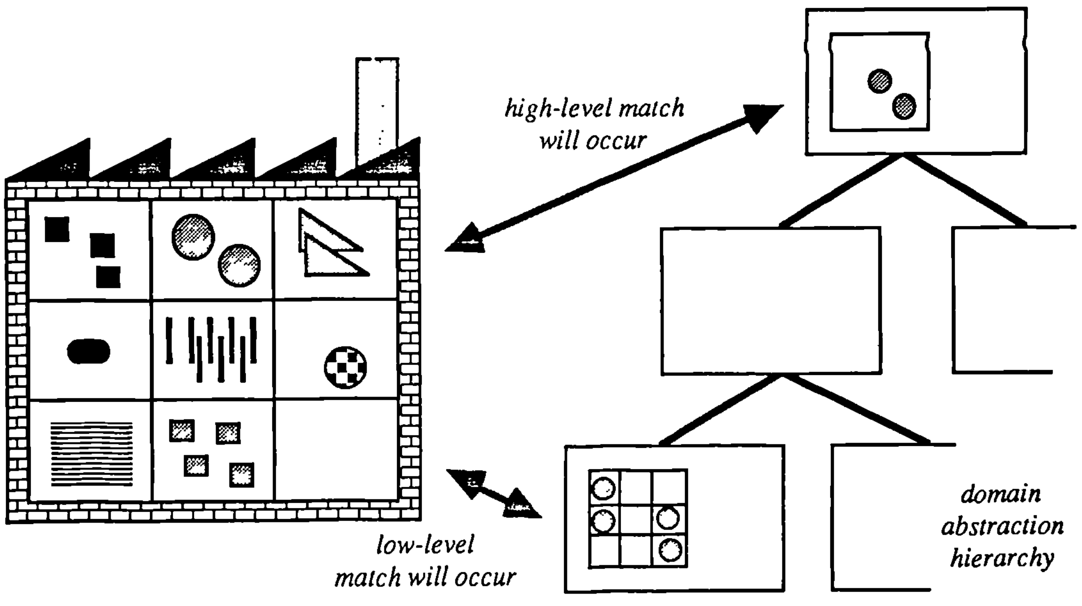

high-level match will occur
low-level match will occur
domain abstraction hierarchy

Figure 5.12(b) - demonstration of massaging the domain description to enhance matching to high-level and low-level domain abstractions

The two specified additions to the structural coherence algorithm are defined in Appendix I. Their inclusion in the analogy engine represents the development of a pragmatic retrieval mechanism for specification reuse rather than a theoretically-based, computational model of analogical reasoning. These modified algorithms represent the analogical mapping mechanism implemented in Ira's prototype described in chapter 6.

5.4.4 Matching Generic Domain Worlds

The logical model of software engineering analogies in chapter 3 incorporates generic domain worlds representing common domains. To recap, one example of such a world instantiates the object monitoring abstraction to a safety-critical transport world in which manned vehicles move and risk collision in two- or three-dimensional spaces such as air traffic control, train management and shipping movements along busy sea lanes. Matching the physical attributes of domain objects can assist matching to the best-fitting domain abstraction. The ATC domain, unlike the FMS domain, belongs to the generic domain world safety-critical-transport, hence matching the ATC description will favour analogical specifications instantiating both the correct domain abstraction and the generic domain world (e.g. a specification of a control system for ship movements) in preference to those which do not.

The analogy engine infers the presence of generic domain worlds from physical attributes belonging to domain objects, for instance physical attributes of objects in the ATC and video hiring domains are matched to similar physical attributes of domain abstractions:

< aircraft, manned-vehicle > maps to **< object, manned-vehicle >**
< video-copy, borrowed-item > maps to **< object, borrowed-item >**
etc..

To sum, physical attributes attached to abstract objects are shown in Appendix J. Physical matching only occurs once a good match has been identified between the domain description and its abstraction. Physical attributes are matched by the analogy engine using a simple measure of percentage lexical match between physical attributes belonging to each domain and abstraction. This simple algorithm is also defined in Appendix I.

5.4.5 Summary of the Analogy Engine

To summarise, the analogy engine consists of three major components. The analogical matcher determines a structurally coherent match between a new domain and its abstraction. The abstraction selector then reasons heuristically about critical differences between matched domain abstractions to choose the best fit. Finally the analogy determiner combines the measures of analogical similarity inferred by the analogical matcher and selector to produce a single measure of similarity between a domain description and its abstraction. This analogy engine represents a pragmatic specification retrieval and explanation mechanism founded on a computational implementation of the logical model of software engineering analogies. The analogy engine differs from earlier computational models of analogy in its implementation of both case-based and rule-based reasoning paradigms. It more closely resembles recent hybrid analogy engines such as CABARET (Rissland & Skalak 1991). It provides the advisor's expertise for both specification retrieval and explanation. The importance of explanation during specification understanding and transfer is examined more closely in the next section.

5.5 The Specification Advisor

The specification advisor explains and assists customisation of retrieved specifications. Its domain expertise is derived from mappings inferred by the analogy engine while its knowledge of software engineers' analogical understanding is founded on the empirically derived malrules and mental models of analogical understanding and the cognitive task and reasoning models which identify strategies for effective specification understanding and transfer. Definition of the specification advisor's architecture is followed by explanation and reuse strategies and tactics from the empirically-derived findings in chapter 4.

5.5.1 The Architecture of the Specification Advisor

The specification advisor's architecture is shown in Figure 5.13. Comprehension of the specification is assisted by explaining the analogical match using its underlying domain abstraction. The specification modeller and explainer then operate in an iterative present-diagnose cycle based on the cognitive model of analogical reasoning to support analogical transfer. During this cycle the software engineer enters modified components and analogical mappings until he/she requests additional explanations or enters an incorrect mapping. Immediate error correction is in keeping with Anderson's et al.'s (1987) guidelines for tutor design. Errors are identified by the analogy engine's inferred mappings and the reuse error library shown in Figure 5.13. The software engineer is informed of the goals and structure of the dialogue throughout, in keeping with another of Anderson et al.'s (1987) tutor guidelines. The remainder of this chapter defines the advisor's expertise, empirically-derived strategies for effective analogical comprehension and transfer, explanations to assist analogical reasoning and rules for error detection and fixing.

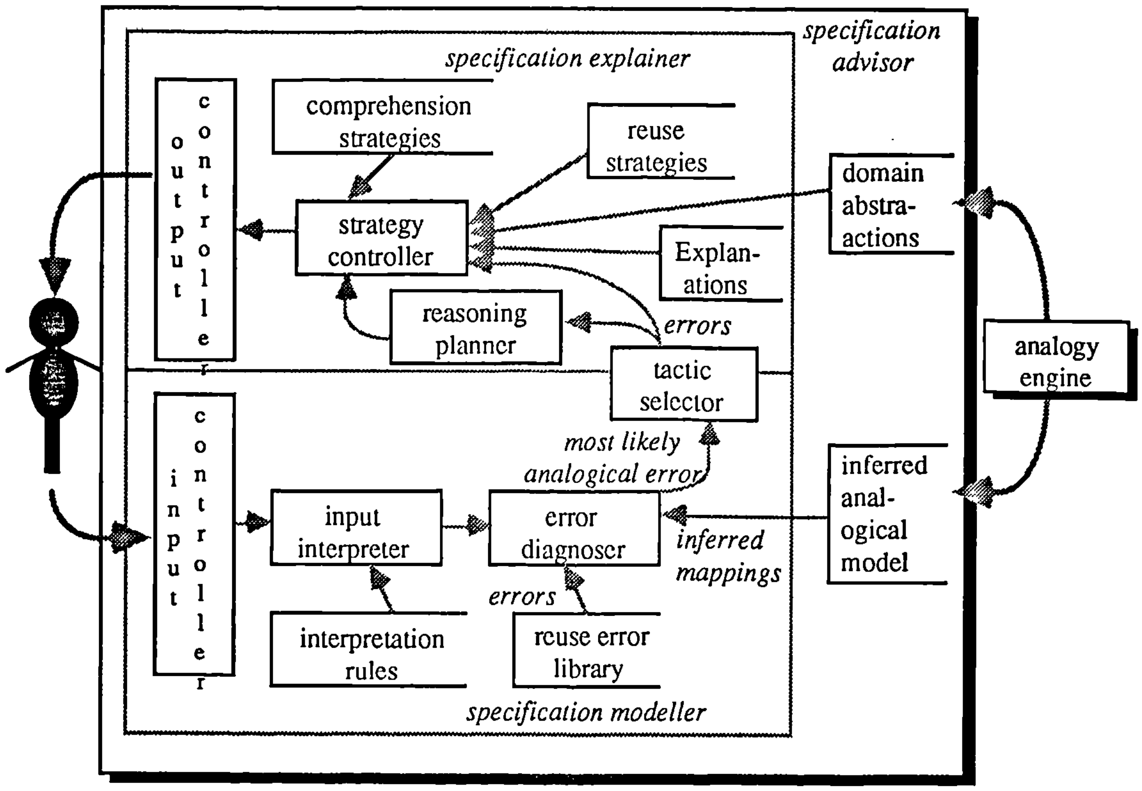


Figure 5.13 - architecture of the specification advisor

5.5.2 The Specification Advisor's Expertise

Explanation and error diagnosis by the advisor are limited by its domain abstractions and inferred analogical mappings. Explanation was enhanced by causal links between knowledge types in the meta-schema (e.g. White & Frederiksen 1990), see Figure 5.14. For example, the causal structure linking the knowledge types is necessary to construct the following explanation for the ALLOCATION system function:

the ALLOCATION function is caused by the state transition moving booking from an unoccupied to an occupying-seat state

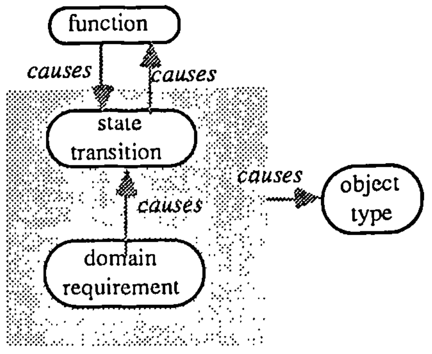


Figure 5.14 - causal links between knowledge types defined in the meta-schema, to permit causal, teleological and function-based justification during explanation

5.5.3 The Specification Explainer

The specification explainer has three parts. First, it assists specification understanding using empirically-derived strategies. Second, these strategies are aided by explanation of the analogy using mappings inferred by the analogy engine. Finally the advisor assists the software engineer to transfer the analogical specification.

5.5.3.1 Support for Specification Understanding

The cognitive task and mental models underpinning the strategies to aid specification understanding are reported in section 4.4.4. The aim of these strategies is to improve the software engineer's understanding of the analogy and its underlying abstraction from their initial interaction with the problem identifier. Two mocked-up examples of the use of these strategies are shown in Figures 5.15 and 5.16. The strategies are:

- promote an abstract understanding of problem solving knowledge, following Anderson et al.'s (1987) guidelines for tutor design. In particular, software engineers must understand key domain abstractions underlying the analogical match;
- minimise working memory load throughout the comprehension phase (Anderson et al. 1987). Present abstract and source domain knowledge in small, manageable chunks. Coordinated explanation dialogues must be based on the rationale underlying knowledge types in the meta-schema;
- encourage a good analogical understanding before transferring the specification. Use spatial diagrams to explain the key domain abstraction and familiar analogies to promote its induction. Spatial diagrams also serve to highlight key analogical mappings. Furthermore, the advisor explains key facts about the reusable source

domain rather than the transferable specification;

- during specification transfer, explain key domain facts alongside specification components to encourage better analogical mapping with these components;
- promote greater analogical reasoning in several ways. Encourage explicit definition of analogical mappings using electronic sketchpads and graphical representations of the analogical match. In particular, promote analogical reasoning with key facts in the source domain before reasoning with non-critical domain facts;
- present different types of knowledge defined in the meta-schema to encourage multi-layered analogical comprehension. Knowledge presentation is controlled by the reasoning planner defined in chapter 4. Analogical understanding is iterative, thus key state transitions and object structures must be understood first, then other knowledge types defined in the meta-schema, then non-critical domain attributes.

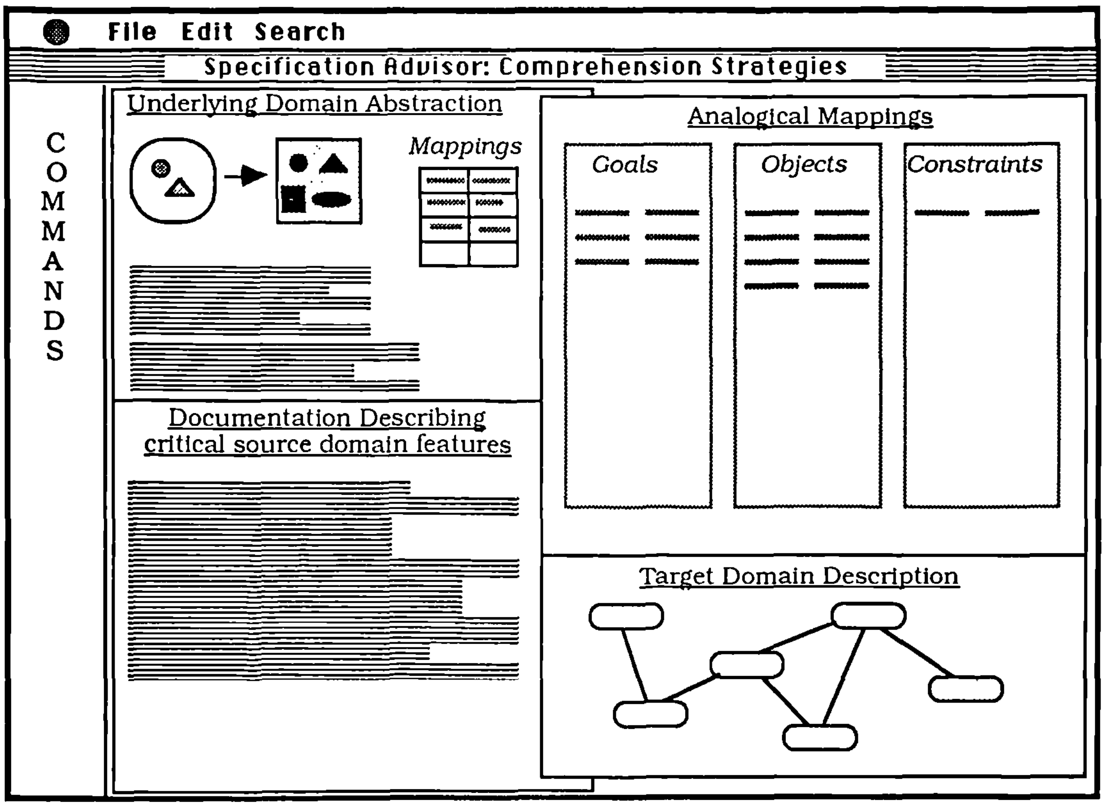

File Edit Search
Specification Advisor: Comprehension Strategies
COMMANDS
Underlying Domain Abstraction
Mappings
Documentation Describing critical source domain features
Analogical Mappings
Goals
Objects
Constraints
Target Domain Description

Figure 5.15 - mockup screen layout demonstrating strategies to aid analogical understanding

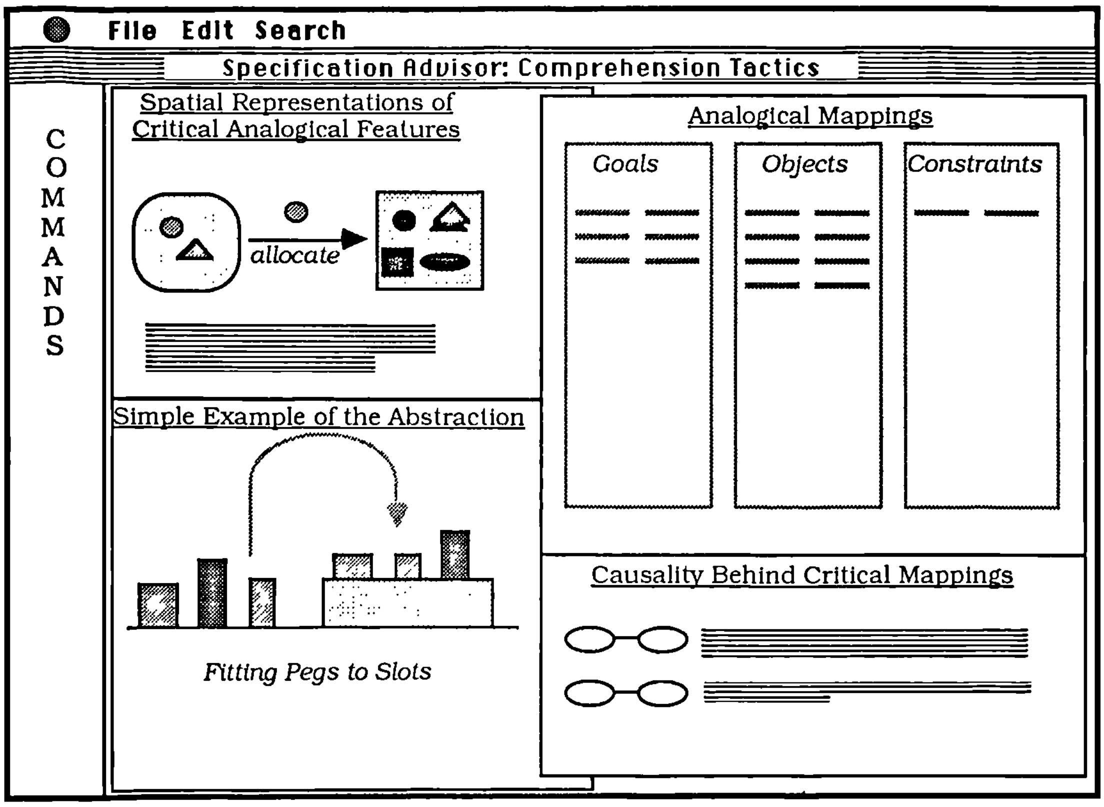

File Edit Search
Specification Advisor: Comprehension Tactics
COMMANDS
Spatial Representations of Critical Analogical Features
allocate
Simple Example of the Abstraction
Fitting Pegs to Slots
Analogical Mappings
Goals
Objects
Constraints
Causality Behind Critical Mappings

Figure 5.16 - mockup screen layout demonstrating strategies to aid analogical understanding

5.5.3.2 Strategies for Transferring Specifications

Strategies leading to effective analogical transfer are demonstrated in the mocked up screen in Figure 5.17. The key components of these strategies at the dialogue level are:

- controlled transfer of specification components to maximise reuse. Dialogue windows present each major component and its neighbours in turn. Initiative remains with the system while it lists all candidate components for reuse, supported by scripted explanation dialogues outlined in section 5.5.2. The software engineer is unable to access other dialogues during this controlled sequence;
- the system hides information about the specification to inhibit copying and focus reasoning attention on components being transferred. Browsable dialogues provide limited views on the specification and support guided discovery and learning about the analogical match (Elsom-Cook et al. 1988);
- the system also provides the specification structure as a template guiding its transfer. Explanatory dialogue is also guided by the specification structure. The reuse 'template' must allow editing of component labels;
- permit customisation of the reusable specification, changing component names and altering the specification structure where permissible as defined by the transfer strategy.

Work spaces are provided to permit exploration and experimentation with analogical matches;

- encourage evaluation of mapped components which are syntactically similar to reusable components. Such analogical errors and mismappings may represent mental laziness, hence the advisor requests additional analogical reasoning to justify mappings;
- encourage evaluation of the customised specification against the target system's functional requirements to avoid omissions, ambiguities and overspecification. Dialogue windows permit browsing of functional requirements. Unchecked requirements can be highlighted by the tool to ensure specification completeness. As such the analogical specification can be checked both manually and automatically.

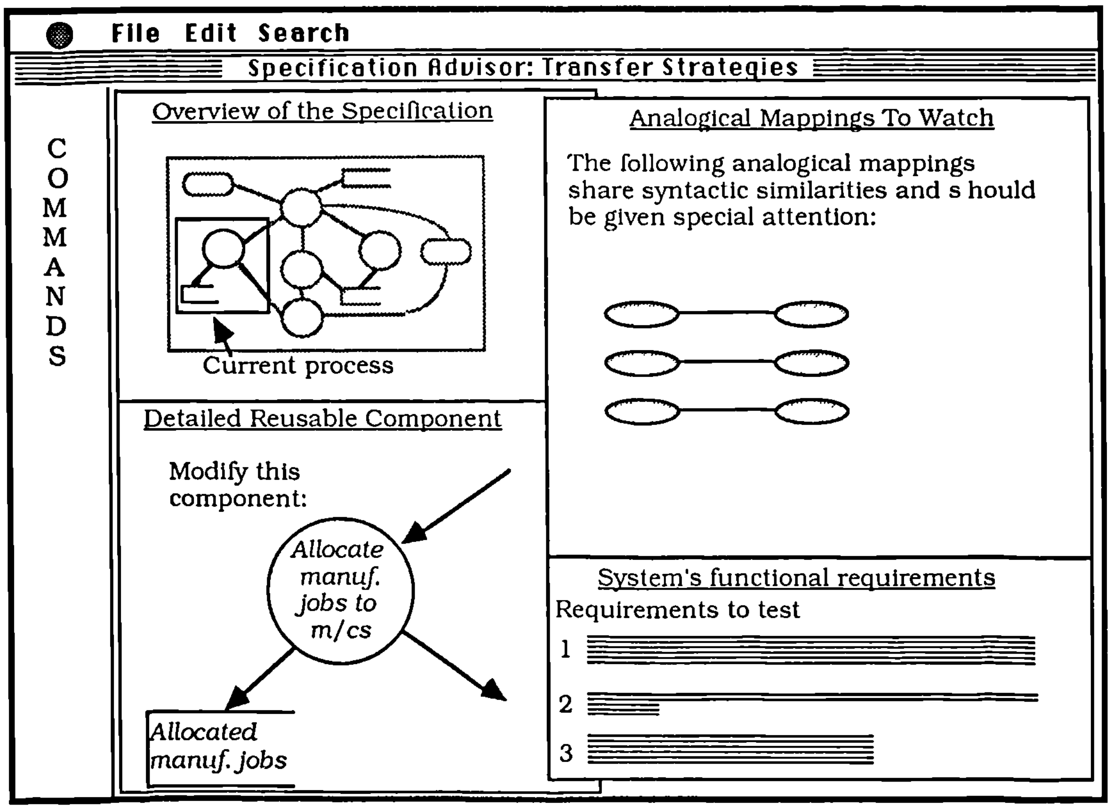

File Edit Search
Specification Advisor: Transfer Strategies
COMMANDS
Overview of the Specification
Current process
Detailed Reusable Component
Modify this component:
Allocate manuf. jobs to m/cs
Allocated manuf. jobs
Analogical Mappings To Watch
The following analogical mappings share syntactic similarities and s hould be given special attention:
System's functional requirements
Requirements to test
1
2
3

Figure 5.17 - mockup screen layout demonstrating strategies to aid specification transfer

5.5.3.3 Strategies for Analogical Reasoning

The reasoning planner defined in section 4.4.4 supports iterative analogical reasoning, during which an analogical hypothesis is generated then developed and tested in the target domain. As such, the planner maintains control of interaction with the software engineer during specification comprehension and transfer. This interaction can be integrated with explanation of analogical mappings and reusable components. Strategies to assist analogical reasoning, understanding and transfer provide a prescriptive basis for guiding an inexperienced software engineer during analogical reuse. However, they provide few clues for detailed explanation of the analogical match, so current theories and models of explanation are reviewed, although they are beyond the direct focus of this research.

5.5.3.4 Explanation of Analogical Specifications

Explanation can be viewed from: (i) what to say, and (ii) how to say it. The advisor's expertise defined in section 5.5.2.1 determines what to say while this section specifies classes of explanation which determine how to say it. Explanation has been researched considerably in the cognitive psychology, intelligent tutoring and artificial intelligence disciplines. Wenger (1987) identified the behavioural, epistemic and individual target levels of didactic operations, the first two of which are operationalised by the advisor. Interventions at the behavioural level can be classified as behavioural guidance (e.g. specific hints or general advice) or exposure to behaviour (e.g. simple demonstrations) while the epistemic target level operates to modify the student's knowledge state, either via direct communication or practice, by organising specific experiences to expose the student to. Explanations are central to dealing with articulation of knowledge at both levels of didactic operation.

Many taxonomies of explanation exist (e.g. Wenger 1987). For instance, Breuker (1988) identified explanation as one of six pedagogical tactics acting on objects in an intelligent tutoring environment. Wenger (1987) proposed an overlapping classification of explanation types, divided into three classes: justifications, integrations and organisations. Justifications include teleological, causal and functional accounts. Integration encompasses explanations described by articulation of genetic, analogical and contrasting relations, and articulation of abstract objects and procedures. Finally organisational explanations include articulation of general principles. Many of these explanations can be incorporated into the advisor, for example:

- justification by teleological, causal and functional accounts exploits causal links between knowledge types in the meta-schema defined in section 5.5.2.1;
- familiar analogies are another tactic for integrating domain abstractions into the software engineers' current understanding. Domain abstractions can be also compared and contrasted to emphasise key differences when understanding and selecting abstractions. Articulation of organisational and general principles also emphasise the importance of key abstractions and the underlying rationale of the analogical match.

These classes of explanation tactic are incorporated into the advisor. The type of explanation is determined by current comprehension and transfer strategies and analogical misconceptions exhibited by software engineers.

5.5.3 The Specification Modeller

The specification modeller captures domain facts and analogical mappings for diagnosing the software engineer's analogical understanding. Analogical mappings can either be inferred from the customised specification (see Figure 5.18) or elicited directly by questioning. Indeed, Self (1988) suggests that direct questioning has been underutilised as a diagnostic strategy in ITS development.

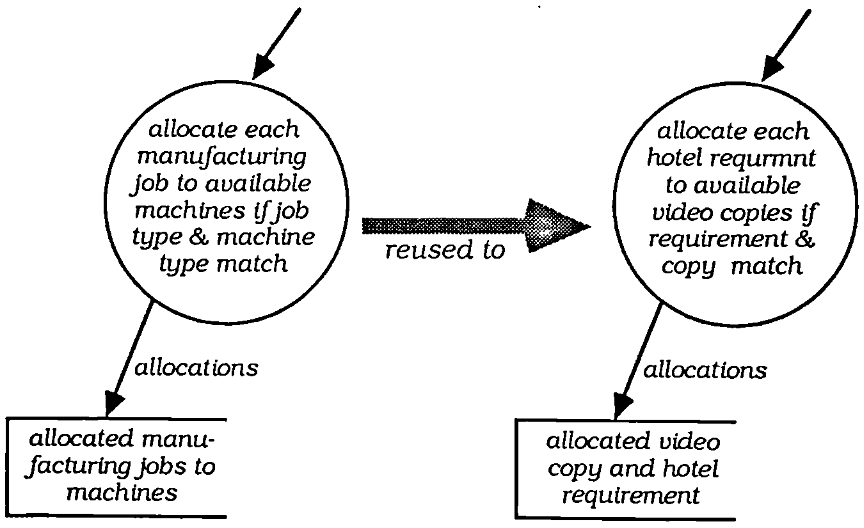

allocate each manufacturing job to available machines if job type & machine type match
reused to
allocate each hotel requrmnt to available video copies if requirement & copy match
allocations
allocations
allocated manu-facturing jobs to machines
allocated video copy and hotel requirement

Figure 5.18 - demonstration of possible lexical mappings during specification transfer, indicating that:
video copy <--> *machine*
hotel requirement <--> *production job*

Subsequent error diagnosis is achieved by comparing the software engineer's analogical understanding to that inferred by the analogy engine. The advisor combines simple overlay and enumerative bug models (e.g. Wenger 1987, VanLehn 1988) to determine candidate errors. These models were considered appropriate given the advisor's limited expertise. Initially models are overlaid to identify incorrect and omitted analogical mappings. Candidate reasons for these errors are then inferred from the enumerative error library derived from errors exhibited in the third reported study. The current error library is defined in Appendix H, although it can be extended to incorporate common errors involving analogical mappings with specific domain abstractions. Dialogue with the software engineer can confirm erroneous mappings and diagnoses or ask the software engineer for other analogical concepts which are poorly understood. Diagnosis informs the advisor's tactic selector so that misunderstood analogical concepts can be explained, thus providing immediate feedback on errors (Anderson et al. 1987).

5.5.4 The Specification Advisor: A Summary

The specification advisor assists the software engineer to understand and transfer analogical specifications. Explanation is founded on the logical model of software engineering analogies defined in chapter 3, empirically-derived strategies for successful analogical comprehension reported in chapter 4 and classes of explanation from the literature on intelligent tutoring systems. Strategies which aid transfer of analogical specifications are empirically-derived from ineffective and successful reuse behaviour reported in the second, third and fourth studies in chapter 4. In addition, explanation of analogical specifications is aided by diagnosis of analogical errors and misconceptions exhibited by software engineers, supported by the error library derived from the third empirical study of analogical reuse. As such, the design of the specification advisor has a strong theoretical and empirical basis not found in many existing software reuse environments (e.g. Woods & Sommerville 1988, Prieto-Diaz 1991, Fugini et al. 1991).

5.6 A Summary of Ira

This chapter specified the advisor's three major components based on the logical model of software engineering analogies defined in chapter 3, empirically-derived cognitive task, reasoning and mental models reported in chapter 4, and existing theoretical and empirical evidence of analogical problem solving and tutor-based explanation. The problem identifier acquires key domain facts prior to analogical matching. An iterative acquisition and retrieval paradigm presents retrieved domain abstractions and concrete examples early in the fact acquisition dialogue to assist domain scoping, structuring and description. The analogy engine matches the acquired domain description to analogical domains belonging to the same domain abstraction. The engine is a computational implementation of the logical model of software engineering analogies defined in chapter 3. The analogy engine provides the tool's expertise to assist understanding and customisation of specifications. The specification advisor uses empirically- and theoretically-derived strategies to aid reuse. These strategies are supported by explanations and diagnosis of analogical errors. The advisor cooperates with the software engineer during specification reuse, a paradigm also implemented in the CODE FINDER, CODE EXPLAINER and other toolkits developed at the University of Colorado (e.g. Fischer, et al. 1991a, Fischer & Nakakoji 1991). The next chapter describes an evaluation of two of these components.

Chapter 6

6: A Prototype of Ira

This chapter describes the prototype implementation of Ira. The prototype supports fact acquisition by the problem identifier, retrieval of domain abstractions by the analogy engine and explanation of these abstractions to software engineer. Development of tool support for specification retrieval was favoured because the effort required to build the specification advisor was likely to be too great. In addition, fact acquisition and matching provided a simpler scenario to evaluate the prototype's effectiveness because it allowed software engineers to approach the domain without prior knowledge, whereas realistic evaluation of the specification advisor would require simulation of interaction and learning with the problem identifier. Partial implementation of a knowledge-based specification advisor was considered, however the time and effort needed to model the strategies and develop an integrated interface was considered to be too great. The prototype advisor was developed on an Apple Macintosh IIcx with 5Mb main memory and 40Mb hard disk using LPA MacProlog version 2.5, to exploit the package's integrated graphics facilities which enabled fast development of powerful, interactive explanation dialogues.

This chapter describes the implemented analogy engine and problem identifier in three parts. First, an evaluation of the analogy engine is reported, then the prototype problem identifier is investigated in two stages. Findings from user studies with a paper-based, free form fact acquisition dialogue revealed the need for the semi-automated fact acquisition strategies specified in chapter 5, then an evaluation of the complete prototype investigates the effectiveness of the combined fact acquisition and analogical matching components.

6.1 The Prototype Analogy Engine

The analogy engine reasons analogically to retrieve and explain analogical specifications. It implements the algorithms described in chapter 5. The engine was populated with the 10 hierarchically structured domain abstractions defined in chapter 3 and Appendix J.

6.1.1 Evaluation of the Analogy Engine

Complete evaluation of the analogy engine would have proved both difficult and time-consuming. Instead, evaluation was example-based, in keeping with evaluation of other computational models of analogical reasoning. For instance, the Structure-Mapping Engine (Falkenhainer et al. 1989) was applied to over 40 analogies to model human responses and act as a machine learning engine. ACME (Holyoak & Thagard 1989) also provided a testbed for many applications of well-known scientific analogies. Similarly the analogy engine was evaluated by matching domain descriptions representing instantiations of known domain abstractions. Evaluation occurred in two phases. First, complete and correct domain descriptions were input to ensure that they matched their domain abstractions, then completeness and consistency of these target domain descriptions were varied to investigate the engine's robustness.

6.1.1.1 Results of the Evaluation

During the first phase of evaluation, domain descriptions were successfully matched to their abstractions. When domain descriptions were matched without key state transitions or object structures, the analogy engine retrieved the correct domain abstraction in over 50% of cases and a correct higher-level abstraction in many other cases, see Table 6.1 and Appendix G. Problems arose when the correct domain abstractions were described by a greater proportion of object structures, thus emphasising the importance of object structures on structure-based analogical matching.

Domain Matched	Domain Abstra-ction	Test Without Object Structure	Test Without State Transition
Stock Control System	OCP-BA	Perfect	Perfect
Personnel System	OCP-BB	Fail	Partial *
Library System	OCP-AB	Perfect	OCP Only
Air Traffic Control System	OMP	Fail	Perfect
Coastguard Patrol System	OPP	Fail	Fail
Simple Cinema System	OAP	Perfect	Perfect
Complex Theatre System	OAP-AA	Perfect	Perfect

** Partial match with OCP, OMP & OAP classes*

Table 6.1 - results of evaluation of the analogy engine when matching incomplete concrete domain descriptions to their correct domain abstractions

6.1.2 Enhancements to the Analogy Engine

The analogy engine's matching algorithms were finely tuned in light of results from this initial evaluation. The systematicity of a match with domain abstractions was measured as a percentage overlap between the abstract and concrete descriptions, so a match between a single domain abstraction and description was quantified as:

$$S = \frac{\Sigma_1^n M}{\Sigma_1^n T} * 100 \text{ where}$$

- M represents the number of matched target and abstract domain facts, and
- T represents the total number of abstract domain facts.

Evaluations led S to be set as follows:

• excellent analogical match:	$80 \leq S \leq 100\%$
• good analogical match:	$50 \leq S < 80\%$
• partial analogical match:	$33 \leq S < 50\%$
• failed analogical match:	$0 \leq S < 33\%$.

The abstraction selector was also tuned during this example based evaluation, and a critical difference was deemed to occur if the percentage of the total number of possible differences was greater than 33%. The full listing of the implemented analogy engine (shown in Appendix L) demonstrates the complexity of the algorithms identifying analogical matches.

6.2 The Prototype Problem Identifier

The prototype problem identifier attempts to overcome incompleteness and inconsistency in domain descriptions by implementing the iterative, example-based fact acquisition and retrieval paradigm defined in chapter 5. To recap, this paradigm incorporates the following strategies:

- example-based explanation of retrieved domain abstractions;
- visualisation of retrieved analogical examples;
- information hiding to avoid example-based copying.

Initially the need for these fact acquisition strategies was evaluated through paper-based mockups of a simplified dialogue design to evaluate the effectiveness of free form fact entry versus semi-automated fact acquisition. The paper-based, free form entry dialogue omitted key strategies such as explanation and visualisation of analogical examples. It was expected that such a free form entry dialogue would lead to incomplete and inconsistent domain descriptions which would not retrieve the correct domain abstractions. Instructions for identifying and representing key domain facts were given to software engineers who then defined the domain on an answer sheet. Results from these mockups demonstrated the need for complex fact acquisition strategies which were evaluated during a second evaluation of the prototype advisor. These two evaluations are described in the next two sections.

6.2.1 The Need for Fact Acquisition Strategies

The first, paper-based evaluation investigated the effectiveness of a free form entry dialogue for describing domains, defined as non-interactive instructions for acquiring key domain facts and a single, non-analogical example to explain the use of permitted domain terms.

6.2.1.1 Method

Four inexperienced software engineers (second-year business computing undergraduates with knowledge of structured analytic techniques) were requested to identify key facts about the production planning domain described in Appendix E. All subjects were given a one-page narrative describing the domain and five pages of instruction also shown in Appendix E which explained how to identify key facts. Subjects had to select between terms for describing the domain and enter these domain terms on the answer sheet, see Appendix E. They were given 40 minutes to assimilate and describe the domain and were expected to complete their description by the end of it. Both written and verbal retrospective questioning elicited clues about the effectiveness of the fact acquisition dialogues. Finally, subjects' descriptions were entered into the analogy engine by the experimenter to determine whether they matched the production planning domain's underlying abstraction.

6.2.1.2 Results

All four subjects described the production planning domain, although these descriptions were incomplete and inconsistent. Analogical matching using the prototype analogy engine revealed that domain descriptions developed by S1 and S2 only partially matched the correct domain abstraction. On the other hand, domain descriptions developed by S3 and S4 failed to match any abstraction (see Appendix J), indicating that the mocked up fact acquisition strategies were unsuccessful. Retrospective questioning revealed that instructions were easy to use, so problems during fact acquisition and modelling were examined more closely.

Subjects' Domain Descriptions

Retrospective questioning revealed that subjects encountered most difficulties identifying key state transitions and object structures. More successful subjects (S1 & S2) defined state transitions correctly, although they confused operators with the concept of unfulfilled production needs. On the other hand, S3 and S4 failed to describe key object structures and state transitions. All subjects were more successful at selecting system functions and object categories. They all selected *allocate* as the key system function correctly, two subjects also selected *assign* because it was functionally equivalent to *allocate* while only S1 additionally selected the *goods-in* and *goods-out* functions, apparently due to poor scoping of the production planning domain. Similarly, all subjects categorised machines as system resources, although only one subject categorised jobs as different-object-types correctly.

Difficulties Encountered by Subjects

Experimenters' sketches of each subject's written domain description revealed the incompleteness and anomalies of these descriptions, see Figure 6.1. Both S3 and S4 modelled *computer* and *organisation* as separate entities rather than as the boundaries of the production planning domain. All four diagrams also contained isolated components which were inconsistent with the rest of the model. They also revealed discrepancies which contradicted subjects' verbalised intentions. In the case of S3, who sketched the domain, her model also revealed discrepancies between the model and the sketch, indicating difficulties when defining object structures, although this may have been due to the instructions provided. These findings revealed the need for more graphic domain representations to overcome inconsistent domain descriptions.

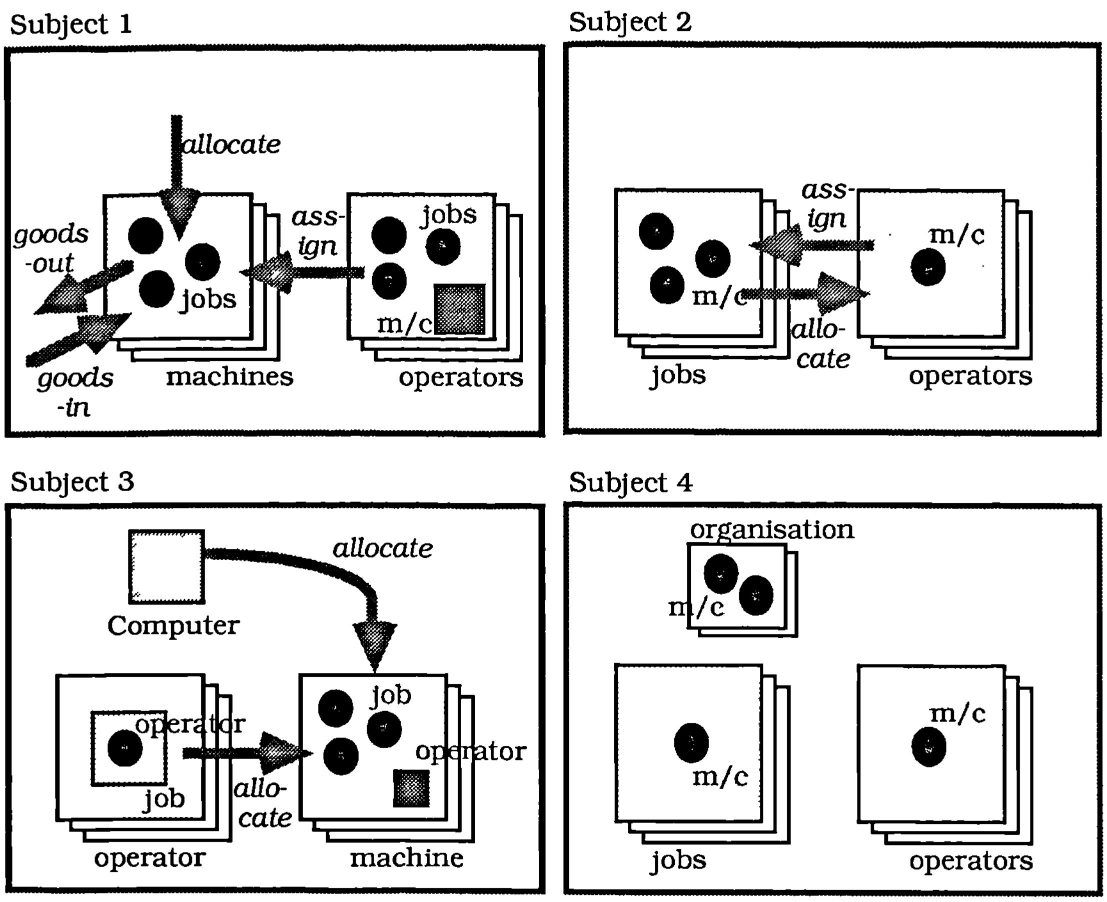


Figure 6.1 - graphic interpretations by the experimenter of the domain descriptions described by subjects on the provided answer sheets

The non-analogical example was expected to aid fact acquisition. S1 and S2 claimed retrospectively to find the example easy to understand while other subjects encountered difficulties using the example. For instance S3 claimed that the example was irrelevant to the target domain while S4 had difficulties understanding it and would have preferred a second, more helpful example. These findings suggest that effective fact acquisition may be better served by analogical rather than non-analogical examples. Furthermore, no subject followed the instructional sequence. Instead, they backtracked to their existing domain description throughout the session to add or modify facts (e.g. Guindon 1990). They also tended to read ahead to determine the goal of the instructions in order to better understand them, so explicit representation of the goal structure may assist fact acquisition (e.g. Anderson et al. 1987).

To conclude, the simplified fact acquisition strategies proved ineffective, thus justifying the need for semi-automated rather than free entry fact acquisition strategies. The final version of the prototype fact acquisition and matching strategies is described in the next section.

6.2.2 The Prototype Problem Identifier

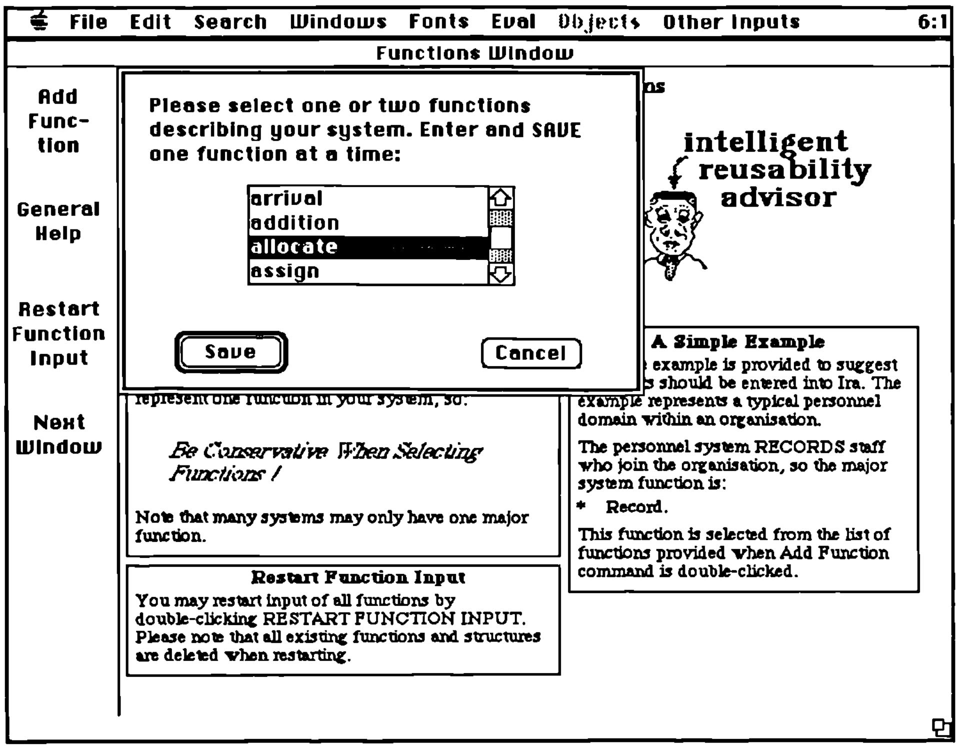
The need for complex fact acquisition strategies was demonstrated by the findings from the previous study. To recap these more complex strategies include sketching the domain to encourage visualisation of key facts, provision of analogical examples to assist schema induction, active explanation of terms for representing the domain and explicit representation of the fact acquisition goal structure (Anderson et al. 1987). The problem identifier is specified in two parts.

6.2.2.1 The Fact Acquisition Dialogue

The fact acquisition dialogue consists of four sequential phases similar to those defined in chapter 5, although there are several differences from the original specification. Most importantly, the prototype lacked facilities for diagrammatic entry of key state transitions and object structures. Instead, software engineers sketched the domains using pen and paper, then entered key facts from these diagrams in text form. The software engineers can abandon fact entry and begin again if required. Pull-down menus are provided to add, modify or delete previously entered facts throughout the dialogue, although access to these menus is controlled so that unexplained facts cannot be entered. Safeguards are also built in to ensure fact consistency, for instance deletion of a key state transition cannot be achieved without first deleting the conditions which trigger the state transition. Finally, the simple retrieval mechanism was streamlined so that domain abstractions were retrieved initially by matched functions or events only.

Selecting System Functions and Domain Events

Key system functions and domain events are selected from scroll menus and matched by function/event using the simple retrieval mechanism, see Figure 6.2.


File Edit Search Windows Fonts Eval Objects Other Inputs 6:1
Functions Window
Add Function
General Help
Restart Function Input
Next Window
Save
Cancel
intelligent reusability advisor

Please select one or two functions describing your system. Enter and SAVE one function at a time:

arrival
addition
allocate
assign

represent one function in your system, so:

Be Conservative When Selecting Functions !

Note that many systems may only have one major function.

Restart Function Input

You may restart input of all functions by double-clicking RESTART FUNCTION INPUT. Please note that all existing functions and structures are deleted when restarting.

A Simple Example

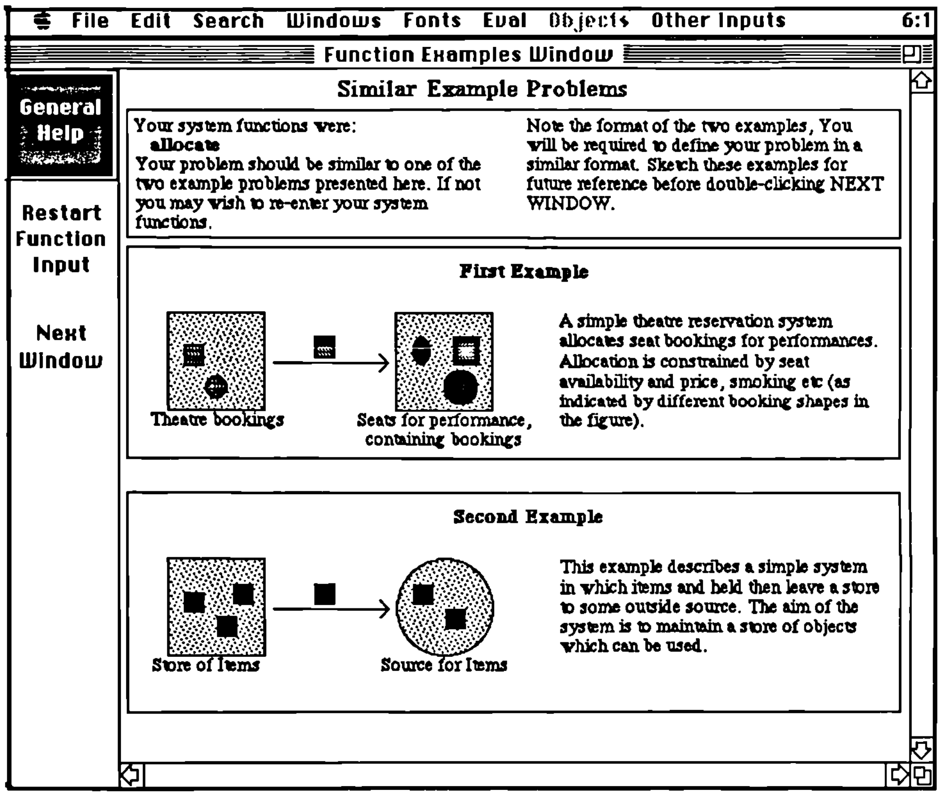
example is provided to suggest s should be entered into Ira. The example represents a typical personnel domain within an organisation.

The personnel system RECORDS staff who join the organisation, so the major system function is:

* Record.

This function is selected from the list of functions provided when Add Function command is double-clicked.

Figure 6.2 - example screen demonstrating selection of the ALLOCATE function using Ira's prototype



Similar Example Problems

Your system functions were:
allocate
Your problem should be similar to one of the two example problems presented here. If not you may wish to re-enter your system functions.

Note the format of the two examples, You will be required to define your problem in a similar format. Sketch these examples for future reference before double-clicking NEXT WINDOW.

First Example

A simple theatre reservation system allocates seat bookings for performances. Allocation is constrained by seat availability and price, smoking etc (as indicated by different booking shapes in the figure).

Second Example

This example describes a simple system in which items and held then leave a store to some outside source. The aim of the system is to maintain a store of objects which can be used.

Figure 6.3 - prototype's screen presenting analogical examples of the retrieved domain abstraction. Examples are also represented graphically to emphasise key state transitions and object structures

Explanation of Domain Abstractions and their Analogical Examples

The prototype presents analogical examples for the two best fitting domain abstractions, see Figure 6.3. One of these examples is selected by the software engineer as a match for the current domain. The top example shown in Figure 6.3 represents an analogical match with the theatre domain. Better mental analogical understanding and schema induction is prompted by requesting the software engineer to sketch the two examples and the current domain using the same presented spatial representation.

Fact Acquisition

Key fact types are entered in text form into the prototype through a series of dialogues defined in Figure 6.4 and exemplified in Figures 6.5, 6.6 & 6.7.

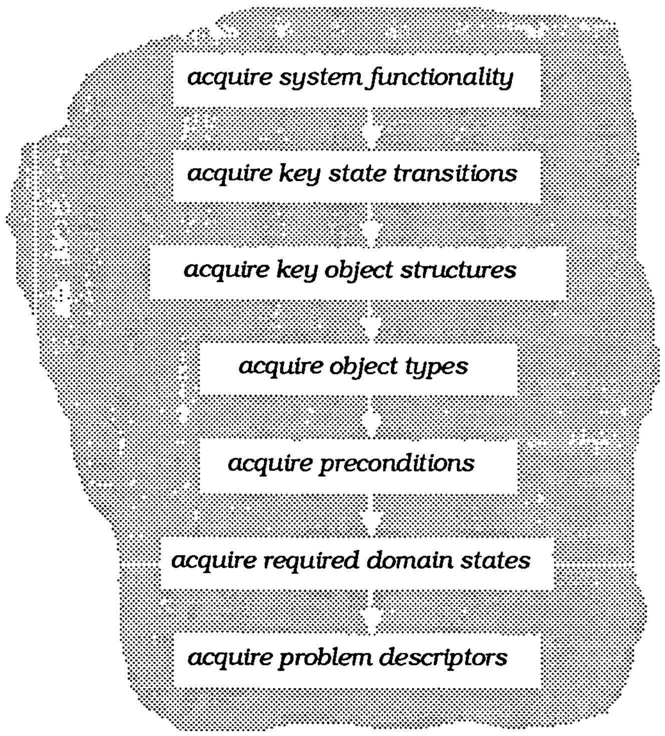
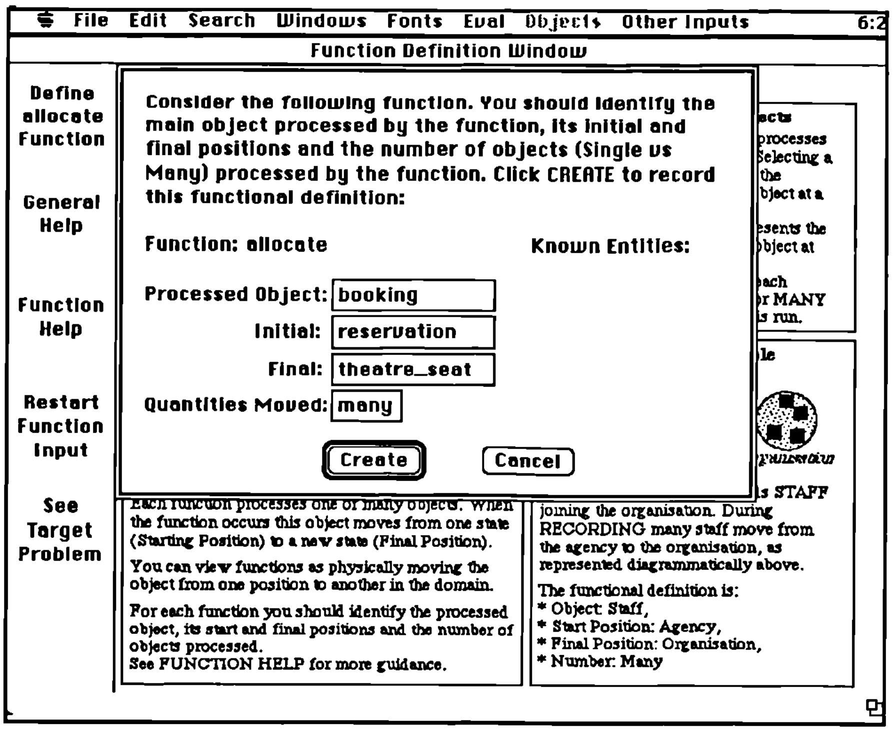

acquire system functionality
acquire key state transitions
acquire key object structures
acquire object types
acquire preconditions
acquire required domain states
acquire problem descriptors

Figure 6.4 - the fact acquisition sequence within the problem identifier



Consider the following function. You should identify the main object processed by the function, its initial and final positions and the number of objects (Single vs Many) processed by the function. Click CREATE to record this functional definition:

Function: allocate

Known Entities:

Processed Object: booking

Initial: reservation

Final: theatre_seat

Quantities Moved: many

Each function processes one or many objects. When the function occurs this object moves from one state (Starting Position) to a new state (Final Position).

You can view functions as physically moving the object from one position to another in the domain.

For each function you should identify the processed object, its start and final positions and the number of objects processed.
See FUNCTION HELP for more guidance.

ects
processes
Selecting a
the
bject at a

esents the
bject at

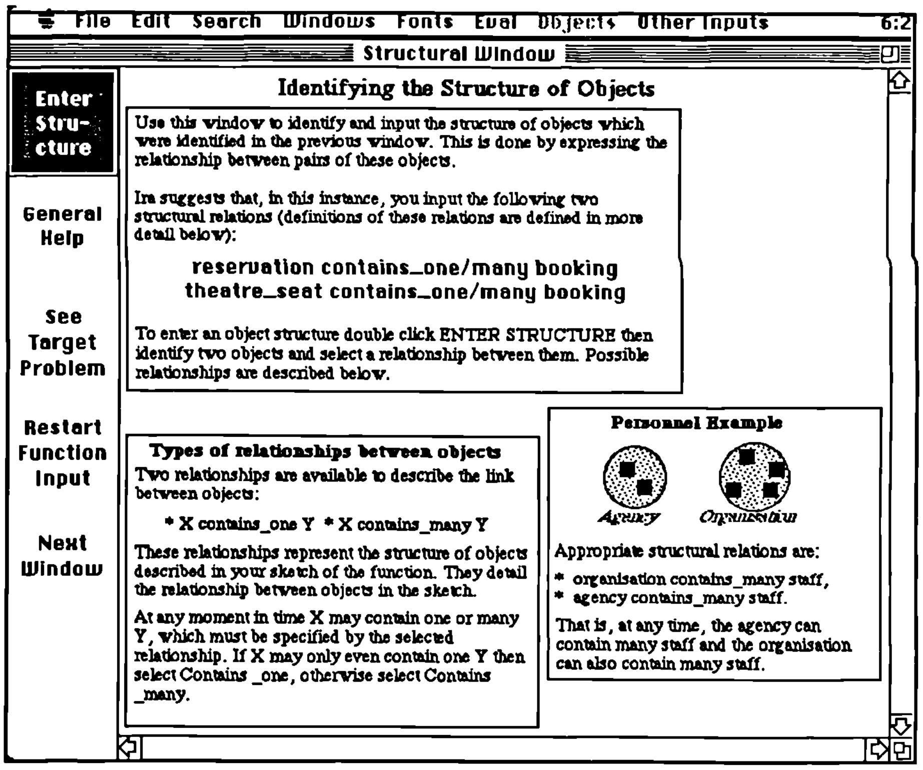
each
r MANY
s run.

le

s STAFF
joining the organisation. During RECORDING many staff move from the agency to the organisation, as represented diagrammatically above.

The functional definition is:
* Object: Staff,
* Start Position: Agency,
* Final Position: Organisation,
* Number: Many

Figure 6.5 - prototype's screen for acquiring key state transitions



Identifying the Structure of Objects

Use this window to identify and input the structure of objects which were identified in the previous window. This is done by expressing the relationship between pairs of these objects.

Ira suggests that, in this instance, you input the following two structural relations (definitions of these relations are defined in more detail below):

reservation contains_one/many booking
theatre_seat contains_one/many booking

To enter an object structure double click ENTER STRUCTURE then identify two objects and select a relationship between them. Possible relationships are described below.

Types of relationships between objects

Two relationships are available to describe the link between objects:

* X contains_one Y * X contains_many Y

These relationships represent the structure of objects described in your sketch of the function. They detail the relationship between objects in the sketch.

At any moment in time X may contain one or many Y, which must be specified by the selected relationship. If X may only even contain one Y then select Contains _one, otherwise select Contains _many.

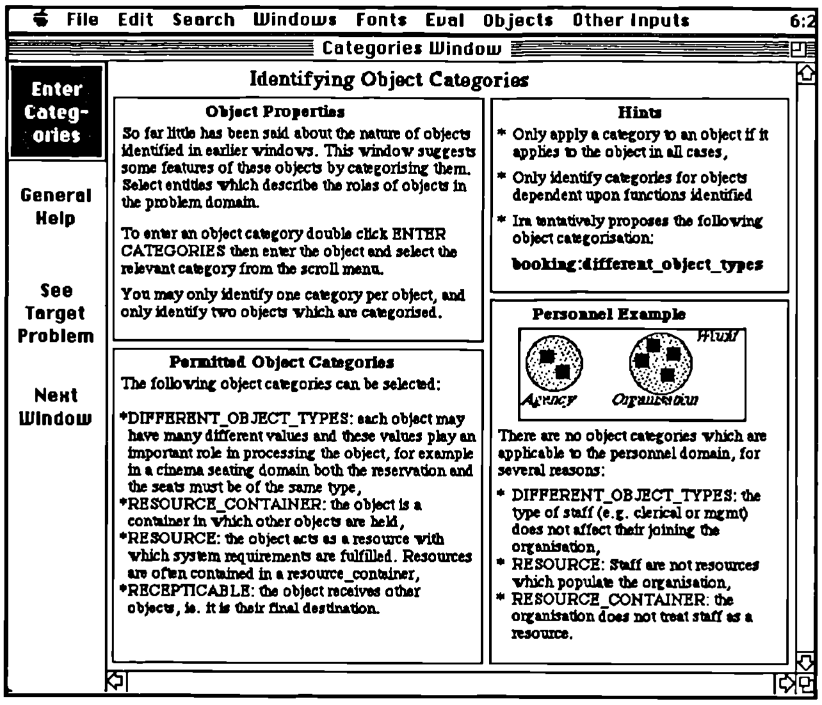
Personnel Example

Appropriate structural relations are:

* organisation contains_many staff,
* agency contains_many staff.

That is, at any time, the agency can contain many staff and the organisation can also contain many staff.

Figure 6.6 - prototype's screen for acquiring key object structures, including hints based on the retrieved domain abstraction, see section 5.3.1



Identifying Object Categories

Object Properties

So far little has been said about the nature of objects identified in earlier windows. This window suggests some features of these objects by categorising them. Select entities which describe the roles of objects in the problem domain.

To enter an object category double click ENTER CATEGORIES then enter the object and select the relevant category from the scroll menu.

You may only identify one category per object, and only identify two objects which are categorised.

Permitted Object Categories

The following object categories can be selected:

*DIFFERENT_OBJECT_TYPES: each object may have many different values and these values play an important role in processing the object, for example in a cinema seating domain both the reservation and the seats must be of the same type,
*RESOURCE_CONTAINER: the object is a container in which other objects are held,
*RESOURCE: the object acts as a resource with which system requirements are fulfilled. Resources are often contained in a resource_container,
*RECEPTICABLE: the object receives other objects, ie. it is their final destination.

Hints

* Only apply a category to an object if it applies to the object in all cases,
* Only identify categories for objects dependent upon functions identified
* Ira tentatively proposes the following object categorisation:

booking:different_object_types

Personnel Example

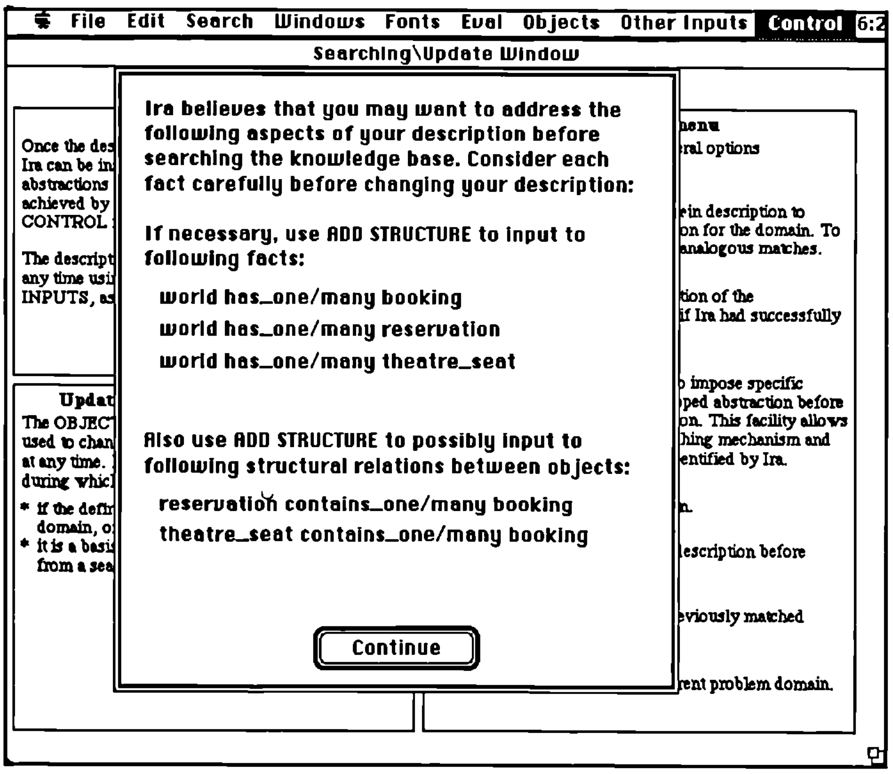
There are no object categories which are applicable to the personnel domain, for several reasons:

* DIFFERENT_OBJECT_TYPES: the type of staff (e.g. clerical or mgmt) does not affect their joining the organisation,
* RESOURCE: Staff are not resources which populate the organisation,
* RESOURCE_CONTAINER: the organisation does not treat staff as a resource.

Figure 6.7 - prototype's screen for acquiring object types

Evaluation of the Domain Description

The domain description is evaluated by matching it to the domain abstraction retrieved earlier. This evaluation identifies omitted state transitions and object structures, see Figure 6.8. The evaluation is supported by prompts to the software engineer to evaluate the rest of the domain description.


File Edit Search Windows Fonts Eval Objects Other Inputs Control
Searching\Update Window
Continue

Ira believes that you may want to address the following aspects of your description before searching the knowledge base. Consider each fact carefully before changing your description:

If necessary, use ADD STRUCTURE to input to following facts:

world has_one/many booking
world has_one/many reservation
world has_one/many theatre_seat

Also use ADD STRUCTURE to possibly input to following structural relations between objects:

reservation contains_one/many booking
theatre_seat contains_one/many booking

Figure 6.8 - prototype's screen identifying possible omitted state transitions and object structures from the domain description

Fact Acquisition Dialogues: A Summary

The prototype implements most of the specified problem identifier from chapter 5 using the 40 dialogues whose network is defined in Figure 6.9. The resulting domain description is passed to the analogy engine to be matched to all known domain abstractions. Dialogues for browsing, understanding and selecting between domain abstractions are defined in the next section.

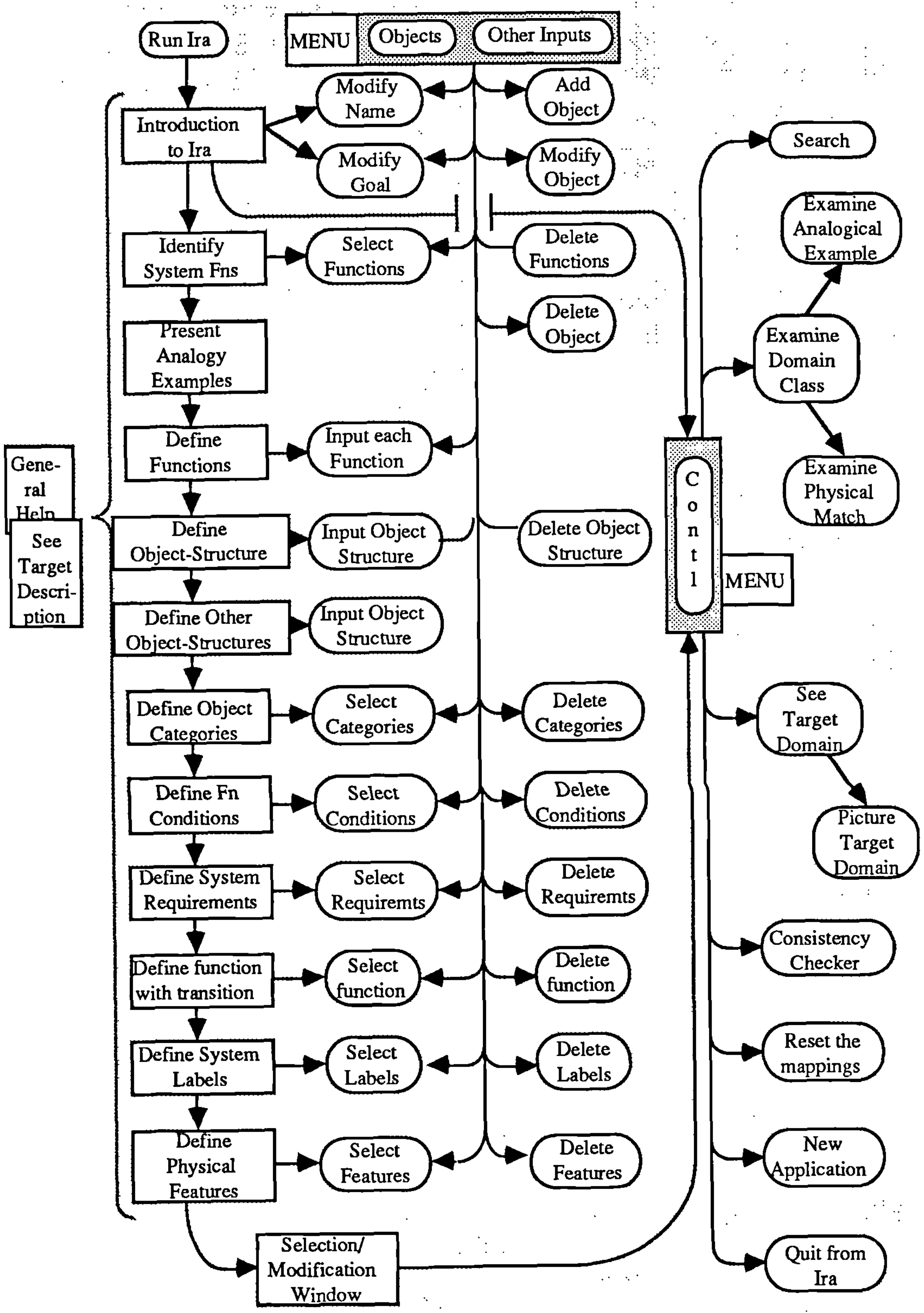


Figure 6.9 - high-level dialogue network showing all screens belonging to the prototype problem identifier.

6.2.2.2 Retrieval of Domain Abstractions

The prototype implements a two-phase strategy for domain abstraction retrieval. This strategy is shown in Figure 6.10. Two phases were implemented to allow for the incompleteness and inconsistency of domain descriptions entered by software engineers. The second phase permits improvement of the domain description in light of feedback from candidate analogical matches. There are several possible results from an analogical match:

- *a failed match*: the problem identifier informs the software engineer of insufficient facts for an analogical match;
- *good matches*: retrieved domain abstractions are explained;
- *a partial match*: the problem identifier acquires additional domain facts necessary for a good match. Unmapped facts in the domain abstraction are assumed to have equivalent facts in the target domain, so the problem identifier proposes candidate state transitions and object structures to complete the domain description which then can be accepted or rejected by the software engineer.

The implementation of this algorithm is shown in Appendix L and an example of the prototype's response to a partial analogical match is shown in Figure 6.11. Failure to match the domain description at the second attempt is assumed to indicate that analogical matching is unsuccessful, although future versions of the advisor will allow more iterative fact acquisition and retrieval of domain abstractions.

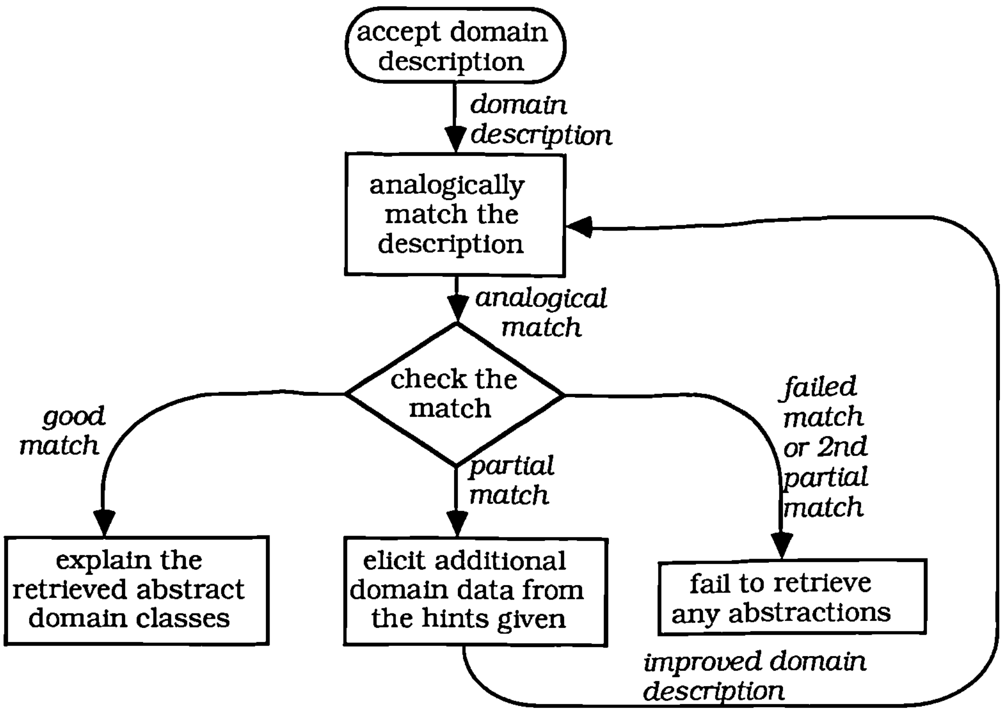
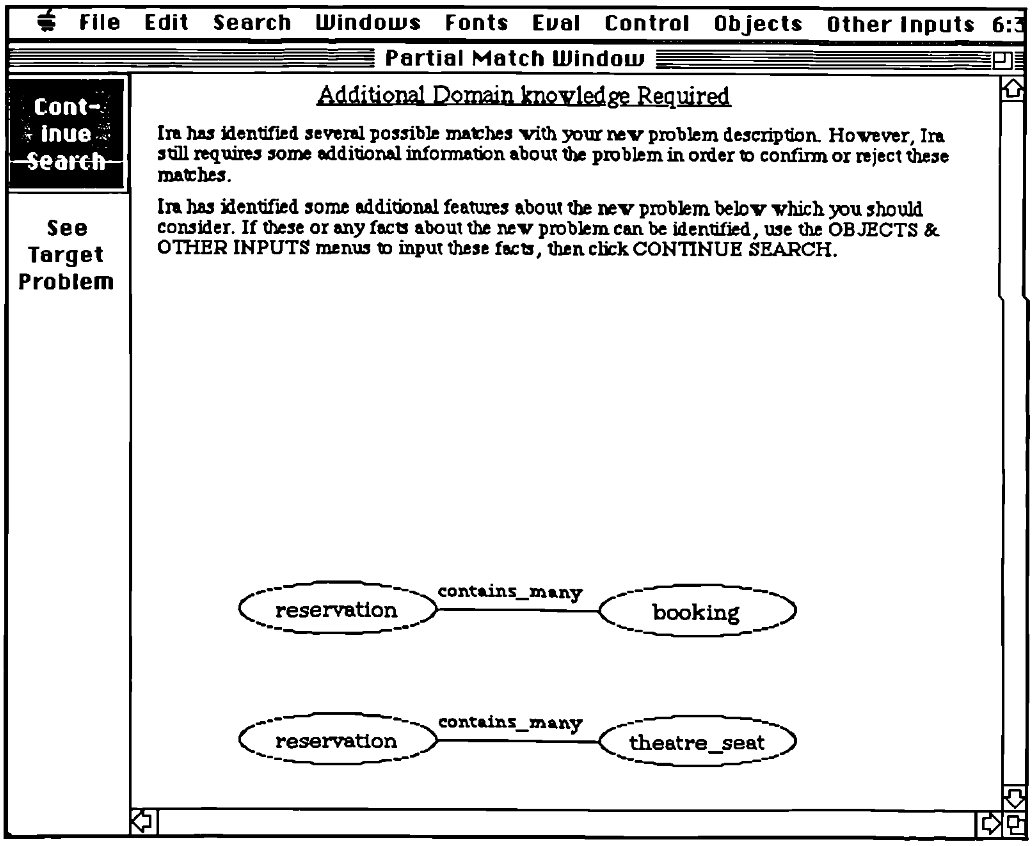

accept domain description
domain description
analogically match the description
analogical match
check the match
good match
partial match
failed match or 2nd partial match
explain the retrieved abstract domain classes
elicit additional domain data from the hints given
fail to retrieve any abstractions
improved domain description

Figure 6.10 - prototype's analogical retrieval strategy


File Edit Search Windows Fonts Eval Control Objects Other Inputs 6:3
Partial Match Window
Cont-inue Search
See Target Problem

Additional Domain knowledge Required

Ira has identified several possible matches with your new problem description. However, Ira still requires some additional information about the problem in order to confirm or reject these matches.

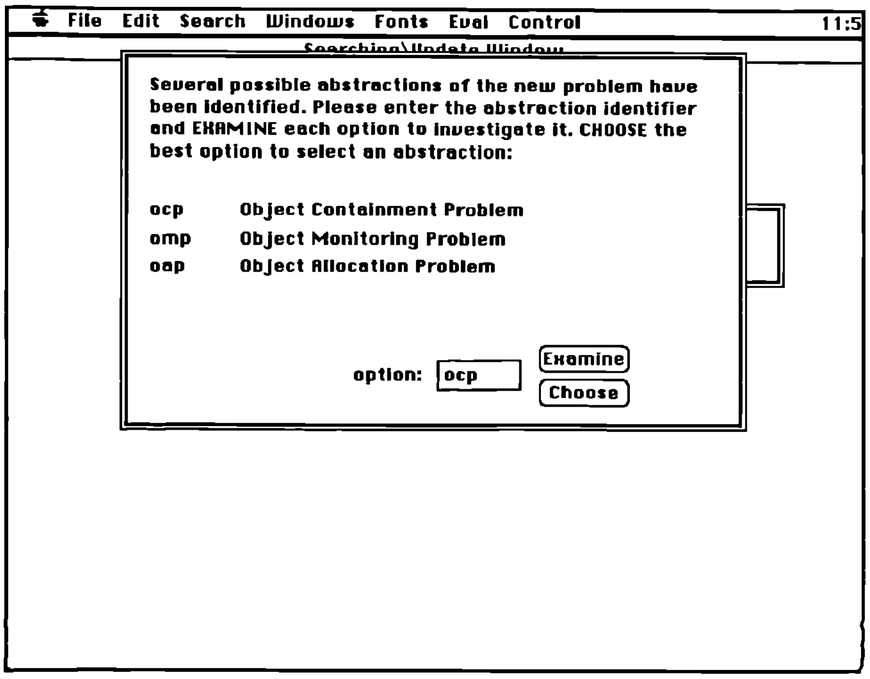
Ira has identified some additional features about the new problem below which you should consider. If these or any facts about the new problem can be identified, use the OBJECTS & OTHER INPUTS menus to input these facts, then click CONTINUE SEARCH.

Figure 6.11 - prototype's screen describing likely omissions from the domain description in the case of a partial analogical match

6.2.2.3 Explanation of Domain Abstractions

The prototype explains domain abstractions so that they can be understood and selected. Strategies are also needed to browse and explain the domain abstractions retrieved by the analogy engine. Currently the prototype only explains analogical mappings with each retrieved abstraction, although future versions will also explain key differences between domain abstractions. The software engineer can browse each retrieved abstraction in turn, see Figure 6.12. Subsequent explanation of each abstraction is achieved using the same tactics as implemented in the problem identifier. Spatial representations and narrative description of key facts about the abstraction are supported by descriptions of inferred analogical mappings with each abstract object. The software engineer can also browse well understood examples representing analogical instantiations of the domain abstraction. Furthermore, the software engineer is presented with analogical examples belonging to the same generic domain world. As such, the prototype permits guided discovery and learning of domain abstractions, see Figure 6.13.

The prototype has an override facility which permits experimentation with the retrieved abstraction by allowing the software engineer to select and fix individual mappings during analogical matching. This facility is particularly useful if the analogy engine infers mappings perceived by the software engineer to be incorrect. In such cases the offending mapping can be corrected by the software engineer and the domain description rematched, as shown in Figure 6.14. Future versions of the advisor will incorporate further embedded explanatory knowledge to permit more effective explanation of analogical matches.

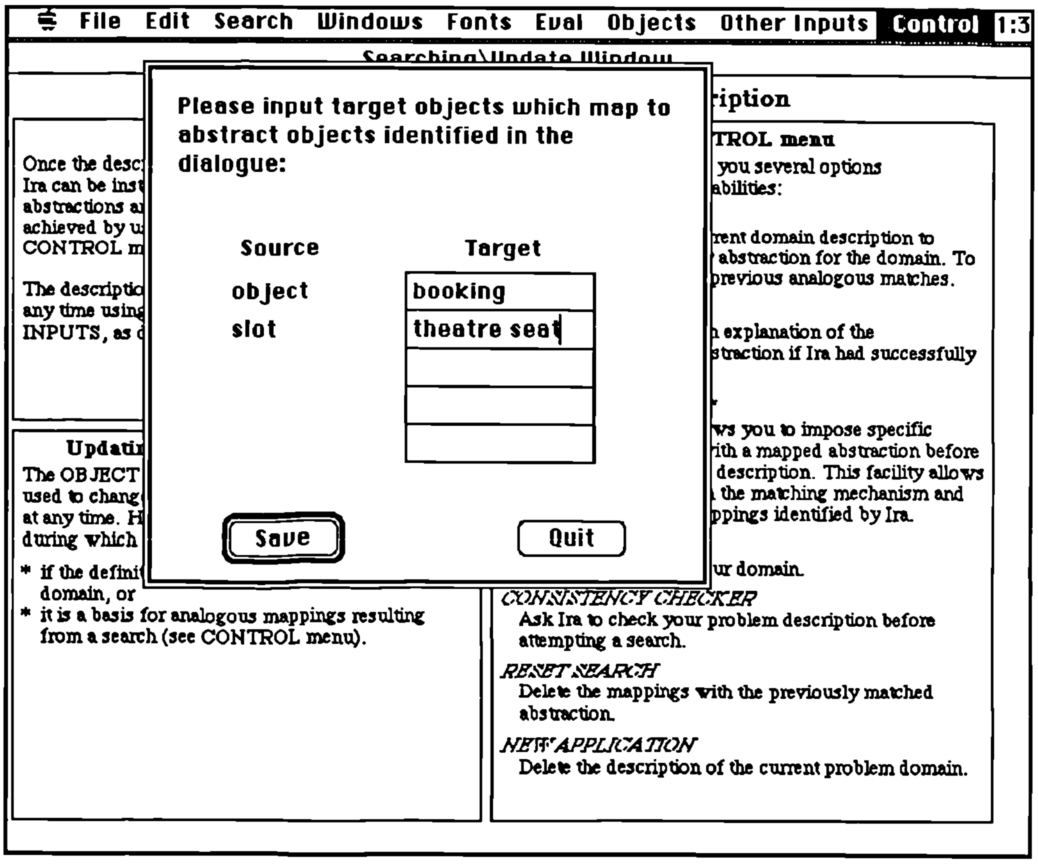


Several possible abstractions of the new problem have been identified. Please enter the abstraction identifier and EXAMINE each option to investigate it. CHOOSE the best option to select an abstraction:

ocp	Object Containment Problem
omp	Object Monitoring Problem
oap	Object Allocation Problem

option: ocp

Figure 6.12 - prototype's screen for browsing retrieved domain abstractions


File Edit Search Windows Fonts Eval Objects Other Inputs Control
1:3
Searching\Update Window
Save
Quit

Please input target objects which map to abstract objects identified in the dialogue:

Source	Target
object	booking
slot	theatre seat

Once the desc
Ira can be inst
abstractions a
achieved by u
CONTROL m

The descriptio
any time using
INPUTS, as d

Updati
The OBJECT
used to chang
at any time. H
during which

* if the defini
domain, or
* it is a basis for analogous mappings resulting from a search (see CONTROL menu).

ription

TROL menu
you several options
abilities:

rent domain description to
abstraction for the domain. To
previous analogous matches.

n explanation of the
straction if Ira had successfully

ws you to impose specific
ith a mapped abstraction before
description. This facility allows
the matching mechanism and
ppings identified by Ira.

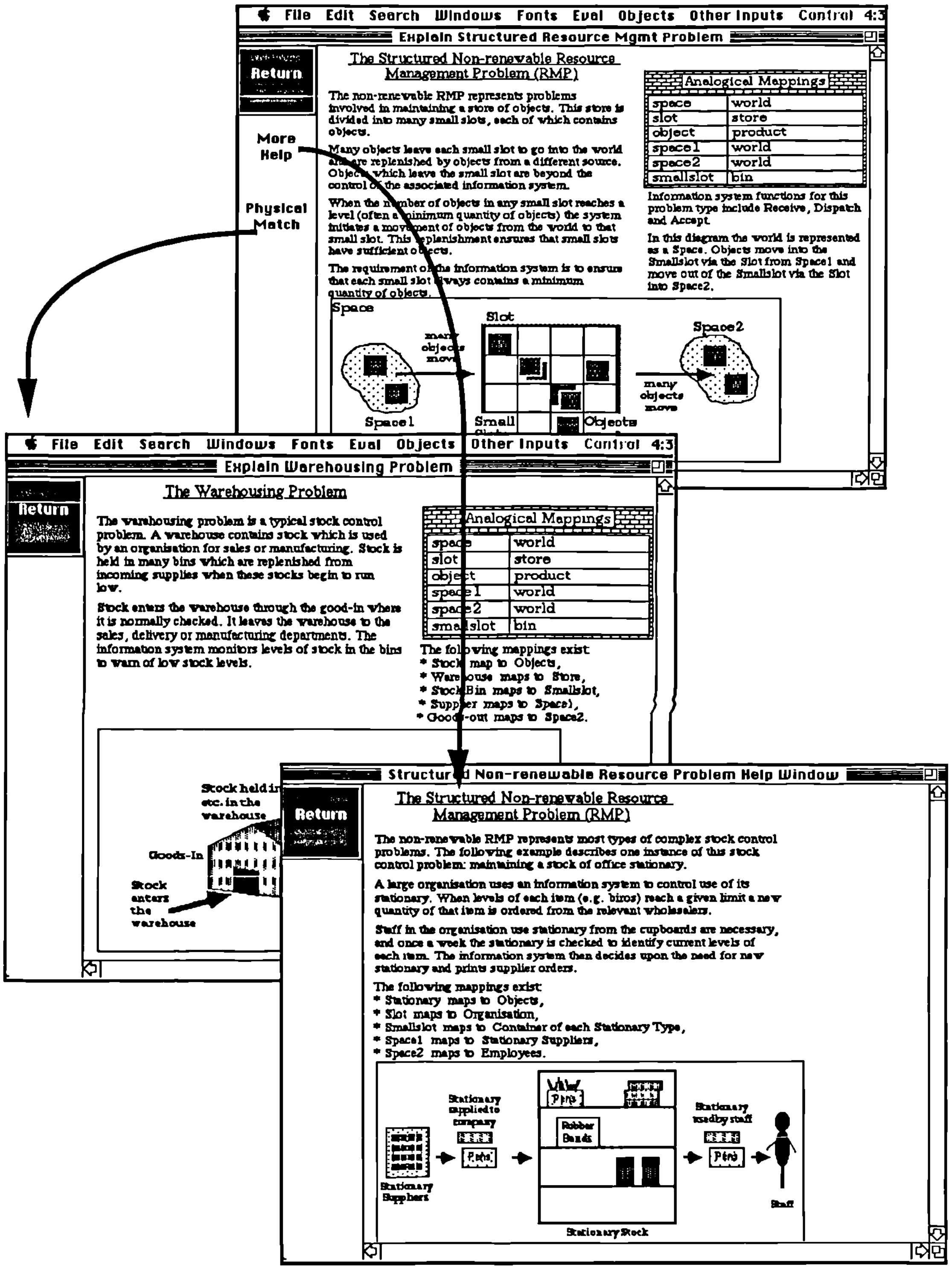
ur domain.

CONSISTENCY CHECKER
Ask Ira to check your problem description before attempting a search.

RESET SEARCH
Delete the mappings with the previously matched abstraction.

NEW APPLICATION
Delete the description of the current problem domain.

Figure 6.14 - prototype's screen for inputting analogical mappings which override inferred analogical mappings


File Edit Search Windows Fonts Eval Objects Other Inputs Control 4:3
Explain Structured Resource Mgmt Problem
Return
More Help
Physical Match
Space
Slot
many objects move
Space2
many objects move
Space1
Small Slots
Objects
File Edit Search Windows Fonts Eval Objects Other Inputs Control 4:3
Explain Warehousing Problem
Return
Stock held in etc. in the warehouse
Goods-In
Stock enters the warehouse
Structured Non-renewable Resource Problem Help Window
Return
Stationary supplied to company
Pens
Rubber Bands
Stationary used by staff
Stationary Suppliers
Stationary Stock
Staff

The Structured Non-renewable Resource Management Problem (RMP)

The non-renewable RMP represents problems involved in maintaining a store of objects. This store is divided into many small slots, each of which contains objects.

Many objects leave each small slot to go into the world and are replenished by objects from a different source. Objects which leave the small slot are beyond the control of the associated information system.

When the number of objects in any small slot reaches a level (often a minimum quantity of objects) the system initiates a movement of objects from the world to that small slot. This replenishment ensures that small slots have sufficient objects.

The requirement of the information system is to ensure that each small slot always contains a minimum quantity of objects.

Analogical Mappings

space	world
slot	store
object	product
space1	world
space2	world
smallslot	bin

Information system functions for this problem type include Receive, Dispatch and Accept.

In this diagram the world is represented as a Space. Objects move into the Smallslot via the Slot from Space1 and move out of the Smallslot via the Slot into Space2.

The Warehousing Problem

The warehousing problem is a typical stock control problem. A warehouse contains stock which is used by an organisation for sales or manufacturing. Stock is held in many bins which are replenished from incoming supplies when these stocks begin to run low.

Stock enters the warehouse through the good-in where it is normally checked. It leaves the warehouse to the sales, delivery or manufacturing departments. The information system monitors levels of stock in the bins to warn of low stock levels.

Analogical Mappings

space	world
slot	store
object	product
space1	world
space2	world
smallslot	bin

The following mappings exist:
* Stock map to Objects,
* Warehouse maps to Store,
* StockBin maps to Smallslot,
* Supplier maps to Space1,
* Goods-out maps to Space2.

The Structured Non-renewable Resource Management Problem (RMP)

The non-renewable RMP represents most types of complex stock control problems. The following example describes one instance of this stock control problem: maintaining a stock of office stationary.

A large organisation uses an information system to control use of its stationary. When levels of each item (e.g. biros) reach a given limit a new quantity of that item is ordered from the relevant wholesalers.

Staff in the organisation use stationary from the cupboards are necessary, and once a week the stationary is checked to identify current levels of each item. The information system then decides upon the need for new stationary and prints supplier orders.

The following mappings exist:
* Stationary maps to Objects,
* Slot maps to Organisation,
* Smallslot maps to Container of each Stationary Type,
* Space1 maps to Stationary Suppliers,
* Space2 maps to Employees.

Figure 6.13 - prototype's screens for explaining a stock control's domain abstraction.
The three screens, from top to bottom, represent: the key domain abstraction; a likely generic domain world, and a well-understood concrete example

6.3 Evaluation of the Prototype

The prototype's effectiveness at acquiring key domain facts and matching them to domain abstractions was measured in a realistic problem scenario. Evaluation was achieved through observation of inexperienced software engineers using the prototype. To add a realistic challenge to this evaluation, the software engineers were given no prior training or exposure to the prototype and were requested to retrieve and understand one of the most complex abstractions known to the advisor.

6.3.1 Method

Four software engineers with moderate or little systems analytic experience used the prototype to investigate the production planning domain (see Appendix F) then understand and select retrieved domain abstractions. Subjects were doctoral students and junior lecturers in the Department of Business Computing at City University with experience of structured analytic techniques (e.g De Marco 1978) equivalent to that possessed by subjects who used to the paper based mockups described in section 6.2.1. Each subject was given a one-page description of the production planning domain shown in Appendix F and an overview of the fact acquisition sequence. Coloured pens and A3 paper simulated the advisor's undeveloped diagramming facility, allowing subjects to sketch the target domain and analogical examples. They had one hour to analyse and describe the production planning domain followed by five minutes to understand and select retrieved domain abstractions. Upon completion of this task the experimenter retrospectively elicited further details of subjects' behaviour using a written questionnaire and verbal questioning. Finally, the prototype's effectiveness was measured by the *goodness of fit* of retrieved domain abstractions, two of which represented the production planning domain, see Appendix J. The prototype was deemed successful if it retrieved either of these two domain abstractions.

6.3.2 Results

All four subjects (S5-S8) entered domain facts but only three of these descriptions matched a correct domain abstraction while S7's description failed to match any abstraction. Successful subjects understood the retrieved abstractions and matched them to the production planning domain. Performance represented a noticeable improvement over that from the free form entry dialogues reported in section 6.1.

Reasons for Subjects' Success

Subjects who used diagrams to represent and understand their domains also developed more accurate and complete descriptions of that domain, although several discrepancies between subjects' diagrams and domain definitions did occur. In addition, S5 retrospectively admitted that diagramming techniques led to considerable improvements in the prototype's usability while S8 claimed to have sketched the production planning domain directly from an example retrieved by the prototype. The need for further diagramming facilities was also emphasised by S8, who made extensive notes and sketches of the domain prior to fact input into the problem identifier. Indeed, S8 reused her existing sketch of a retrieved example to model the target domain, suggesting the importance of visualisation through diagrams.

The prototype promotes mental abstraction by partial exposure to pictorially represented concrete examples, so the effectiveness of these key strategies was examined more closely. Successful subjects claimed to understand the relevance of retrieved examples to production planning while the unsuccessful subject failed to understand the example or map it to the production planning domain. Indeed, S5 claimed that the example was well-explained. However, a note of caution should be sounded, since S8 retrospectively claimed to rely too much on the example and extended similarities with the example beyond the validity of the 'analogical' match (Halsasz & Moran 1982, De Boulay 1989). Finally, successful domain description also depended on effective use of visualisation and immediate feedback. The retrospective questionnaire revealed that these strategies resulted in more effective acquisition of key state transitions and object structures compared with other knowledge types.

Problems Encountered by Subjects

Despite the qualified success of the study, all subjects encountered difficulties while using the prototype. The failed subject (S7) succeeded initially in describing key state transitions and object structures which, if matched at that moment, would have retrieved the correct abstraction. However, he subsequently extended this description to include incorrect and unnecessary facts. Retrospective questioning revealed that the subject found the description too simple in comparison with structured analysis notations such as data flow diagrams (De Marco 1978), so he overspecified his domain description and added a second state transition and further object structures to describe entity-relationships between domain objects. The result was to obscure key domain facts so that the analogy engine was unable to retrieve any abstraction. Interference from structured analytic techniques presented an unforeseen problem for the prototype. Indeed, both S7 and S8 expressed doubts about the simplicity of the domain description in comparison with complex models developed using structured analytic techniques. Further explanatory dialogues will be necessary to justify the scale of domain descriptions acquired by the prototype. Another problem encountered in this study was mental laziness: S7 admitted to being lazy and copied the retrieved examples during the latter stages of domain description, despite the strategies implemented in the prototype. These findings suggest that mental laziness will be difficult to discourage.

6.3.3 Summary of the Evaluation

Evaluation of the prototype proved successful in that most software engineers were able to understand and select the correct domain abstraction for a new application. Presentation of concrete examples, visualisation of domains through sketching, guided fact acquisition and partial exposure all proved effective for three of the four subjects. Although the scale of this study is small, results do indicate that retrieval of domain abstractions and concrete examples was effective. In addition, the tool assists software engineers to model the key facts about a new domain which can then elaborated during subsequent phases of requirements engineering.

6.4 Success of the Prototype

Overall, the prototype problem identifier and analogy engine were a qualified success. Analogical examples assisted understanding of domain abstractions while spatial representations aided both understanding of these abstractions and domain structuring and scoping, although some mental laziness manifest as copying did occur. Indeed, early presentation of examples was necessary to explain the otherwise difficult concept of state transitions with respect to object structures to software engineers. The analogical examples retrieved by the prototype permit problem formulation and understanding prior to reuse of more comprehensive analogical specifications. Furthermore, mechanised fact acquisition from a restricted set of domain terms proved effective. Thus, the development and evaluation of this prototype has implications for requirements engineering as well as analogical specification reuse.

Evaluation of the prototype did reveal several problems which remain to be solved. First, copying of analogical examples remains problematic, despite incorporating several strategies into the prototype to inhibit it. Further research of the tradeoff between knowledge provision and mental laziness during problem exploration is needed. Second, the prototype's fact acquisition strategies must be integrated with existing structured analytic techniques, both to provide a framework for mechanised requirements engineering and to avoid interference with structured notations during acquisition of key domain facts. These implications are discussed more fully in chapter 7.

The implemented analogy engine retrieved the correct domain abstractions from key facts acquired from software engineers, although these descriptions were only partially complete and consistent. This success underlined the robustness of the analogy engine. However, a note of caution should be sounded since the search space of domain abstractions was constrained for the purposes of the prototype. Problems may be encountered when analogical matching is scaled up to searching many domain abstractions. This additional functionality for Ira is discussed further in chapter 7.

Chapter 7

7: Conclusions and Future Work

This research proposed analogical specification reuse as a paradigm for improving the productivity of the requirements engineering process and quality of its artifacts. Automated analogical specification reuse is beyond the knowledge and reasoning capabilities of software engineering environments, so a cooperative paradigm exploiting the knowledge of support tools and skills of the software engineer is proposed. The validity of this paradigm was demonstrated by empirical studies of analogical specification reuse, the logical model of software engineering analogies and its computational implementation. Reuse of readily-available specifications represents an advance over existing keyword and object-oriented reuse paradigms by exploiting the rich seam of domain and method knowledge held in specifications. Indeed, it goes beyond the use of small-scale components by retrieving large artifacts for wholesale customisation, thus providing greater improvements in productivity and quality. This chapter summarises the work reported in the thesis and emphasises the benefits of a multi-disciplinary paradigm for effective specification reuse. Possible extensions to the model of domain abstraction in requirements engineering are followed by a discussion of the current limitations of the analogical specification reuse paradigm. The thesis is concluded by a review of future research directions.

7.1 A Multi-disciplinary Paradigm

A multi-disciplinary paradigm for analogical specification reuse is necessary otherwise effective reuse cannot be achieved. This thesis proposes a cognitive engineering approach (Woods & Roth 1988) to exploit the different skills and knowledge possessed by software engineers and support tools (e.g. Kolodner 1991). This is possible by dividing the work between the software engineer and tool then designing the advisor in light of this division. The deliverables of the research reflect its multi-disciplinary nature:

- a logical model of software engineering analogies which justifies the existence of such analogies. This model supports retrieval and explanation of analogical specifications;
- a set of domain abstractions representing key determinants of software engineering analogies derived from example-based analyses of these analogies;
- cognitive task and reasoning models of analogical reuse and mental models of

analogical comprehension exhibited by both inexperienced and expert software engineers. These models provide an empirical basis for the design of tool support during analogical recognition, comprehension and transfer;

- cooperative, tool-based strategies for the retrieval, understanding and customisation of analogical specifications derived from empirical studies of analogical reuse reported in this thesis and studies of cooperative problem solving reported elsewhere;
- the design of an intelligent advisor informed by the logical model of software engineering analogies, cognitive task and reasoning models of analogical reuse, mental models of analogical understanding and existing guidelines for the design of intelligent tutoring systems;
- a partially-implemented prototype of the intelligent reuse advisor;
- a successful evaluation of this prototype to demonstrate the effectiveness of the proposed cooperative paradigm for specification retrieval and fact acquisition during requirements engineering.

Problems with existing software and knowledge reuse paradigms introduced in chapter 2 are reviewed, then the role of cooperative analogical reasoning in overcoming these problems is discussed.

7.1.1 Why a Cooperative, Analogical Reasoning Paradigm

Existing software reuse paradigms fail to support effective specification reuse for two reasons. First, current paradigms assume complete and correct domain knowledge for component matching and retrieval, however acquiring and modelling this knowledge is, to say the least, difficult. Second, they propose automation of software reuse based on complex reasoning mechanisms and complete domain knowledge which overlook the fact that humans often possess more domain knowledge and are better reasoners than existing automated reasoners. On the other hand, analogy lends itself to a cooperative paradigm which emphasises the importance of human reasoning during analogical recognition, comprehension and transfer. This is confirmed by the current trend towards cooperative case-based reasoners for problem solving (e.g. Ashley 1991, Branting 1991) rather than fully automated computational implementations of analogical reasoning (e.g Hall 1989).

7.1.2 The Benefits of Cooperation

This research claims several benefits from a cooperative, analogical reasoning paradigm for specification reuse. First, complex analogical retrieval of components represents an advance over existing, inadequate keyword classification techniques. Second, the knowledge acquisition bottleneck is overcome by modelling domain abstractions (e.g. Greiner 1988a, 1988b) rather than numerous concrete instances. Third, analogical understanding emphasises mapping between source and target domains. This mapping process is sadly neglected in current software reuse paradigms. On the other hand, human analogical understanding and transfer have received more research attention, so analogical problem solving provides a richer theoretical and empirical basis for informing design of software reuse tools. Fourth, analogical transfer occurs between domains sharing notable differences. In this respect analogy is closer to inter-application reuse and more likely to provide important clues about the problems which effective specification reuse must overcome. Emphasising similarities and differences between domains as analogical mappings may also be more likely to highlight the necessity of specification customisation during reuse.

7.1.3 Cooperation During Requirements Engineering

Cooperation is also needed during other software engineering activities, however it must be founded on an understanding of the roles most effectively carried out by machines and software engineers, to assist software engineers in tasks which they find difficult and encourage them to undertake tasks which they do well. Unfortunately few studies of software engineering practice have been reported in the literature, so more empirical investigations of software engineering practice are needed. This research provides a starting point for such a cooperative approach to requirements engineering.

Findings from all four empirical studies have implications for the design of requirements engineering environments. Methods often pay little attention to the early phases of analysis and have been criticised for poor support of requirements engineering (Flynn et al. 1986). First, domain knowledge must be incorporated into such tools to assist structuring, scoping and determination of key domain facts. As already suggested, populating such tools with a set of domain abstractions can provide the mental schemata which inexperienced software engineers do not possess. Hypothesis testing through domain scenarios was another important success factor in the first empirical study, hence tools may benefit by embedding more explicit testing steps within development stages (e.g. Fickas & Nagarajan 1988). Whereas many current software development tools support such testing at the syntactic level of diagram correctness and specification consistency, testing the semantics of specifications is likely to be more important. Here integration of domain and method knowledge in software development tools may be the way forward since introducing such domain knowledge into structured methods appears difficult.

Flexible and easy-to-use electronic notepads (e.g. Haddley & Sommerville 1990) can replace paper-based notetaking. Hypermedia (e.g. Conklin 1987) is one technique which can be used to structure and present domain and method knowledge to the software engineer. Problem structuring can also be enhanced by tool-based techniques which record the software engineer's goals and design rationale (e.g. Mostow 1989), thus extending their working memory as well as eliciting additional knowledge for a wider diagnostic module (e.g. Anderson 1988).

Another implication is to acknowledge the flexibility in human approaches to problem solving and not to rigidly prescribe development procedures and steps. This may be viewed as an argument for methods based on a pick and mix of techniques (the tool-box approach) without strict operational guidance rather than the cook-book approach found in leading methods such as SA/SD (De Marco 1978) and JSD (Jackson 1983). However, tool support may be necessary to assist inexperienced software engineers to select the most appropriate technique. Furthermore, CASE tools must recognise these variations to provide effective support. An intelligent advisor can assist error detection and correction and provide effective strategies during software engineering activities. Another advantage is that the combination of learning and problem solving activities in software development tools permits training of software engineers without removing them from their work place. However, to be effective, design of intelligent advisors should be informed by cognitive task and reasoning models of software engineering behaviour.

To sum, this research has provided an empirical basis for cooperation during analogical specification reuse and evidence of software engineering practice during other activities. However, more empirical research of software engineering practice is needed to inform design of truly effective requirements engineering environments.

7.2 Limitations of a Cooperative Paradigm for Analogical Specification Reuse

The paradigm for analogical specification reuse defined in this thesis is limited. Evaluation of the prototype advisor in chapter 6 demonstrated the effectiveness of cooperation during fact acquisition and specification retrieval while empirical studies reported in chapter 4 indicate that understanding and transferring analogical specifications is both possible and effective. However, these studies do not validate the complete paradigm or examine its full potential, so limitations are examined in more detail.

7.2.1 Scope

The current paradigm only supports reuse between complex but relatively small software engineering domains. To be truly effective, the paradigm must be scaled up in terms of the number and types of matched facts to support reuse between large and complex domains. It has no tool-based facilities to capture and structure more complex requirements or domain descriptions. It will also require more rule-based reasoning by the analogy engine to structure the description and guide exhaustive analogical matching, an approach adopted in existing case-based reasoning tools (e.g. Branting 1991).

During other phases, the specification advisor has no strategies to assist software engineers to select the best fitting specification from retrieved candidates ranked by the advisor. One solution may be to investigate the selection strategies of expert software engineers when choosing the best analogical match from several similar candidates. Common errors made by inexperienced software engineers during the same task can inform the design of support tools for specification selection. This empirical approach to tool design is in keeping with existing strategies which assist specification understanding and transfer. The effectiveness of strategies assisting comprehension and transfer of analogical specifications also remains to be evaluated, especially during reuse of large specifications.

7.2.2 Validity

The validity of the paradigm remains to be shown. The logical model of domain abstraction has withstood the relatively weak test of proof by evaluation with limited examples, however further evaluation is needed to investigate the key issues of the coverage and granularity of domain abstractions for effective reuse. Two possible strategies for validating domain abstractions are to evaluate them against real-world domains found in large organisations or to compare them to mental schemata possessed by expert software engineers. The effectiveness of the empirically derived strategies for specification understanding and transfer defined in chapters 4 and 5 also remains to be confirmed, despite their importance in the proposed paradigm. In addition, further user studies with the prototype advisor are needed to investigate the effectiveness of the fact acquisition strategies, the analogy engine and explanation of domain abstractions during domain description, especially when analysing more complex problems.

7.2.3 Completeness

The completeness of the analogical specification reuse paradigm is dependent on the coverage of its domain abstractions. The current set of domain abstractions is limited to examples from case studies, textbooks, real-world domains and conference case studies. It tends to represent business information rather than real-time applications. Four strategies for evaluating the completeness of the defined set of domain abstractions are available. First, further example-based studies of complex software engineering analogies will extend the coverage of key domain abstractions. Second, these example-based abstractions will be validated by domain modelling in large organisations using interviewing, system examination and reviews of application and organisation documentation (e.g. Prieto-Diaz 1990, Iscoe 1991). Derived domain models can be abstracted using the meta-schema of knowledge types defined in chapter 3 to determine the correctness of the current set of domain abstractions.

Third, domain abstractions can also be evaluated against memory schemata representing domain abstractions possessed by expert software engineers with exposure to diverse software engineering domains. Software engineers with expertise of single applications such as banking may possess domain-specific abstractions for validating generic domain worlds while consultants with a wide range of experience with different applications are more likely to possess mental schemata equivalent to higher-level domain abstractions. A host of knowledge acquisition techniques (e.g. Littman 1986, Cooke & McDonald 1987, Garg-Janardan & Salvendy 1987) exist to elicit experts' mental schemata. An interesting comparison would be to determine whether the schemata possessed by experts differ from those abstractions derived from organisation-wide domain analyses. Finally, observing schema acquisition by software engineers during analogical abstraction can also indicate the nature of these mental schemata, although eliciting this process is, to say the least, problematic. Even the most complete concurrent protocols during studies 3 and 4 failed to capture the schema acquisition process, indicating that such learning is unconscious and can, at best, only be partially captured during retrospective questioning.

The remainder of the thesis examines the advantages of domain abstractions in software engineering, then further research of analogical specification reuse is discussed in light of current limitations.

7.3 Domain Abstraction

Domain abstraction is central to the proposed logical model of software engineering analogies. Analogy emphasises reuse of coherent knowledge structures linking domain objects, so it is a more effective paradigm for reuse of large, complex domains which minimise the importance of component functionality. On the other hand, it appears to be less-effective for reuse of computationally-intensive systems such as accounting or statistics. Previous domain analyses of complex real-time applications such as air traffic control and traffic light control systems have led to the derivation of many low-level concepts (e.g. *vehicle*) supporting functional-level reuse rather than higher-level analogical mapping (Johnson 1991, Johnson & Harris 1991). Thus, analogy may be less effective for reuse of real-time systems, although further example-based and empirical research is required to support this conclusion.

Analogy may be the best research direction for defining the most effective coverage and granularity of domain abstractions. Coverage is linked to the need to validate the model of software engineering domains in many realistic scenarios. Extensive coverage establishes confidence in the model, although it should be emphasised that complete coverage of domains cannot be asserted theoretically. Granularity concerns the scale and size of domain abstractions for effective specification reuse. Domain abstractions can be aggregated or specialised to determine this most effective level of granularity. Analogy provides a novel and alternative approach to determining the coverage and granularity of software engineering abstractions.

Domain abstractions can assist software engineering practice in many ways, for instance a strong theoretical model of domain abstraction in software engineering can support business and domain modelling (e.g. Prieto-Diaz 1990, Greenspan et al. 1991) by providing a framework for defining domains. Analogically-matched domain abstractions can assist validation of requirements specifications (e.g. Reubenstein & Waters 1991) and other software engineering tasks including reverse engineering and view integration. Assistance for these two tasks is examined more closely.

7.3.1 Reverse System Engineering

Current reverse-engineering strategies have been limited by the availability of domain and design knowledge (e.g. Byrne 1991), however one potential solution may be to interpret system code and low-level designs using domain abstractions. Although domain abstractions are too broad to assist low-level reverse-engineering to designs, they can assist the derivation of system specifications from these designs by structuring low-level designs within functional requirements. Furthermore, incorporating design abstractions into models of domain abstraction can assist this reverse-engineering activity by linking generic designs to domains and providing 'hooks' between system specification and implementation.

7.3.2 Viewpoint Integration

Domain abstractions can be specialised to incorporate different views of instantiated concrete domains. Differing viewpoints and the need for contextual interpretation of complex requirements have been identified as critical to supporting software engineers (e.g. Leite 1989, Easterbrook 1991). Identifying and modelling these viewpoints can allow intelligent environments to facilitate and assist viewpoint recognition through context identification and reconciliation. For instance, the domain abstraction underlying the theatre reservation example may be viewed differently by end-users and software developers, as shown in Figure 7.1. The theatre manager is primarily interested in overall patterns of auditorium use, the sales assistant at the ticket kiosk requires a two-dimensional theatre model for identifying available seats and the application programmer reasons about specific allocation algorithms. It is hypothesised that a common set of domain viewpoints can be identified for each domain class, and that these viewpoints can greatly assist the identification and reconciliation of individual differences in domain understanding.

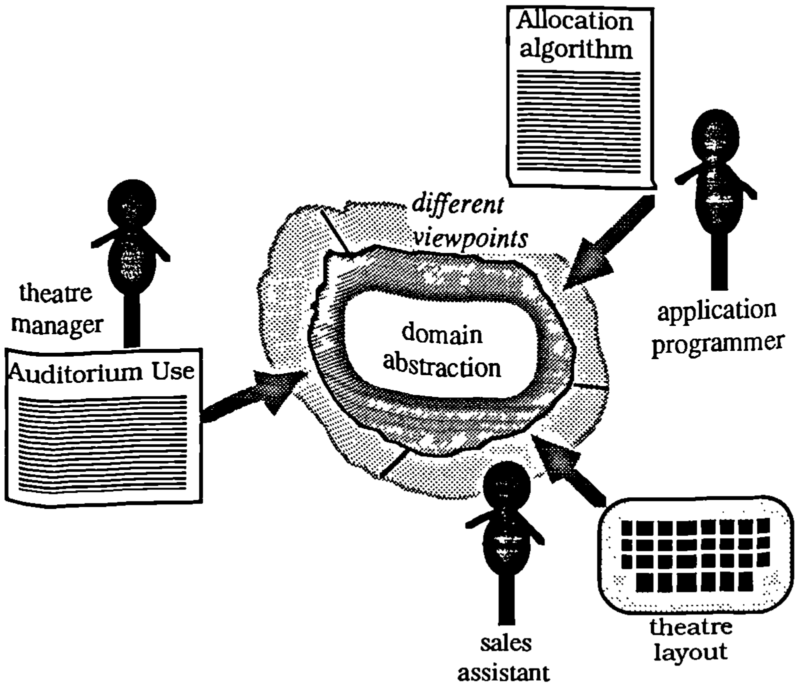


Figure 7.1 - simple example demonstrating the integration of viewpoints through single domain abstractions

7.3.3 Domain Abstractions: A Summary

To sum, domain abstractions are an effective means of capturing, storing and reusing domain knowledge during requirements engineering. Domain abstractions can also permit the classification of software engineering domains to support other activities including reverse engineering, viewpoint reconciliation and domain modelling. Domain abstractions best assist these activities if supported by effective reasoning mechanisms such as analogical matching. Combining this paradigm with more complex hybrid reasoning mechanisms such as the Requirement's Apprentice's layered reasoning facilities (Reubenstein & Waters 1991) is another direction for future research. The use of knowledge acquisition techniques for eliciting and modelling generic domain knowledge, similar to the approach adopted by the KADS project (Wielinga et al. 1991), also warrants investigation. Finally, this research has modelled domain knowledge rather than problems which trigger requirements, however problems causing requirements are complex and vary. They include social or organisational issues, indicating the need for more complex domain and organisational models to be effective (e.g. Goguen et al. 1991). Models should represent the causality underlying requirements (e.g. Yu 1991). Thus, a third area for future research is to integrate domain abstractions with organisational models representing personal conflicts, power balances, communication channels and stakeholders.

7.4 Future Work

Although this research has provided some useful results the scope, validity and completeness of the paradigm remain to be evaluated in future work.

7.4.1 Further Implementation of the Advisor

Further implementation of the intelligent reuse advisor is envisaged. In the first stage, the specification advisor will be implemented as specified in chapter 5 to evaluate the analogical comprehension and transfer strategies derived empirically in chapter 4. However, effective and large-scale analogical reuse is likely to require additional functionality to assist selection between candidate analogical specifications prior to customisation. The second stage of this evaluation will be in two parts. First, the specification advisor can be tested in isolation to determine its effectiveness on analytic performance, measured by the improved completeness and validity of specifications and understanding of key analogical determinants using measures similar to those reported in chapter 4. In addition, retrospective questioning will determine the effectiveness of the strategies for analogical comprehension and transfer. Second, all three components of the reuse advisor will be evaluated together as a coherent toolkit, thus permitting evaluation of a complete phase of analogical specification reuse from domain description to specification customisation.

7.4.2 Aggregation of Domain Abstractions

Solving software engineering problems is complex and often involves domains which, when instantiated, are larger than individual domain abstractions. Unfortunately, the proposed single hierarchical structure may not always represent the true complexity and scale of many software engineering domains, so multi-dimensional storage and access to domain abstractions may be needed. Riesbeck & Schank (1989) hypothesised that people also need more complex knowledge structures when reasoning about complex situations, for instance 'Going on a Vacation' and 'Going on a Business Trip' are specialisations of 'Going on a Trip' events (p. 34). Both events involve the 'Getting a Ticket' scene, however 'Getting a Ticket' may be scenes in many other events such as 'Going to the Cinema'. This example suggests that class hierarchies may not represent all software engineering domains. In particular, domain aggregations can comprise several domain abstractions, and each domain abstraction can occur in several aggregations. The *stock-lending* domain aggregation underlying the video hiring store shown in Figure 7.2 can be defined by at least two domain abstractions. The book lending activities in the library instantiate the non-renewable resource management abstraction while key facts about the inventory control activities are represented in the renewable resource management abstraction. These two domain abstractions can be aggregated by common occurrence since they also can be instantiated in other analogical domains such video lending, tool hire and car rental domains. Other aggregations are shown in Appendix I. As such, multi-dimensional storage and access to domain abstractions can supplement the hierarchical model of domain abstraction for large-scale specification reuse.

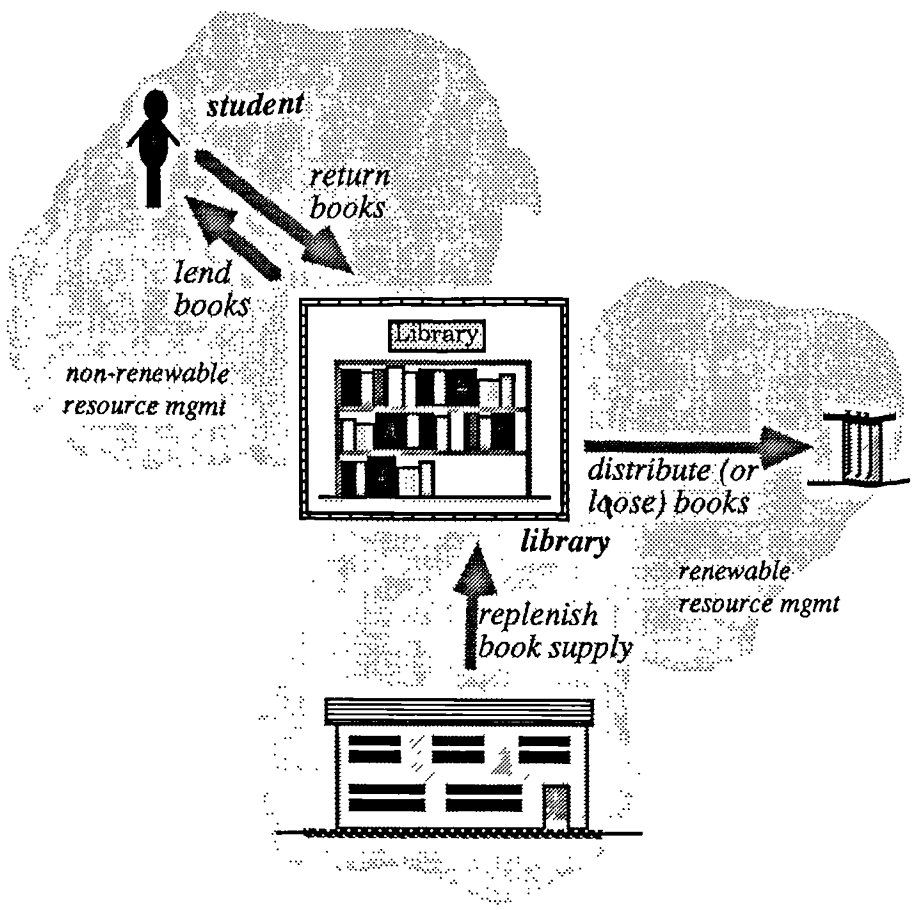


Figure 7.2 - lending library domain

7.4.3 Design Abstractions & Non-Functional Requirements

The proposed model of domain abstraction can also be extended to represent design abstractions in the design space as well as domain abstractions in the domain space. The current model of software engineering analogies does not match design components although design matching can extend analogical reuse towards the design space (see Figure 7.3). This endeavour is tractable because mapping at the design level of abstraction has been proposed in transformational programming (e.g. Barstow 1985, Feather 1987, Doerry et al. 1990, Fickas & Helm 1990, Fickas & Helm 1991, Chung et al. 1990) and reuse (e.g. Harandi & Lubars 1987), so the model will add an intermediate level of abstraction linking designs to requirements. Designs represent algorithms which differ from transformational programming involving the automatic change of formal program structures. For example, several algorithms exist for sorting and matching resources to requirements in object allocation domains. The problem of determining a complete and valid set of design abstractions is analogous to that for domain abstractions. Therefore similar research paradigms are envisaged to populate the design space with generic designs.

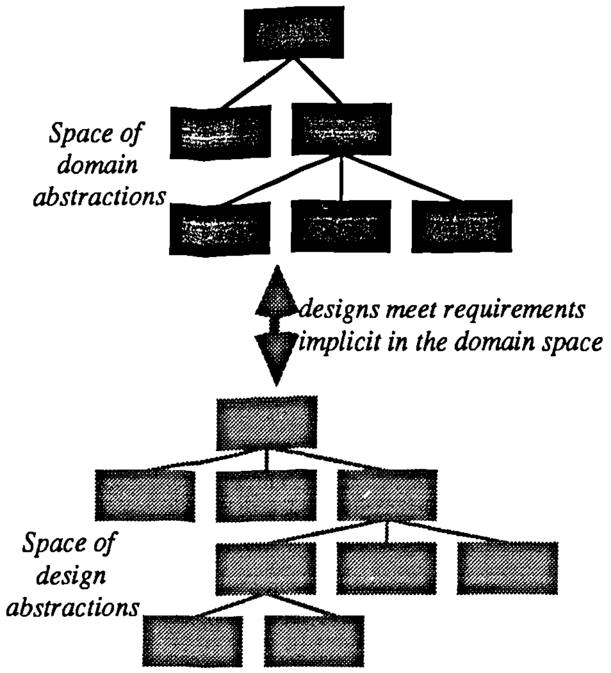

Space of domain abstractions
designs meet requirements implicit in the domain space
Space of design abstractions

Figure 7.3 - domain and design spaces

Non-functional requirements associated with software designs and algorithms such as adaptability, maintainability and complexity may be able to assist selection of reusable components using taxonomies of requirements based on existing definitions and metrics. Unfortunately current research of non-functional requirements is immature, and a better understanding of non-functional requirements is needed before this extension to the framework can be achieved. As a result, component retrieval and selection by functional rather than non-functional requirements is more likely to provide benefits in the near future.

7.4.4 Intelligent Requirements Engineering

The logical model of domain abstraction and the advisor's fact acquisition strategies have important implications for intelligent requirements engineering. First, they provide a theory of domain knowledge to inform design of support tools. Fact acquisition dialogues incorporated into the advisor can assist the structuring, scoping and classification of new domains. Analogical examples of candidate domains can also be used to confirm and guide analytic behaviour in terms of more well understood concepts (e.g. Fischer et al. 1991b). The role of concrete examples in component retrieval and problem formulation is not new, for instance both the Programmer's and Requirement's Apprentices (Rich & Waters 1988, Reubenstein & Waters 1991) provided concrete cliches as both reusable solutions and bases for common communication between tool and machine. Second, terms and semantics defining domain abstractions have implications for knowledge representation languages. The Requirements Apprentice (Reubenstein & Waters 1991) recalled and reasoned with domain knowledge in the form of cliches to identify inconsistencies, omissions and ambiguities in requirements specifications. However, these cliches appeared to be developed in an ad hoc fashion without any underlying theory of domain knowledge. Similarly, languages developed for requirements modelling (e.g. Greenspan 1984, Borgida et al 1985, Greenspan et al. 1986, Lubars 1988, Mylopoulos et al. 1990) have no underlying theory of domain knowledge, although effective requirements capture needs such a theory. The framework of software engineering analogies constrains knowledge representations to those key fact types describing the essence of software engineering domains, around which more elaborate and richer knowledge representations can be developed. Thus, the framework has implications for representing domain knowledge in requirements engineering.

7.5 Questions Still to be Answered

This research claims to have proposed and answered important questions about analogical specification reuse during requirements engineering. However, it has also provoked some wide-ranging questions which are beyond its scope. Object-oriented analysis has been proposed as an effective means of requirements modelling (e.g. Shlaer & Mellor 1988, Coad & Yourdon 1990), although derivation of objects during requirements engineering appears anecdotal and limited to identification of *things* in a domain. Domain abstractions may provide a theoretical and logical basis for the identification of objects and their relationships during requirements engineering.

This research has modelled domain knowledge in requirements engineering rather than the requirements engineering process (Curtis et al. 1990, Carr & Koestler 1990), however integrating domain and process knowledge during requirements engineering is likely to provide the most effective support during requirements engineering (Nature 1992). Furthermore, a better understanding of the requirements engineering process at the macro and micro levels, supported by cognitive models of individual software engineering tasks, is necessary to support requirements engineering. To this end, Pennington & Grabowski (1990) categorised software engineering processes into types. Process models can be developed at two levels. A theory of process engineering based on models of software development activities at macro and micro levels of detail can demonstrate the contextual links between activity organisation, the nature of individual activities and their triggering conditions. At the detailed level, process modelling will investigate specific activities (e.g. requirements validation, view integration) and the role of domain knowledge in requirements definition. These detailed processes may be domain-specific, thus identifying a link between candidate domain abstractions and requirements engineering process models for larger-scale procedural support. Thus, providing the most effective tool support for requirements engineering needs knowledge of both the process and the domain. The domain theory outlined in this thesis will be developed into a wider theory of requirements engineering to inform design of truly effective support tools.

Glossary

The following glossary defines key terms and concepts presented in this thesis.

Analogy Engine

One of Ira's three main components. The analogy engine is a computational implementation of the logical model of software engineering analogies. This implementation is a hybrid, combining both analogical and heuristic reasoning algorithms for the retrieval and explanation of specifications.

Analogical Match

Degree of similarity and abstract relationship between a pair of domain descriptions, either between two concrete domains or between a concrete and abstract domain description.

Analogical Recognition

Identification of one or several key similarities between a target domain description and descriptions of either a source domain or domain abstraction.

Analogical Comprehension

A mental state possessed by the software engineer represented as analogical mappings between a source and target domain and a structural isomorphism between mapped objects in both domains.

Analogical Specification

A reusable specification which can be matched analogically to the target domain. Specifications are represented using structured notations such as entity-relationship diagrams, data flow diagrams or structure charts.

Analogical Transfer

Customisation of an analogical specification by a software engineer to fit the target domain. This transfer is achieved using analogical mappings between the domains which represent the software engineer's analogical comprehension.

Assertion (Mental Behaviour Category)

Verbalisation of a belief or statement of facts about the target or source domains directly attributable to a problem narrative or reusable specification.

CASE (Computer-Aided Software Engineering) Tool

Tool-based software development environment containing a repository of specifications which can be reused analogically.

Cognitive Distance

The degree of semantic and syntactic dissimilarity between the software engineer's understanding of two analogical domains.

Cognitive Models of Analogical Reuse

Overall representation of the mental processes and models which occur during analogical specification reuse. Mental processes are defined using cognitive task and reasoning models while mental models of analogical understanding specify the extent and structure of analogical mappings between objects in the target, source and abstract domains. The cognitive model of analogical reuse consists of the cognitive task model, cognitive reasoning model and mental model of analogical understanding.

Cognitive Reasoning Model

Representation of the mental processes and hypotheses of analogical reasoning during specification reuse. This model represents software engineering behaviour at a finer level of detail than the cognitive task model. Reasoning is represented as transitions between hypothesis states, such as develop, test and discard. The cognitive reasoning model is similar to GOMS (Card et al. 1983) in its identification of goals and operators.

Cognitive Task Model

Representation of a sequence of steps and components in mental tasks during analogical specification reuse, examples of which include strategies such as information gathering and specification reuse. Reasoning about topics within each of these tasks is represented using the cognitive reasoning model.

Construct Specification (Analytic Strategy)

Develop a structured diagram representing a specification without reusing an analogical specification.

Cooperative Specification Reuse

Paradigm for specification reuse in which the division of work is shared between the software engineer and the tool. This thesis proposes that analogical retrieval is requires extensive intervention by the advisor. On the other hand analogical comprehension and transfer are tasks undertaken primarily by the software engineer, with support from the intelligent advisor.

Coverage Problem

The need to validate the defined domain abstractions by their coverage of software engineering domains in realistic scenarios. Extensive coverage is needed to establish confidence in the model, although it should be emphasised that complete coverage of domains cannot be asserted.

Design Space

Space of all known design abstractions which may be matched.

Diagram-based Testing (Mental Category)

A mental behaviour exhibited by software engineers during empirical studies when generating multiple tests to evaluate an existing structured diagram. The thematic sequence of these tests is guided by the structure of the diagram.

Domain Abstractions

A logically-defined generic representation of a software engineering domain, composed of abstract facts defined in the meta-schema of knowledge types. Domain abstractions are key to analogical matching between source and target domains. They are derived by example-based analysis of software engineering analogies to identify key domain facts belonging to all instances of each analogy. Individual domain abstractions belong to the logical model of domain abstraction.

Domain Description

A description of the components of a concrete target or source domain using a restricted set of terms.

Domain Requirements

A type of knowledge defined in the meta-schema representing a broad category of system requirements including functional needs, desired domain states and constraints on the functionality and implementation of the information system. Requirements in the meta-schema are expressed using natural language statements.

Domain Space

Space of all known domain abstractions which may be matched. It is the same as the logical model of domain abstraction.

Domain Terms

A set of terms for defining and distinguishing domain abstractions and their instances within the meta-schema of knowledge types. These terms also define all domain instances belonging to the defined set of domain abstractions.

Evaluate Against the Analogy (Analytic Strategy)

Test an existing specification by comparing it with a reusable specification.

Evaluate Against the Target (Analytic Strategy)

Test an existing specification against the original needs statement for that specification.

Framework of Software Engineering Analogies

A theoretically-derived definition of an analogical match between software engineering domains. Central to this framework are a meta-schema of knowledge types representing critical analogical determinants for retrieval of specifications and a logical model of domain abstractions representing critical features of known classes of domain. In addition, structure matching algorithms determine the existence of a structural match between software engineering domains, and heuristics identifying key differences between domain abstractions to aid selection of the best-fitting domain abstraction for a concrete domain.

Functional Requirement

A desired characteristic of some component independent of its implementation. Functional requirements are of three types. Problem-driven requirements are caused by failure of the old system which must be corrected in the new system. Goal-driven requirements determine new system features to be added in the new implementation. Finally constraints describe events, functions or states which must never occur in the new implementation.

Gather information (Analytic Strategy)

Assimilate source or target domain knowledge by reading either the target document or the reusable specification.

Generic Domain World

A specialisation of a domain abstraction, representing common physical attributes of a significant, predetermined set of domain instantiations. All instances of a generic domain world are also specialisations of the same, higher-level domain abstraction. Typically, generic domain worlds represent domains at a level of abstraction equivalent to application templates (e.g. Fugini et al. 1991).

Granularity Problem

The need to determine the most effective scale and size of domain abstractions for supporting analogical specification reuse and other requirements engineering activities. Domain abstractions can be aggregated or specialised to determine the most effective level of abstraction.

Intelligent Advisor

A support tool which uses partial knowledge of the user and domain to cooperate with the user during the solution of a complex problem. Intelligent advisors differ from intelligent tutors by their knowledge of the domain and nature of interaction with the user.

Ira (Intelligent Reuse Advisor)

An intelligent advisor designed to cooperate with the software engineer during analogical specification retrieval, comprehension and transfer.

Intelligent Tutoring System (ITS)

Intelligent tutoring systems are one paradigm for computer-assisted instruction. Intelligent tutors have intelligence in three areas, namely knowledge of the domain of expertise, ability to deduce a learner's approximation of that knowledge and the ability to implement pedagogical strategies which reduce the difference between expert and student performance.

Key Object Mappings

A set of analogical mappings between source and target domain objects which instantiate the shared domain abstraction.

Logical Model of Domain Abstraction

A theoretically-derived definition of the structure and space of all domain abstractions. Each individual abstraction is defined and distinguished from other abstractions using key facts representing using the meta-schema of knowledge types.

Logical Model of Software Engineering Analogies

A definition of similarity between software engineering domains, consisting of two components. The meta-schema of knowledge types represents critical determinants of analogies and the logical model of domain abstraction represents critical features of known classes of domain.

Memory Schema

A cognitive representation possessed by a software engineer of key abstractions about a software engineering domain. Memory schema are induced through exposure and experience of reasoning about concrete domain instances. They exist in either working or long term memory. They represent the mental equivalent of the logically defined domain abstractions.

Mental Model of Analogical Understanding

A mental state possessed by a software engineer which represents the software engineer's understanding of the analogical match, represented as analogical mappings between semantic objects in the source and target domains, structural isomorphism between these mapped objects and the abstraction representing key constructs of both domains. These mental models are closer to Gentner's (1983) use of the term rather than that used by Johnson-Laird (1983).

Meta-schema of Knowledge Types

A description of seven knowledge types which represent key domain facts for the retrieval and explanation of analogical specifications. This meta-schema can be instantiated to represent both concrete software engineering domains and their domain abstractions. The seven knowledge types defined in the meta schema are key state transitions, object structures, domain requirements, preconditions on state transitions, object types, system functionality and domain events, and functions achieving state transitions.

Model-based Reasoning (Mental Category)

Model-based reasoning involves the generation and development of reasoning hypotheses linked by a single thematic strand related to components in the structured diagram.

Model Recording (Non-Mental Category)

A physical behaviour exhibited by software engineers during empirical studies in chapter 4 when representing the problem domain or a tentative solution using structured analytic notations such as entity-relationship, data flow and process structure diagrams.

Needs Statement

An informal document describing existing problems and functional and non-functional requirements of a new system. This document is often incomplete, inconsistent and ambiguous.

Object Structures (Type of Knowledge in the Meta-Schema)

A type of knowledge defined in the meta-schema.

Object Types (A Type of Knowledge in the Meta-Schema)

A type of knowledge defined in the meta-schema. Objects may be classified by their role during key state transitions.

Planning (Mental Category)

Meta-level control over the analytic process. Two types of plan are distinguished by content of method knowledge and SSA heuristics, or general heuristics.

Preconditions on State Transitions (Type of Knowledge in the Meta-Schema)

A type of knowledge defined in the meta-schema representing triggers for key state transitions. These triggers are necessary for the state transition.

Problem Identifier

One of Ira's three main components. The problem identifier acquires key facts from the software engineer prior to specification retrieval and explains domain abstractions to the software engineer for confirmation or rejection of the analogical match.

Reasoning (Mental Category)

Category inferred from verbalisation of the creation, development and testing of hypotheses. Reasoning utterances were further categorised to identify subjects' topic focus: (i) reasoning about the target domain, (ii) reasoning about the source domain, (iii) reasoning about analogical concepts between the source and target domains, and (iv) reasoning about general concepts which do not describe the target or source domains, or the analogical links between them.

Requirements Engineering

The first activity in the development of a software system, during which functional and non-functional requirements are captured, modelled and validated during an interactive process between software engineers and end users of the required system.

Reusable Specification

Functional definition of an existing software system held in a CASE tool repository and available for analogical reuse. Reusable specifications are typically represented using notations belonging to structured methods such as data flow and entity-relationship diagrams supported by narrative documents.

Reuse Specification (Analytic Strategy)

Reuse the FMS specification to develop a structured diagram representing the specification of a solution to the target problem.

Revise Specification (Analytic Strategy)

Redraw a structured diagram from a first-draft, less well-defined structured diagram or informal sketch.

Simple Retrieval Mechanism

A simple, keyword-based mechanism for the retrieval of domain abstractions early in a fact acquisition dialogue. Retrieval is based on matched system functionality and domain events as defined in the meta-schema of knowledge types.

Software Engineering Environment

An integrated toolkit which supports the software engineer during all phases of software development.

Source Domain

The reusable domain represented by any candidate reusable specification.

Specification Advisor

One of Ira's three major components. The specification advisor acts as an intelligent assistant during analogical comprehension and transfer of a retrieved specification. It implements strategies and explanations for effective reuse, supported by diagnoses of analogical errors and misconceptions exhibited by software engineers.

State Transitions (Type of Knowledge in the Meta-Schema)

A type of knowledge defined in the meta-schema. State transitions occur in respect to an object structure in the domain.

Structured Diagrams

Notations for representing software engineering specifications and designs. Structured diagrams can represent both target domains or analogical specifications. Examples of such notations include data flow, entity-relationship or entity life history diagrams.

Summarise Data (Analytic Strategy)

Summarise the contents of the target document or the reusable specification.

System Functionality & Domain Events (Types of Knowledge in the Meta-Schema)

Types of knowledge defined in the meta-schema. System functions and domain events elaborate the definition of key state transitions by their association with these state transitions.

Target Domain

The new domain in which requirements engineering is taking place and to which analogical specifications must be matched.

Bibliography & References

References

Adelson B., 1984, 'When Novices Surpass Experts: the Difficulty of a Task may Increase with Expertise', *Journal of Experimental Psychology: Learning, Memory and Cognition* **10(3)**, 483-495.

Adelson B. & Soloway E., 1985, 'The Role of Domain Experience in software Design', *IEEE Transactions on Software Engineering*, **SE-11(11)**, 1351-1360.

Agard, 1973, 'AGARD Conference Proceedings 105 on Air Traffic Control Systems'.

Akin O., 1986, 'Psychology of Architectural Design', Pion Ltd.

Amadeus, 1986, 'AMADEUS Project: A Report on Task B1: A Report Classifying System Types', internal report.

Anderson J.R., 1990, 'The Adaptive Character of Thought', Hillsdale NJ, Erlbaum.

Anderson J.R., 1988, 'The Expert Module', *Foundations of Intelligent Tutoring Systems*, ed. M.C, Polson & J.J. Richardson, Lawrence Erlbaum Associates.

Anderson J.R., Franklin Boyle C., Corbett A.T. & Lewis M.W., 1990, 'Cognitive Modeling and Intelligent Tutoring', *Artificial Intelligence* **42**, 7-49.

Anderson J.R., Franklin-Boyle C., Farrell R. & Reiser B.J., 1987, 'Cognitive Principles in the Design of Computer Tutors', Modelling Cognition, ed. P. Morris, John Wiley & Sons, 93-133.

Arango G., 1988, 'Domain Engineering for Mechanical Reuse', internal document, Department of Information and Computer Science, University of California, Irvine.

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

Arango G., Bruneau L., Cloarec J.F. & Feroldi A., 1991, 'A Tool Shell for Tracking Design Decisions', *IEEE Software*, March 1991, 75-83.

Arango G. & Freeman P., 1985, 'Modelling Knowledge for Software Development', Proceedings of 3rd Intl Workshop on Software Specification and Design, London, IEEE Computer Society Press, 63-66.

Ashley K.D., 1991, 'Reasoning with Cases and Hypotheticals in Hypo', *International Journal of Man-Machine Studies* 34, 753-796.

Ashley K.D. & Rissland E.L., 1988, 'A Case-based Approach to Modelling Legal Expertise', *IEEE Expert*, Autumn 1988, 70-77.

Balzer R., 1981, 'Transformational Implementation: An Example', *IEEE Transactions on Software Engineering* **SE-7(1)**, 3-14.

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

Barstow D.R., 1985, 'Domain-Specific Automated Programming', *IEEE Transactions on Software Engineering* **SE-11(11)**, 1321-1336.

Basili V.R., 1990, 'Viewing Maintenance as Reuse-Oriented Software Development', *IEEE Software*, January 1990, 19-25.

Basili V.R. & Musa J.D., 1991, 'The Future Engineering of Software: A Management Perspective', *IEEE Computer*, September 1991, 90-96.

Biggerstaff T.J., 1989, 'Design Recovery for Maintenance and Reuse', *IEEE Computer*, July 1989, 36-49.

Biggerstaff T.J., 1987, 'Hypermedia as a Tool to Aid Large Scale Reuse', MCC Technical Report STP-202-87, Software Technology Program, July 1987, MCC, Austin TX.

Biggerstaff T.J. & Richter C., 1987, 'Reusability Framework, Assessment, and Directions', *IEEE Software*, March 1987, 41-49.

Boehm B.W., 1981, 'Software Engineering Economics', Prentice-Hall Inc., Englewood Cliffs, N.J..

Boldyreff C., 1989, 'Reuse, Software Concepts, Descriptive Methods and the Practitioner Project', *ACM SIGSOFT Software Engineering Notes* **14(2)**, 25-31.

Borgida A., Greenspan S. & Mylopoulos J., 1985, 'Knowledge Representation as the Basis for Requirements Specifications', *IEEE Computer*, April 1985, 82-90.

Bott M.F. & Wallis P.J.L., 1988, 'Ada and Software Reuse', *Software Engineering Journal* 3(3), 177-183.

Branting L.K., 1991, 'Building Explanations from Rules and Structured Cases', *International Journal of Man-Machine Studies* **34**, 797-837.

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.

Burstein M.H., 1988a, 'Incremental Learning from Multiple Analogies', *Analogica*, Pitman (London), 37-61.

Burstein M.H., 1988b, 'Combining Analogies in Mental Models', *Analogical Reasoning*, ed. Helman D.H., Kluwer Academic Publishers, 179-203.

Burton B.A., Aragon R.W., Bailey S.A., Koehler K.D. & Mayers L.A., 1987, 'The Reusable Software Library', *IEEE Software*, July 1987, 25-32.

Byrne E.J., 1991, 'Software Reverse Engineering: A Case Study', *Software - Practice and Experience* **21(12)**, 1349-1364.

Caplan L.J. & Schooler C., 1990, 'Problem Solving by Reference to Rules or Previous Episodes: The Effects of Organised Training, Analogical Models, and Subsequent Complexity and Experience', *Memory and Cognition* **18(2)**, 215-227.

Carbonell J.C., 1985, 'Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition', Technical Report CMU-CS-85-115, Computer Science Department, Carnegie-Mellon University, March 1985.

Carbonell J.C., 1988, 'Experimental Learning in Analogical Problem Solving', internal document, Carnegie-Mellon University.

Carr D. & Koestler R., 1990, 'System Dynamics Models of Software Developments', Proceedings 5th International Software Process Workshop (Experiences with Software Process Models), Kennebunkport ME, 10-13 October 1989, IEEE Computer Society Press, 46-48.

Carroll J.M., Smith-Kerker P.L., Ford J.L. & Mazur-Rimetz S.A., 1988, 'The Minimal Manual', *Human-Computer Interaction* **3**, 123-153.

Cheng P.W. & Holyoak K.J., 1985, 'Pragmatic Reasoning Schemas', *Cognitive Psychology* **17**, 391-416.

Cheng P.W., Holyoak K.J., Nisbett R.E. & Oliver L.M., 1986, 'Pragmatic vs Syntactic Approaches to Training Deductive Reasoning', *Cognitive Psychology* **18**, 293-328.

Chenoweth S.V., 1991, 'On the NP-Hardness of Blocks World', Proceedings of AAAI'91, AAAI Press/MIT Press, 623-628.

Chi M.T.H., Bassok M., Lewis M.W., Reimann P. & Glaser R., 1989, 'Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems', Cognitive Science 13, 145-182.

Chi M.T.H., Glaser R. & Rees E., 1982, 'Expertise in Problem Solving', *Advances in the Psychology of Human Intelligence*, ed. R. Sternberg, Lawrence Erlbaum Associates, 7-75.

Chung L., Katalagarianos P., Marakakis M., Mertikas M., Mylopoulos J. & Vassiliou Y., 1990, 'From Information System Requirements to Designs: A Mapping Framework', Technical Report CSRI-245, Computer Systems Research Institute, University of Toronto, September 1990.

Coad P. & Yourdon E., 1990, 'Object-Oriented Analysis', Prentice-Hall.

Conklin J., 1987, 'Hypertext: An Introduction and Survey', *IEEE Computer* **20(9)**, 17-41.

Cooke N.M. & McDonald J.E., 1987, 'The Application of Psychological Scaling Techniques to Knowledge Elicitation for Knowledge-Based Systems', *International Journal of Man-Machine Studies* **26**, 533-550.

Crinnion J., 1991, 'Evolutionary Systems Development: a Practical Guide to the Use of Prototyping within a Structured Systems Methodology', Pitman.

Cumming G. & Self J., 1989, 'Learner Modelling in Collaborative Intelligent Educational Systems', *Teaching Knowledge and Intelligent Tutoring*, ed. P. Goodyear, Norwood N.J., Ablex.

Curry G.A. & Ayers R.M., 1984, 'Experience with Traits in the Xerox Star Workstation', *IEEE Transactions on Software Engineering* **SE-10(5)**, 519-527.

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.

Curtis B. & Walz D., 1990, 'The Psychology of Programming in the Large: Team and Organisational Behaviour', *Psychology of Programming*, ed. J.M. Hoc, T. Green, R. Samurcay & D. Gilmore, Academic Press.

Curtis B., Walz D. & Elam J., 1990, 'Studying the Process of Software Design Teams', Proceedings 5th International Software Process Workshop (Experiences with Software Process Models), Kennebunkport ME, 10-13 October 1989, IEEE Computer Society Press, 52-53.

Cutts G., 1987, 'SSADM - Structured Systems Analysis and Design Methodology', Paradigm Publishing.

Czuchry A.J. & Harris D.R., 1988, 'KBRA: A New Paradigm for Requirements Engineering', *IEEE Expert*, Winter 1988, 21-35.

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.

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

Derry S.J., Hawkes L.W. & Ziegler U., 1988, 'A Plan-Based Opportunistic Architecture for Intelligent Tutoring', Proceedings of ITS-88, June 1-3 1988, Montreal, Canada, 116-123.

Dhar V. & Jarke M., 1988, 'Dependency-Directed Reasoning and Learning in Systems Maintenance Support', *IEEE Transactions on Software Engineering* **14(2)**, 211-227.

Doerry E., Fickas S., Helm R. & Feather M., 1991, 'A Model for Composite System Design', Proceedings 6th International Workshop on Software Specification and Design, Como (It), IEEE Computer Society Press, 216-219.

Dubisy F. & van Lamsweerde A., 1990, 'Requirements Acquisition by Analogy', Technical Report No. 13, Institut d'Informatique, Facultes Universitaires de Namur (Belgium), June 1990.

Dusink L. & Hall P., 1991, 'Software Reuse, Utrecht 1989', Workshops in Computing Series, Springer-Verlag.

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.

Easterbrook S., 1991, 'Handling Conflict between Domain Descriptions with Computer-Supported Negotiation, *Knowledge Acquisition* **3**, 255-289.

Elsom-Cook M., 1988, 'Guided Discovery Tutoring and Bounded User Modeling', *Artificial Intelligence and Human Learning (Intelligent Computer-Aided Instruction)*, ed. J.A. Self, Chapman & Hall Computing, 165-196.

Elzer P.F., 1991, 'Reuse, a Problem of "Understanding" Designs', Proceedings of 1st Intl Workshop on Software Reusability, Dortmund July 3-5 1991, 12-17.

Ericsson K.A. & Simon H.A., 1980, 'Verbal Reports as Data', *Psychological Review* **87**, May 1980, 215-251.

Ericsson K.A. & Simon H.A., 1984, 'Protocol Analysis', MIT Press.

Escott J.A. & McCalla G.I., 1988, 'Problem Solving by Analogy: A Source of Errors in

Novice LISP Programming', Proceedings of ITS-88, June 1-3 1988, Montreal, Canada, 312-319.

Falkenhainer B., Forbus K.D. & Gentner D., 1989, 'The Structure-Mapping Engine: Algorithm and Examples', *Artificial Intelligence* **41**, 1-63.

Feather M.S., 1987, 'A Survey and Classification of some Program Transformation Approaches and Techniques', *Program Specification and Transformation*, ed. L.G.L.T. Meertens, Elsevier Science Publishers.

Feather M.S., Fickas S. & Helm R., 1991, 'Composite System Design: the Good News and the Bad News', Proceedings of 6th Knowledge-Based Software Engineering Conference, Syracuse NY, IEEE Computer Society Press, 16-25.

Fickas S., 1987, 'Automating the Specification Process', Technical Report CIS-TR-87-05, Department of Computer and Information Science, University of Oregon.

Fickas S., Collins S. & Oliver S., 1988, 'Problem Acquisition in Software Analysis: A Preliminary Study', Report CIS-TR-87-04, Department of Computer and Information Science, University of Oregon.

Fickas S. & Helm R., 1991, 'Knowledge Representation and Reasoning for the Design of Composite Systems', internal report, Department of Computer Science, University of Oregon, December 1991.

Fickas S. & Helm R., 1990, 'A Transformational Approach to Composite System Specification', Technical Report CIS-TR-90-19, Department of Computer and Information Science, University of Oregon, November 1990.

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

Finkelstein A., 1988, 'Reuse of Formatted Requirements Specifications', *Software Engineering Journal* 3(3), 186-197.

Fischer G., 1987, 'Cognitive View of Reuse and Redesign', *IEEE Software*, July 1987, 60-72.

Fischer G., Henninger S. & Redmiles D., 1991a, 'Cognitive Tools for Locating and Comprehending Software Objects for Reuse', Proceedings of 13th International Conference on Software Engineering at Austin Texas, May 1991.

Fischer G., Henninger S. & Redmiles D., 1991b, 'Intertwining Query Construction and Relevance Evaluation, Proceedings of CHI'91, ed. S.P. Robertson, G.M. Olson & J.S. Olson, ACM Press, 55-62.

Fischer G., Lemke A.C., McCall R. & Morch A.I., 1991c, 'Making Argumentation Serve Design', to appear in Human-Computer Interaction.

Fischer G., McCall R. & Morch A.I., 1989, 'Design Environments for Constructive and Argumentative Design', Proceedings of CHI'89 Conference, ed. K. Bice & C. Lewis, ACM Press, 269-275.

Fischer G. & Nakakoji K., 1991, 'Making Design Objects Relevant to the Task at Hand', Proceedings of AAAI'91, AAAI Presss/MIT Press, 67-73.

Fischer G. & Reeves B., 1991, 'Exploring and Analysing Success Models of Cooperative Problem Solving', Technical Report CU-CS-510-910, Department of Computer Science, University of Colorado at Boulder, January 1991.

Flynn D.J., Layzell P.J. & Loucopoulos P., 1986, 'Assisting the Analyst: Aims and Approaches of the Analyst Assist Project', Software Engineering-86, ed. D. Barnes & P. Brown., Peter Peregrinus/BCS, 19-26.

Frenkel K.A., 1985, 'Toward Automated the Software Development Cycle', *Communications of the ACM* **28(6)**, 578-589.

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), May 1991, ed. R. Anderson, J.A. Bubenko & A. Solvberg, Lecture Notes in Computer Science 498, Springer-Verlag, 50-78.

Furnas G.W., Landauer T.K., Gomez L.M. & Dumais S.T., 1987, 'The Vocabulary Problem in Human-System Communication', *Communcations of the ACM* **30(11)**, 964-971.

Garg-Janardan C. & Salvendy G., 1987, 'A Conceptual Framework for Knowledge Elicitation', *International Journal of Man-Machine Studies* **26**, 521-531.

Gentner D., 1989, 'Finding the Needle: Accessing and Reasoning from Prior Cases', Proceedings of 2nd Case-Based Reasoning Workshop, May 1989, FL, 137-143.

Gentner D., 1983, 'Structure-Mapping: A Theoretical Framework for Analogy', *Cognitive Science* **5**, 121-152.

Gentner D. & Stevens A.L., 1983, 'Mental Models', Lawrence Erlbaum Associates.

Gick M.L., 1989, 'Two Functions of Diagrams in Problem Solving by Analogy', *Knowledge Acquisition from Text and Pictures*, ed. H Mandi & J.R. Levin, Elsevier Science Publishers B.V. (North-Holland) 1989, 215-231.

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

Gick M.L. & Holyoak K.J., 1980, 'Analogical Problem Solving', *Cognitive Psychology* **12**, 306-355.

Gilmore D.J. & Green T.R.G., 1988, 'Programming Plans and Programming Experience', *Quarterly Journal of Experimental Psychology* **40A**, 423-442.

Glucksberg S. & Keysar B., 1990, 'Understanding Metaphorical Comparisons: Beyond Similarity', *Psychological Review* **97(1)**, 3-18.

Goguen J.A., 1986, 'Reusing and Interconnecting Software Components', *IEEE Computer* **19**, April 1986, 16-28.

Goguen J.A., Jirotka M. & Bickerton M.J., 1991, 'Research on Requirements Capture and Analysis', Internal Document, Oxford University Computing Laboratory, Programming Research Group, Centre for Requirements and Foundations, December 1991.

Green T.R.G., Gilmore D.J., Blumenthal B.B., Davies S. & Winder R., 1992, 'Towards a Cognitive Browser for OOPS', *International Journal on Human-Computer Interaction* **4(1)**, 1-34.

Greenspan S., 1984, 'Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition', PhD Thesis CSRG-155, Computer Systems Research Group, University of Toronto, March 1984.

Greenspan S., Borgida A. & Mylopoulos J., 1986, 'A Requirements Modeling Language and its Logic', *Information Systems* **11(1)**, 9-23.

Greenspan S., Feblowitz M., Shekaran C. & Tremlett J., 1991, 'Addressing Requirements Issues Within A Conceptual Modelling Environment', Proceedings 6th International Workshop on System Specification and Design, Como (It) 25-26th October 1991, IEEE Computer Society Press, 212-215.

Greiner R., 1988a 'Abstraction-based Analogical Inference', *Analogical Reasoning*, ed. D.H. Helman, Kluwer Academic Publishers, 147-170.

Greiner R., 1988b, 'Learning by Understanding Analogies', *Artificial Intelligence* 35, 81-125.

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

Guindon R., 1989, 'The Process of Knowledge Discovery in System Design', MCC Technical Report STP-166-89, Austin Texas, April 1989.

Guindon R. & Curtis B., 1988, 'Control of Cognitive Processes During Software Design: What Tools are Needed ?', Proceedings of CHI'88, ed. E. Soloway, D. Frye & S.B. Sheppard, ACM Press, 263-269.

Gupta N. & Nau D.S., 1991, 'Complexity Results for Blocks-World Planning', Proceedings of AAAI'91/MIT Press, 629-633.

Haddley N. & Sommerville I., 1990, 'Integrated Support for Systems Design', *Software Engineering Journal* 5(6), 331-338.

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.

Halff H.M., 1988, 'Curriculum and Instruction in Automated Tutors', *Foundations of Intelligent Tutoring Systems*, ed. M.C, Polson & J.J. Richardson, Lawrence Erlbaum Associates, 79-107.

Hall R.P., 1989, 'Computational Approaches to Analogical Reasoning: A Comparative Analysis', *Artificial Intelligence* **39**, 39-120.

Harandi M.T. & Lee H-Y., 1991, 'Acquiring Software Design Schemas: A Machine Learning Perspective', Proceedings of 6th Knowledge-Based Conference on Software Engineering, Syracuse NY, 22-25 September 1991, 239-250.

Harandi M.T. & Lubars M.D., 1985, 'A Knowledge-based Design Aid for Software Systems', Proceedings of Softfair II: Second Conference on Software Development Tools, Techniques and Alternatives, California, December 1985, Computer Society Press, 67-74.

Harandi M.T. & Young F.H., 1985, 'Template-Based Specification and Design', Proceedings of 3rd Intl Workshop on Software Specification and Design, London 1985, IEEE Computer Society Press, 94-97.

Harris D.R. & Johnson W.L., 1991, 'Sharing and Reuse of Requirements Knowledge', Proceedings of 6th Knowledge-Based Software Engineering Conference, Syracuse NY, 22-25th September 1991, 65-77.

Hayes-Roth B. & Hayes-Roth F., 1979, 'A Cognitive Model of Planning', *Cognitive Science* **3**, 275 - 310.

Hoc J-M. & Nguyen-Xuan A., 1990, 'Language Semantics, Mental Models and Analogy', *Psychology of Programming*, ed. J-M. Hoc, T. Green, R. Samurcay & D. Gilmore, Academic Press.

Hollan J.H., Hutchins E.L. & Weitzman I., 1984, 'Steamer: An Interactive Inspectable Simulation-Based Training System', *AI Magazine* **5**, 15-27.

Holyoak K.J. & Thagard P., 1989, 'Analogical Mapping by Constraint Satisfaction', *Cognitive Science* **13**, 295-355.

Holt R.W., Boehm-Davis D.A. & Schultz A.C., 1987, 'Mental Representations of Programs for Student and Professional Programmers', *2nd Workshop of Empirical Studies of Programmers,* ed. G. Olson, S. Sheppard and E. Soloway, Ablex, 33-46.

Huff K.E. & Thomson R., 1991, 'Supporting Understanding and Adaptation in Software Reuse', Proceedings of 1st Intl Workshop on Software Reusability, Dortmund July 3-5 1991, 45-50.

Indurkhya B., 1987, 'Approximative Semantic Transference: A Computational Theory of Metaphors and Analogies', *Cognitive Science* **11**, 445-480.

Iscoe N., 1991, 'Domain Modeling: Evolving Research', Proceedings of 6th Knowledge-Based Software Engineering Conference', Syracuse NY, IEEE Computer Society Press, 234-236.

Jackson M., 1983, 'Systems Development', Prentice-Hall International.

Jefferies R., Turner A.A., Polson P.G. & Atwood M.E., 1981, 'The Process Involved in Designing Software', *Cognitive Skills and Their Acquisition,* Lawrence Erlbaum Associates.

Johnson W.L., 1991, 'Interactive Acquisition of Requirements for Large Systems', Proceeding of Automated Software Design Workshop, AAAI-91, USC/ISI Technical Report RS-91-287, 61-70.

Johnson W.L., 1990, 'Understanding and Debugging Novice Programs', *Artificial Intelligence* **42(1)**, 51-97.

Johnson W.L., Feather M.S. & Harris D.R., 1991, 'The KBSA Requirements/ Specification Facet: ARIES', Proceedings of 6th Knowledge-Based Software Engineering Conference, Syracuse NY, IEEE Computer Society Press, 48-56.

Johnson-Laird P.N., 1983, 'Mental Models', Cambridge University Press.

Kaiser G.E. & Garlan D., 1987, 'Melding Software Systems from Reusable Building Blocks', *IEEE Software,* January 1987, 17-24.

Karakostas V., 1989, 'Requirements for CASE Tools in Early Software Reuse', *ACM*

SIGSOFT Software Engineering Notes **14(2)**, April 1989, 39-41.

Katsouli E. & Loucopoulos P., 1991, 'Business Rules in Information Systems Development', Proceedings of 2nd Workshop on the Next Generation of CASE Tools, ed. Tahvanainen V-P. & Lyytinen K., 481-504.

Katz S., Richter C. & Khe-Sing T., 1987, 'PARIS: A System for Reusing Partially Interpreting Schemas', MCC Technical Report Number STP-026-87, January 1987.

Keane M., 1987, 'On Retrieving Analogues When Solving Problems', *The Quarterly Journal of Experimental Psychology* **39A**, 29-41.

Kedar-Cabelli S., 1988a, 'Towards a Computational Model of Purpose-directed Analogy', *Analogica*, Pitman (London), 89-105.

Kedar-Cabelli S., 1988b, 'Analogy - from a unified perspective', *Analogical Reasoning*, ed. Helman D.H., Kluwer Academic Publishers.

Kolodner J.L., 1991, 'Improving Human Decision Making through Case-based Decision Aiding', *AI Magazine* **12**, Summer 1991, 52-68.

Koulek R.J., Salvendy G., Dunsmore H.E. & Lebold W.K., 1989, 'Cognitive Issues in the Process of Software Development: Review and Reappraisal', *International Journal of Man-Machine Studies* **30**, 171-191.

Lafontaine C., Ledru Y & Schobbens P-Y., 1991, 'An Experiment in Formal Software Development: Using in B Theorm Prover on a VDU Case Study', *Communications of the ACM* **34(5)**, 62-87.

Lange B.M. & Moher T.G., 1989, 'Some Strategies of Reuse in an Object-Oriented Programming Environment', Proceedings of CHI'89, ed. K. Bice & C. Lewis, ACM Press, 69-73.

Lawson B., 1980, 'How Designers Think: The Design Process Demystified', The Architectural Press, London.

Lee H-Y. & Harandi M.T., 1991a, 'Some Interative Aspects of a Software Design Schema Acquisition Tool', Proceeding of Automated Software Design Workshop, AAAI-91, USC/ISI Technical Report RS-91-287, 81-86.

Lee H-Y. & Harandi M.T., 1991b, 'Overcoming Shortcomings in Schema-based Software Design Systems', Proceedings 6th Intl Workshop on Software Specification and Design, Como (It) 25-26th October 1991, IEEE Computer Society Press, 246-249.

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.

Lenz M., Schmid H.A. & Wolf P.F., 1987, 'Software Reuse through Building Blocks', *IEEE Software*, January 1987, 34-42.

Littman D.C., 1987, 'Modeling Human Expertise in Knowledge Engineering: Some Preliminary Observations', *International Journal of Man-Machine Studies* **26**, 81-92.

Loucopoulos P. & Champion R.E.M., 1989, 'Knowledge-Based Support for Software Engineering', *Journal of Information Software and Technology* **31(3)**, 123-135.

Lubars M.D., 1988, 'A Domain Modelling Representation', MCC Technical Report STP-366-88, Software Technology Program, MCC, Austin Texas, November 1988.

Lubars M.D. & Harandi M.T., 1988, 'Addressing Software Reuse Through Knowledge-based Design', MCC Technical Report STP-058-88, Austin Texas, February 1988.

Lubars M.D. & Harandi M.T., 1986, 'Intelligent Support for Software Specification and Design', *IEEE Expert*, Winter 1986, 33-41.

Luqi, 1989, 'Software Evolution Through Rapid Prototyping', *IEEE Computer*, May 1989, 13-25.

Luqi & Ketabchi M., 1988, 'A Computer-Aided Prototyping System', *IEEE Software*, March 1988, 66-72.

Maarek Y.S., Berry D.M. & Kaiser G.E., 1991, 'An Information Retrieval Approach for Automatically Constructing Software Libraries', *IEEE Transactions on Software Engineering* **SE-17(8)**, 800-813.

Maiden N.A.M., 1991, 'Saving Reuse from the Noose: Reuse of Analogical Specifications through Human Involvement in the Reuse Process, *Journal of Information & Software Technology* **33(8)**, 1-11.

McCalla G.I. & Greer J.E., 1988, 'Intelligent Advising in Problem Solving Domains: The Scent-3 Architecture', Proceedings of ITS-88, June 1-3 1988, Montreal, Canada, 124-131.

McKeithen K.B., Reitman J.S., Reuter H.H. & Hirtle S.C., 1985, 'Knowledge Organisation and Skill Differences in Computer Programmers', *Cognitive Psychology* **17**, 26-85.

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

Miriyala K. & Harandi M.T., 1991, 'Automatic Derivation of Formal Software Specifications From Informal Descriptions', *IEEE Transactions on Software Engineering* **SE-17(10)**, 1126-1142

Mittermeir R.T. & Oppitz M., 1987, 'Software Bases for the Flexible Composition of Application Systems', *IEEE Transactions on Software Engineering* **SE-13(4)**, 440-460.

Mostow J., 1989, 'Design by Derivational Analogy: Issues in the Automated Replay of Design Plans', *Artificial Intelligence* **40**, 119-184.

Mylopoulos J. & Rose T., 1991, 'Case-based Reuse for Information Systems Development: The Techne Project', Proceedings of 1st Intl Workshop on Software Reusability, Dortmund Germany, 3-5 July 1991, 174-179.

Mylopoulos J., Borgida A., Jarke M. & Koubarakis M., 1990, 'Telos: A Language for Representing Knowledge About Information Systems (Revised)', Technical Report KRR-TR-89-1, Department of Computer Science, University of Toronto, August 1990.

Nanja M. & Cook R.C., 1987, 'An Analysis of the Online Debugging Process', *2nd Workshop of Empirical Studies of Programmers*, ed. G. Olson, S. Sheppard and E. Soloway, Ablex, 172-184.

Nathan M.J., 1990, 'Empowering the Student: Prospects for an Unintelligent Tutoring System', Proceedings of CHI'90, ed. J.C. Chew & J. Whiteside, ACM Press, 407-414.

Nature, 1992, Technical Annex ESPRIT Project 6353 (NATURE - Novel Approaches to Theories Underlying Requirements Engineering).

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

Neighbors J.M., 1980, 'Software Construction using Components', Ph.D. Dissertation, Department of Information and Computer Science, University of California, Irvine.

Novick L.R., 1988, 'Analogical Transfer, Problem Similarity, and Expertise', *Journal of Experimental Psychology: Learning, Memory and Cognition* **14(3)**, 510-520.

Novick L.R. & Holyoak K.J., 1991, 'Mathematical Problem Solving by Analogy', *Journal of Experimental Psychology: Learning, Memory, and Cognition* **17(3)**, 398-415.

Nwana H.S., 1991, 'FITS: A Fraction Intelligent Tutoring System', Proceedings AAAI'91, AAAIPress/MIT Press, 49-54.

Ohlsson S., 1986, Some Principles of Intelligent Tutoring Systems, *Instructional Science* **14**, 293 - 326.

Olson J.S., Olson G.M., Lisbeth M.A. & Wellner P., 1990, 'Concurrent Editing: The Group's Interface', Proceedings of INTERACT'90, ed. D. Diaper, D. Gilmore, G. Cockton & B. Shackel, North-Holland, 835-840.

Parnas D.L. & Clements P.C., 1986, 'A Rationale Design Process: How and Why to Fake It', *IEEE Transactions on Software Engineering* **SE-12(2)**, 251-257.

Payne S., 1988, 'Methods and Mental Models in Theories of Cognitive Skill', *Artificial Intelligence and Human Learning (Intelligent Computer-Aided Instruction)*, ed. J.A. Self, Chapman & Hall Computing, 69-87.

Pennington N. & Grabowski B., 1990, 'The Tasks of Programming', *Psychology of Programming*, ed. J.M. Hoc, T. Green, R. Samurcay & D. Gilmore, Academic Press.

Pennington N., 1987, 'Comprehension Strategies in Programming', *2nd Workshop of*

Empirical Studies of Programmers, ed. G. Olson, S. Sheppard and E. Soloway, Ablex, 100 - 113.

Perry T.S., 1991, 'Improving the World's Largest, Most Advanced System', *IEEE Spectrum*, February 1991, 22-36.

Polson M.C. & Richardson J.J., 1988, 'Foundations of Intelligent Tutoring Systems', Lawrence Erlbaum Associates.

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

Prieto-Diaz R., 1990, 'Domain Analysis: An Introduction', *ACM SIGSOFT Software Engineering Notes* **15**(2), April 1990, 47-54.

Prieto-Diaz R., 1985, 'A Software Classification Scheme', PhD Dissertation, Department of Information and Computer Science, University of California, Irvine.

Prieto-Diaz R. & Freeman P., 1987, 'Classifying Software for Reusability', *IEEE Software*, January 1987, 6-16.

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.

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

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

Reubenstein H.B. & Waters R.C., 1989, 'The Requirements Apprentice: An Initial Scenario', Proceedings of 5th Intl Workshop on Software Specification and Design, Pittsburgh, May 19-20 1989, 211-218.

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

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

Rissland E.L. & Skalak D.B., 1991, 'CABARET: Rule Interpretation in a Hybrid Architecture', *International Journal of Man-Machine Studies* **34**, 839-887.

Rist R.S., 1991, 'Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers', *Human-Computer Interaction* **6**, 1-46.

Rizzo A., Bagnara S. & Visciola M., 1988, 'Human Error Detection Processes', *Cognitive Engineering in Complex Dynamic Worlds*, Academic Press.

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

Rosch E., Mervis C.B., Grey W.D., Johnson D.M. & Boyes-Braem P., 1976, 'Basic Objects in Natural Categories', Academic Press.

Ross B.H., 1989, 'Distinguishing Types of Superficial Similarities: Different Effects on the Access and Use of Earlier Problems', *Journal of Experimental Psychology: Learning, Memory and Cognition* **15(3)**, 456-468.

Ross B.H., 1987, 'This is Like That: The Use of Earlier Problems and the Separation of Similarity Effects', *Journal of Experimental Psychology: Learning, Memory and Cognition* **13(4)**, 629-639.

Ross D.T. & Schoman K.E., 1977, 'Structured Analysis for Requirements Definition', *IEEE Transactions on Software Engineering* **3(1)**, 6-15.

Rosson M.B., Carroll J.M. & Bellamy R.K.E., 1990, 'Smalltalk Scaffolding: A Case Study of Minimalist Instruction', Proceedings CHI'90, ed. J.C. Chew & J. Whiteside, ACM Press, 423-429.

Rosson M.B., Maass S. & Kellogg W.A., 1988, 'The Designer as User: Building Requirements for Design Tools From Design Practice', *Communications of the ACM* **31(11)**, 1288-1297.

Roth E.M., Bennett K.B. & Woods D.D., 1988, 'Human Interaction with an "Intelligent" Machine', *Cognitive Engineering in Complex Dynamic Worlds*, Academic Press.

Russell S.J., 1989, 'The Use of Knowledge in Analogy and Induction' Pitman (London).

Ryan K., 1988, 'Capturing and Classifying the Software Developers Expertise', Proceedings of the International Workshop on Knowledge-Based Systems in Software Engineering, UMIST, March 1988.

Scacchi W., 1984, 'Managing Software Engineering Projects: A Social Analysis', *IEEE Transactions on Software Engineering* **10(1)**, 49-59.

Schank R.C., 1982, 'Dynamic Memory: A Theory of Reminding and Learning in Computers and People', Cambridge University Press.

Sein M.W., 1988, 'Conceptual Models in Training Novice Users of Computer Systems: Effectiveness of Abstract Vs Analogical Models and Influence of Individual Differences', PhD Thesis, School of Business, Indiana University, January 1988.

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

Sharp H., 1991, 'The Role of Domain Knowledge in Software Design', *Behaviour & Information Technology* **10(5)**, 383-401.

Shlaer S. & Mellor S.J., 1988, 'Object-Oriented Systems Analysis', Prentice-Hall.

Silverman B.G., 1985, 'The Use of Analogs in the Innovation Process: A Software Engineering Protocol Analysis', *IEEE Transactions on Systems, Man and Cybernetics* **SMC-15(1)**, 30-44.

Silverman B.G., 1983, 'Analogy in Systems Management: A Theoretical Enquiry', *IEEE Transactions on Systems, Man and Cybernetics* **SMC-13(6)**, 1049-1075.

Skwarecki E.J., 1988, 'Improving the Engineering of Model-Tracing Diagnosis', Proceedings of ITS-88, June 1-3 1988, Montreal, Canada, 215-221.

Sleeman D. & Brown J.S., 1982, 'Intelligent Tutoring Systems', Academic Press.

Smith D.R., Kotik G.B. & Westfold S.J., 1985, 'Research on Knowledge-Based Software Environments at Kestrel Institute', *IEEE Transactions on Software Engineering* **SE-11(11)**, 1278-1295.

Sommerville I., Mariani J., Haddley N. & Thomson R., 1989, 'Software Design with Reuse', internal document, Department of Computing, University of Lancaster, UK.

Sowa J.F., 1984, '*Conceptual Structures: Information Processing in Mind and Machine*', Addison-Wesley.

Sutcliffe A.G. & Maiden N.A.M., 1992, 'Analysing the Novice Analyst: Cognitive Models of Software Engineering', *International Journal of Man-Machine Studies* **36**, 719-740.

Sutcliffe A.G. & Maiden N.A.M., 1990a, 'Software Reusability: Delivering Productivity Gains or Short Cuts', Proceedings of INTERACT'90, eds. D. Diaper, D. Gilmore, G. Cockton & B. Shackel, North-Holland, 895-901.

Sutcliffe A.G. & Maiden N.A.M., 1990b, 'How Specification Reuse can Support Requirements Analysis', Proceedings of Software Engineering'90, edited by P. Hall, Brighton UK, July 24-27 1990, Cambridge University Press, 489-509.

Sweller J., 1988, 'Cognitive Load During Problem Solving: Effects on Learning', *Cognitive Science* **12**, 257-285.

Thagard P., 1988, 'Dimensions of Analogy', *Analogical Reasoning*, ed. Helman D.H., Kluwer Academic Publishers, 105-124.

Tracz W., 1990, 'Where Does Reuse Start?', *ACM SIGSOFT Software Engineering Notes* **15**(2), April 1990, 42-46.

Tsai J. J-P. & Ridge J.C., 1988, 'Intelligent Support for Specifications Transformation', *IEEE Software*, November 1988, 28-35.

VanLehn K., 1988, 'Student Modeling', *Foundations of Intelligent Tutoring Systems*, ed. M.C. Polson & J.J. Richardson, Lawrence Erlbaum Associates, 55-77.

Visser W. & Hoc J-M., 1990, 'Expert Software Design Strategies', *Psychology of Programming*, ed. J.M. Hoc, T. Green, R. Samurcay & Gilmore D., Academic Press.

Vitalari N.P., 1981, 'An Investigation of the Problem Solving Behaviour of Systems Analysts', PhD Thesis, University of Minnesota.

Vitalari N.P. & Dickson G.W., 1983, Problem Solving for Effective Systems Analysis: An Experimental Exploration', *Communications of the ACM* **26(11)**, November 1983, 948-956.

Volpano D.M. & Kierburtz R.B., 1989, 'The Templates Approach to Software Reuse', *Software Reusability (1)*, edited by T. Biggerstaff & A. Perlis, Academic Press.

Waters R.C., 1985, 'The Programmer's Apprentice: A Session with KBEmacs', *IEEE Transactions on Software Engineering* **SE-11(11)**, 1296-1320.

Weitzenfeld J.S., 1984, 'Valid Reasoning by Analogy', *Philosophy of Science* **51**, 137-149.

Wenger E., 1987, 'Artificial Intelligence and Tutoring Systems: Computational Approaches to the Communication of Knowledge', Morgan-Kauffman, Los Altos.

White B.Y. & Frederiksen J.R., 1990, 'Causal Model Progressions as a Foundation for Intelligent Learning Environments', *Artificial Intelligence* **42**, 99-157.

Wielinga B.J., Schreiber A.Th. & Breuker J.A., 1991, 'KADS: A Modelling Approach to Knowledge Engineering', Technical Report ESPRIT Project P5248 KADS-II, May 1991.

Wile D.S., 1983, 'Program Developments: Formal Explanations of Implmentations', *Communications of the ACM* **26(11)**, 902-911.

Winston P.H., 1982, 'Learning New Principles from Precedents and Exercises', *Artificial Intelligence* **19**, 321-350.

Winston P.H., 1980, 'Learning and Reasoning by Analogy', *Communications of the ACM* **23(12)**, 689-703.

Wood M. & Sommerville I., 1988, 'An Information Retrieval System for Software Components', *Software Engineering Journal* 3(3), 198-207.

Woods D.D. & Roth E.M., 1988, 'Cognitive Engineering: Human Problem Solving with Tools', *Human Factors* 30(4), 415-430.

Woolf B., 1988, 'Representing Complex Knowledge in an Intelligent Machine Tutor', *Artificial Intelligence and Human Learning (Intelligent Computer-Aided Instruction)*, ed. J.A. Self, Chapman & Hall Computing, 3-27.

Woolf B., Murray T., Suthers D. & Schultz K., 1988, 'Knowledge Primitives for Tutoring Systems', Proceedings of ITS-88, June 1-3 1988, Montreal, Canada, 491-498.

Yu E., 1991, 'Organisation Modelling and Composite Systems', Working Notes, AAAI Spring Symposium on 'Design of Composite Systems', Stanford University, March 26-28, 1991.

Bibliography

Journal Publications:

Maiden N.A.M. & Sutcliffe A.G., 1992, Exploiting reusable specifications through analogy, *Communications of the ACM.* **34(5)**, April 1992, 55-64.

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.

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

Maiden N.A.M., 1991, Saving reuse from the noose: reuse of analogous specifications through human involvement in the reuse process, *Information & Software Technology* **33(10)**, 780-790.

Sutcliffe A.G. & Maiden N.A.M., 1991, Analogical Software Reuse (Empirical Investigations of Analogy-Based Reuse and Software Engineering Practices), *Acta Psychologica* **78 (1-3)**, 1991, 173-197, also in Cognitive Ergonomics: Contributions from Experimental Psychology (1992), edited by G.C. van der Veer, S. Bagnara and G.A.M. Kempen, Elsevier Science, North-Holland.

Book Publications:

Maiden N.A.M. & Sutcliffe A.G., 1992, Specification reuse by analogy, *Next generation of CASE tools*, edited by V-P. Tehvanainen & K. Lyytinen, IOS Press. Also in Proceedings of 2nd workshop, next generation of CASE tools, 11-13th May 1991, Trondheim, Norway, edited by V-P. Tehvanainen & K. Lyytinen.

Maiden N.A.M. & Sutcliffe A.G., 1991, The abuse of reuse: why cognitive aspects of software reusability are important, Chapter 10, *Software Reuse (Utrecht 1989)*, edited by L. Dusink & P. Hall, Springer-Verlag (Workshops in Computing Series).

Conference Publications:

Sutcliffe A.G. & Maiden N.A.M., 1992, Supporting component matching for software reuse, Proceedings 4th Conference on Advanced Information Software Engineering

(CAiSE'92), Manchester 1992, edited by P. Loucopoulos, Lecture Notes in Computer Science 593 (Springer-Verlag) edited by G. Goos & J. Hartmanis, 290-303.

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.

Sutcliffe A.G. & Maiden N.A.M., 1990, How specification reuse can support requirements analysis, *Proceedings of Software Engineering'90*, edited by P. Hall, Brighton UK, July 24-27 1990, 489-509.

Sutcliffe A.G. & Maiden N.A.M., 1990, Software reusability: delivering productivity gains or short cuts, *Human Computer Interaction: Proceedings of INTERACT'90*, edited by D. Diaper, D. Gilmore, G. Cockton & B. Shackel, North-Holland, 895-901.

Workshop Publications:

Maiden N.A.M. & Sutcliffe A.G., 1991, Reuse of analogous specifications during requirements analysis, *Proceedings 6th Intl Workshop System Specification & Design*, Como (It) 25-26 October 1991, IEEE Computer Society Press, 220-223.

Maiden N.A.M. & Sutcliffe A.G., 1991, Interactive tool support for specification reuse by analogy, USC/ISI Technical Report RS-91-287 (*Proceedings of AAAI-91 workshop* - 'Automating Software Design: Interactive Design), 104-114.

Maiden N.A.M., 1991, Human issues in software reuse, *Proceedings Integrated Software Development with Reuse Seminar*, 3-4 December 1991, Heathrow UK, UNICOM.

Maiden N.A.M. & Sutcliffe A.G., 1990, Cognitive Studies in Software Engineering, *Proceedings of 5th European Conference on Cognitive Ergonomics (ECCE-5)*, Urbino Italy, September 3-6 1990, 157-171.

Maiden N.A.M. & Sutcliffe A.G., 1990, Exploiting reusable specifications through analogy, *Proceedings of 4th Intl. Workshop on CASE*, Irvine California, December 5-8 1990.

Sutcliffe A.G. & Maiden N.A.M., 1990, Assisting requirements analysis through specification reuse, *Proceedings of Workshop on The Next Generation of CASE Tools*, edited by S. Brinkemper and G. Wijers, Noordwijkerhout NL, April 9-10 1990.

Sutcliffe A.G. & Maiden N.A.M., 1990, Analysing the analyst: requirements for the next generation of CASE tools, *Proceedings of Workshop on The Next Generation of CASE Tools*, edited by S. Brinkemper and G. Wijers, Noordwijkerhout NL, April 9-10 1990.

Sutcliffe A.G. & Maiden N.A.M., 1989, Analogy in the reuse of structured specifications in a CASE environment, *Proceedings of 3rd Intl. Workshop on CASE*, Imperial College, London UK, July 17-21 1989.