[MEY85] Meyer B., 1985, 'On Formalism in Specifications', IEEE Software, January 1985, 6-26.

[MYL90] Mylopopoulos J., Borgida A., Jarke M. & Koubarakis M, 1990, 'Telos: Representing Knowledge about Information Systems', ACM Transactions on Office Information Systems 8(4), 325.

[PRI90] Prieto-Diaz R., 1990, 'Domain Analysis: An Introduction', ACM SIGSOFT Software Engineering Notes 15(2), April 1990, 47-54.

[PRE92] Pressman R.S., 1992, 'Software Engineering - A Practitioner's Approach', New-York, McGraw-Hill International.

[REU91] Reubenstein H.B. & Waters R.C., 1991, 'The Requirements Apprentice: Automated Assistance for Requirements Acquisition', IEEE Transactions on Software Engineering 17(3), 226-240.

[RIE89] Riesbeck C.K. & Schank R.C., 1989, 'Inside Case-based Reasoning', Lawrence Erlbaum Associates, Hillsdale NJ.

[ROS83] Rosch, E., 1983, 'Prototype Classification and Logical Classification: The Two Systems'. In New trends in conceptual representation: Challenges to Piaget's theory? (ed. E.K. Scholnick), Lawrence Erlbaum.

[SHA91] Shaw M., 1991, 'Heterogeneous Design Idioms for Software Architecture', Proceeding 6th International Workshop on Software Specification and Design', IEEE Computer Society Press, 158-165.

[SUT94] Sutcliffe A.G. & Maiden N.A.M., 1994, 'A Theory of Domain Knowledge in Requirements Engineering', Technical Report CU-94-00A, Centre for Human-Computer Interface Design, City University.

[SUT93] Sutcliffe A.G. & Maiden N.A.M., 1993, 'Bridging the Requirements Gap: Policies, Goals and Domains'. Proceedings 7th Intl Workshop on Software Specification & Design, IEEE Computer Society Press, 52-55.

[TAY94] Taylor C.N., Sutcliffe A.G., Maiden N.A.M. & Till D., 'Formal Representations For Domain Knowledge', Technical Report CU-94-00K, Centre for Human-Computer Interface Design, City University

[GIC83] Gick M.L. & Holyoak K.J., 1983, 'Schema Induction and Analogical Transfer', Cognitive Psychology 15, 1-38.

[HAR91] Harandi M.T. & Lee M.Y., 1991, 'Acquiring Software Design Schema: A Machine Learning Perspective', Proceedings 6th Knowledge Based Software Engineering Conf., IEEE Computer Society Press, 188-197.

[JAC83] Jackson M., 1983, 'Systems Development', Prentice-Hall International.

[JAR93a] Jarke M., Bubenko Y., Rolland C., Sutcliffe A.G. & Vassiliou Y., 1993, 'Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis', Proceedings of IEEE Symposium on Requirements Engineering, IEEE Computer Society Press, 19-31.

[JAR93b] Jarke M., Pohl K., Jacobs S., Bubenko J., Assenova P., Holm P., Wangler P., Rolland C., Plihon V., J. Schmitt, Sutcliffe A.G., Jones S., Maiden N.A.M., Till D., Vassilou Y., Constantopoulos P. & Spandoudakis G., 1993b, 'Requirements Engineering: An Integrated View of Representation', Proceedings 4th European Software Engineering Conference, Garmesh-Partenkirchen, ed. I. Sommerville, P. Manfred, Springer-Verlag, Lecture Notes in Computer Science 717, 100-114.

[JOH88] Johnson P., Johnson H., Waddington R. & Shouls A., 1988, 'Task-Related Knowledge Structures: Analysis, Modelling and Application', People and Computers IV (HCI-88), Cambridge University Press, 36-52.

[KED88] Kedar-Cabelli S., 1988, 'Analogy - from a unified perspective', Analogical Reasoning, ed. Helman D.H., Kluwer Academic Publishers.

[MAI94a] Maiden N.A.M., 1994, 'The Domain Space for Resource Containment', Nature Report CU-94-00E, Centre for Human-Computer Interface Design, City University.

[MAI94b] Maiden N.A.M., 1994, 'The Domain Space for Object Sensing', Nature Report CU-94-00D, Centre for Human-Computer Interface Design, City University.

[MAI93a] Maiden N.A.M., 1993, 'AIR's Problem Classifier', Nature Report CU-93-00H, Department of Business Computing, City University.

[MAI93b] Maiden N.A.M and Sutcliffe A.G. 1993, 'Requirements Engineering by Example', Proceedings 1st Intl Symposium on Requirements Engineering, IEEE Computer Society Press 104-112.

[MAI92a] Maiden N.A.M & Sutcliffe A.G. 1992a, 'Exploiting Reusable Specifications Through Analogy', Communications of the ACM 35(4), 55-64.

[MAI92b] Maiden N.A.M. & Sutcliffe A.G., 1992b, 'Analogously-based reusability', Behaviour & Information Technology 11(2), 79-98.

[MAI94c] Maiden N.A.M. & Sutcliffe A.G., 1994, 'Computational Mechanisms for Analogical Retrieval of Domain Knowledge during Requirements Engineering', Technical Report CU-94-00C, Centre for Human-Computer Interface Design, City University.

[MAI94d] Maiden N.A.M., Mistry P. & Sutcliffe A.G., 1994, 'How People Categorise Requirements for Reuse: A Natural Approach', Technical Report CU-94-00J, Centre for Human-Computer Interface Design, City University.

[MAI94e] Maiden N.A.M. & Sutcliffe A.G., 1994, 'Requirements Critiquing Using Domain Abstractions', Proceedings of IEEE Conference on Requirements Engineering, IEEE Computer Society Press, 184-193.

provide more than a few examples of abstractions. In contrast, our research represents the first systematic attempt to categorise software engineering problems and model the fundamental features of these categories for reuse.

The current space of problem abstractions numbers over 150 object system models. The domain modeling language has been used to define the high-level models listed in section 2.1 (e.g. [MAI94a, MAI94b]). Object system models for resource containment and allocation represent the largest segments of the problem space. These formal definitions are being used to populate the data base of problem abstractions which is central to the next version of the AIR toolkit.

The set of problem abstractions are also being validated. Formalisation of abstractions using order-sorted, first-order logic ensures their internal consistency [TAY94], which is a necessary precursor for successful reuse. Empirical studies has investigated the categorisation, structure and contents of the problem abstractions. Mental criteria and categories used by experienced software engineers to structure the space of software engineering problems have been elicited using card sorts to validate, and sometimes change the structure of the problem space [MAI94d]. Further validation will be supplemented using direct probing to determine the semantic contents of experts' mental abstractions.

Our reuse-oriented requirements engineering paradigm will also be validated through implementation of a working research prototype of the AIR toolkit [MAI92a]. Studies will evaluate the effectiveness of the paradigm through observation of requirements engineers will hands-on experience with the tools.

Acknowledgements

This research was funded by the European Union ESPRIT III programme 6353 NATURE basic research action. The authors wish to thank their colleagues for advice and comments.

References

[CHE85] Cheng P.W. & Holyoak K.J., 1985, 'Pragmatic Reasoning Schemas', Cognitive Psychology 17, 391-416.

[CON91] Constantopoulos P., Jarke M., Mylopoulos J. & Vassiliou Y., 1991, 'Software Information Base: A Server for Reuse', Technical Report, FORTH Research Institute, Univ of Heraklion, Crete.

[DEM78] De Marco T., 1978, 'Structured Systems Analysis and Specification', Prentice-Hall International.

[FAL89] Falkenhainer B., Forbus K.D. & Gentner D., 1989, 'The Structure-Mapping Engine: Algorithm and Examples', Artificial Intelligence 41, 1-63.

[GEN83] Gentner D., 1983, 'Structure-Mapping: A Theoretical Framework for Analogy', Cognitive Science 5, 121-152.

All COS system models are demonstrated using prototypical software engineering problems of each abstraction:

COS11: Sensing train movement for collision avoidance;
COS12: Sensing for collision avoidance in flexible manufacturing;
COS21 Sensing to maintain coastguard patrols in coastal water areas;
COS22 Sensing to position fire trucks at strategic city locations;
COS31: Ship navigation according to a predefined plan;
COS32: Flexible manufacturing according to a predefined plan;
COS41: Missile tracking towards a moving target aircraft;
COS42: Monitoring of direction of two ships towards a mid-ocean rendezvous;
COS52: Sensing to ensure that ships does not enter dangerous areas in the Thames estuary;
COS54: Sensing to ensure that factory robots do not deviate from paths into dangerous areas of the plant;
COS61: Controlled access of certain personnel to areas of a large building;
COS62: Controlled access of certain aircraft to airspaces around an airport;
COS63: Controlled access of discrete to non-permited areas;
COS64: Controlled access of robots to controlled areas of a factory.

3.2.5: Further Domain Modeling

This space of object system models provides a starting point for further, more detailed domain modeling. This provides an opportunity to overcome problems of deriving a complete ontology for software engineering problems by focusing on constructs belonging to certain problem classes. For example, real-time safety-critical problems are more likely to involve the representation of temporal issues. Object system models can be distinguished using different object properties, including whether the SpaceOne and SpaceTwo spaces are fixed or move in space and whether SpaceTwo is a space forbidden to all objects or whether it necessitates a warning. The former properties enable discrimination between collision avoidance in air traffic control and train signalling domains while the latter properties enable discrimination between sensing for aircraft collision avoidance and shipping lane guidance, were lane deviation does not result in immediate danger. Further domain modeling work in this direction is planned.

4: Conclusions and Future Research

This paper proposes reuse of formal problem abstractions as one mechanism for providing intelligent assistance for requirements engineers, as part of the ESPRIT 6353 NATURE basic research action [JAR93a]. This idea is not new. Indeed, a variety of models at different levels of abstraction have been put forward, for example clichés [REU91], patterns [HAR91, SHA91] and generalised application frames [CON91], to name but a few. However most approaches have failed to

The COS5 and COS6 system models are also specialised by the possible addition of one stative condition, similar to specialisation of the SOS1 and SOS2 system models in section 3.1, see Figure 5. Again there are two new object system models. One constrains the domain behaviour so that the invariant goal state cannot occur. The other does not stop the invariant goal state occuring, so no condition is applicable, however agents are warned so that corrective behaviour can be undertaken. The stative condition applies to object system models COS61 and COS63 but not to state transitions to COS52, COS554, COS62 and COS64. The COS51 and COS53 system models are not defined because, if access is not permitted to a space, stative conditions on movement to this space are not applicable. The formal definition of the stative condition applied to other COS system models is:

```
ControlAccess in StativeCondition, SimpleClass with
    attribute
        permits
            behaviour: ComplexObjectMovement;
        condition1
            when:       ObjectInSpace;
        condition2
            and:            PermitedObject
end ControlAccess
```

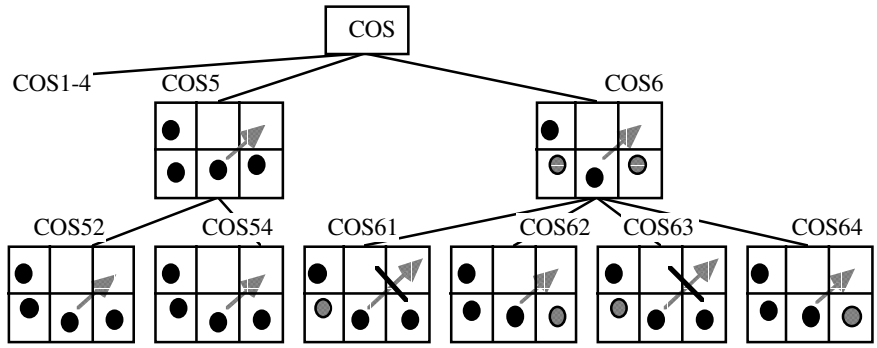


Figure 5. Specialisation of COS5 and COS6 system models through addition of stative conditions

3.2.3: Object Properties for Complex Object Sensing (COS)

Again all objects in complex object sensing system models are physical as defined for simple object sensing models, therefore no new object system models are generated.

3.2.4: Prototypical Examples for Complex Object Sensing (COS) System Models

```
end ObjectAtLocation
```

That is, for all instances of moving objects, each object must be in a predefined object space at a predefined future time. For most object system models, the default time constraint is that the goal state holds at all times in the domain. However, this is too restrictive, so several time constraints have been defined as goal state properties. A similar goal state is defined for the COS4 system model. The goal states for the COS5 and COS6 system models are similar to those for the SOS1 and SOS2 system models, except that the goal states are applicable to Spaces rather than SpaceTwo, as demonstrated by the following goal state for COS5:

```
RestrictedSpaceAccess in Structuregoal, SimpleClass with
    attribute
        level
            levelis:    Forall;
        goaltype
            oftype:Invariant;
        goalstate
            not:        ObjectInSpace;
        goalattribute
            and:        RestrictedSpace
end RestrictedSpaceAccess
```

where the attribute 'RestrictedSpace' is specialised to either a space which merits a warning or which is forbidden to moving objects.

3.2.2: Events and Stative Conditions in the Complex Object Sensing (COS) System Model

Complex object sensing system models are specialised using different types of event which trigger state transitions. Either object movement is continuous, so no discrete event can be defined, or a discrete event triggers one instance of the state transition. This discrete event is defined using the domain modeling language:

```
TriggerOneTransition in Event, Token with
    attribute
        eventname
            is called:      TriggerOne;
        starts
            triggers:       ComplexObjectMovement;
        eventattribute
            number:         One
end TriggerOneTransition
```

COS3: an object must be in one space at a predefined time;
COS4: multiple objects must be in one space at a predefined time;
COS5: no object must be in spaces which are prohibited;
COS6: no unpermited objects must be in spaces which are prohibited.

The goal states for COS1-4 have not been applied to the SOS system models. The COS3 and COS4 models represent location sensing against a plan, which is inapplicable if each moving object can only be in one of two spaces. There is a stronger case for inclusion of goal states from COS1 and COS2, however few real-world examples have been produced to support this case, and their inclusion in the problem space awaits empirical validation.

As before, the domain modeling language is used to define some of these goal states. The goal state which specialises for the COS1 system model is:

```
NeverGreaterThanQuantity in Structuregoal, SimpleClass with
    attribute
        level
            levelis:    Forall;
        goaltype
            oftype:Invariant;
        goalstate
            not:        ObjectInSpace;
        goalattribute
            and:        GreaterThanQuantity
end NeverGreaterThanQuantity
```

That is, for all objects, the number of objects in the space must not exceed a predefined quantity. This goal definition introduces different goal state properties. 'GreaterThanQuantity' imposes an additional constraint on the basic invariant state. A similar goal state is defined for the COS2 system model, except that the goal state quantity is 'LessThanQuantity'. Properties also have important role in the definition of the goal state for the COS3 system model:

```
ObjectAtLocation in Structuregoal, SimpleClass with
    attribute
        level
            oflevel:    Forall;
        goaltype
            oftype:Positive;
        goalstate
            and:        ObjectInSpace;
        goalattribute
            and:        SpaceIsDestination;
        timeconstraints
            and:        AtFutureTimeT
```

an ATM outside banking hours, giving access to bank customers only, and SOS22 instantiates to control of aircraft to a zone for which permission is needed. Other examples are given in [MAI94b].

3.2: Complex Object Sensing (COS) System Models

The space of complex object sensing system models is larger. However, all models are composed of the following objects:

```
MovingObject in KeyObject, SimpleClass end MovingObject
Spaces in StructureObject, SimpleClass end Spaces
Sensor in Agent, SimpleClass end Sensor
```

Each model has more than two spaces between which objects move. Each space is called a 'Space' in the domain modeling language and can be typed as a permited, warning or forbidden space, as described later in the paper. The behaviour of moving objects moving between different spaces is defined using the following state transition:

```
ComplexObjectMovement in PrimaryStateTransition, SimpleClass with
    inputobject
        takes:      Object;
    inputcardinality
        number:     One;
    outputobject
        creates:    Object;
    outputcardinality
        number:     One;
    from
        fromthe:    Spaces;
    to
        tothe:      Spaces;
    frequency
        howoften: Normal
end ComplexObjectMovement
```

3.2.1: Goal States for Complex Object Sensing

Six different goal states are added to specialise complex object sensing system models. These states define the required 'end-state' of a large number of problem instances including collision avoidance, plan adherence and monitoring for violation of restricted spaces. Each of the 6 states is considered in turn:

COS1: the quantity of objects in a space must never be greater than a quantity;
COS2: the quantity of objects in a space must never be less than a quantity;

3.1.3: Stative Conditions for Simple Object Sensing

The SOS1 and SOS2 object system models are further specialised by adding different stative conditions to state transitions, also see Figure 4. Stative conditions represent constraints on behaviour which are beyond the control of the system. Two new object system models are defined. One constrains the domain behaviour so that the invariant goal state cannot occur. The other does not stop the invariant goal state occurring, however agents are warned so that corrective behaviour can be undertaken. Therefore, a stative condition is included in object system models SOS12 and SOS22 whereas no condition is included for state transitions in models SOS11 and SOS21:

```
Controlaccess in StativeCondition, SimpleClass with
    attribute
        permits
            behaviour: Objectmoving;
        condition1
            when:       ObjectInSpace1;
        condition2
            and:              PermitedObject
end Controlaccess
```

That is, a specific MovingObject can only move from SpaceOne to SpaceTwo if and only if each object is in SpaceOne and the object is a 'PermitedObject'.

3.1.4: Object Properties for Simple Object Sensing

Objects in simple object sensing are all physical due to the physical nature of sensing the position of objects. MovingObject is assigned a simple attribute to replace the earlier definition of an object:

```
MovingObject in Keyobject, SimpleClass with
    attribute
        objectproperty
          is:     Physical
end MovingObject
```

This completes specialisation of the SOS system models.

3.1.5: Examples of Simple Object Sensing System Models

All simple object sensing system models can be demonstrated using prototypical software engineering problems. For example, SOS11 can be instantiated to forbid access for all personnel to a nuclear plant room, SOS12 instantiates to control of all aircraft around a forbidden fly zone, SOS21 instantiates to controlled access to

This definition demonstrates the use of slots. Each slot can either be a domain-independent term in the modeling language, such as the frequency *Normal*, or another construct defined using the language, such as the state 'ObjectSpaceOne'. Domain-independent terms are in *italics* and are used in the definition of all object system models.

3.1.2: Goal States for Simple Object Sensing

Two goal states further specialise simple object sensing, see Figure 4. First, no objects are permited to enter the restricted space SpaceTwo (SOS1). This goal state applies for all moving object instances, and is an invariant because it defines states which must not occur. Second, only a subset of permitted objects can enter this restricted space (SOS2). This second goal state is also an invariant and also applies for all object instances which are permited to enter the space. Both goal states are defined using the domain modeling language. For the first goal state (SOS1):

```
RestrictedspaceOne in Structuregoal, SimpleClass with
    attribute
        goallevel
            oflevel:        Forall;
        goaltype
            oftype:     Invariant;
        goalstate
            not:            ObjectSpaceTwo
end RestrictedspaceOne
```

For the second goal state (SOS2):

```
RestrictedspaceTwo in Structuregoal, SimpleClass with
    attribute
        goallevel
            oflevel:        Forall;
        goaltype
            oftype:     Invariant;
        goalstate
            not:            ObjectSpaceTwo;
        goalattribute
            and:            UnPermitedObject
end RestrictedspaceTwo
```

This second goal state reads that "All MovingObjects are forbidden to be in SpaceTwo if these objects are UnPermitedObjects", where 'UnPermitedObject' is a term in the domain modeling language. It is possible to define other, more detailed goal states, however further goals do not improve the level of potential reuse of object sensing system models.

'SpaceTwo', both of which can hold multiple objects. Definitions for SpaceOne are given below. Similar definitions exist for SpaceTwo.

```
SpaceOneInWorld in PrimaryState, SimpleClass with
    attribute
        firstobject
            is:          World;
        secondobject
            is:          SpaceOne;
        objectstructure
            linkedby:    Containsone
end SpaceOneInWorld

ObjectSpaceOne in PrimaryState, SimpleClass with
    attribute
        firstobject
            is:          SpaceOne;
        secondobject
            is:          Object;
        objectstructure
            linkedby:    Contains
end ObjectSpaceOne
```

One state transition changes the state of each moving object instance. The movement of objects is defined as normal rather than exceptional. Input and output cardinalities define that the moving object instance does not undergo transformation during the transition, unlike problems in which objects are composed, decomposed or destroyed [SUT94]:

```
SimpleObjectMovement in PrimaryStateTransition, SimpleClass with
    inputobject
        takes:       Object;
    inputcardinality
        number:      One;
    outputobject
        creates:     Object;
    outputcardinality
        number:      One;
    from
        fromthe:     ObjectSpaceOne;
    to
        tothe:       ObjectSpaceTwo;
    frequency
        howoften: Normal
end SimpleObjectMovement
```

Figure 3. Differences between simple and complex object sensing. Simple object sensing is composed of two spaces between which objects move, while complex object sensing has many spaces between which objects move

SOS and COS system models are distinguished by different numbers of spaces which influences the number, positioning and functionality of sensor agents central to the system. Furthermore, these sensors detect the existence of states in COS system models but changes between states in SOS system models when each moving object goes from SpaceOne to SpaceTwo.

3.1: Simple Object Sensing (SOS) System Models

The space of object system models which specialises the SOS system models is shown in Figure 4. These models have the following objects which are used to define their behaviour, structure, goals and constraints, as described in the following sections. Each object is defined using the domain modeling language:

MovingObject in KeyObject, SimpleClass end MovingObject
SpaceOne in StructureObject, SimpleClass end SpaceOne
SpaceTwo in StructureObject, SimpleClass end SpaceTwo
Sensor in Agent, SimpleClass end Sensor

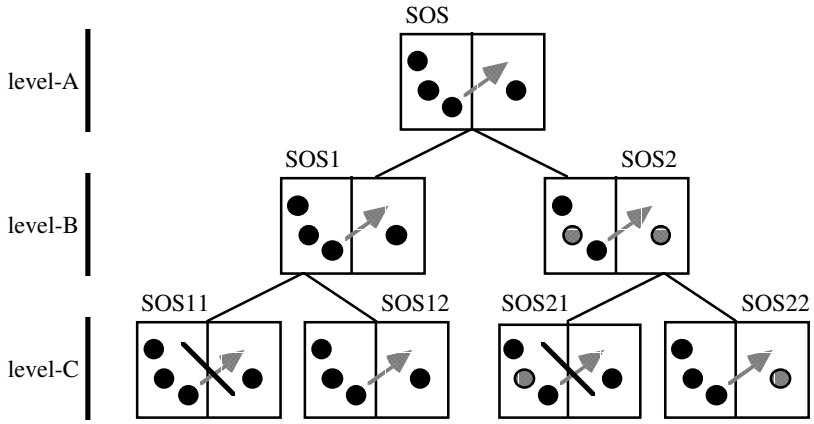


Figure 4. Object system models which specialise simple object sensing

3.1.1: Behaviour and Structure in SOS System Models

SOS system models have two spaces, and each moving object instance is always either in one space or the other. Critical domain behaviour is defined as the movement of objects between rather than within these spaces. The domain modeling language is used to define the existence of two spaces 'SpaceOne' and

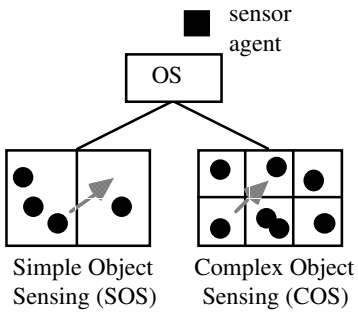
enable the critic to alert requirements engineers to problematic situations, avoid unnecessary interruptions of the requirements engineering process and allow user control over the critic's intervention strategy. These strategies are selected to aid detection and resolution of problem situations. Critiquing strategies assist detection of inconsistencies and ambiguities in a requirement specification. The requirements critic combines guidance and critiquing strategies in a mixed initiative dialogue with the requirements engineer.

All four tools exploit the set of problem abstractions for intelligent guidance. The remainder of this paper describes one segment of the problem space in more detail as a basis for describing how these tools operate.

3: The Segment of the Space for Object Sensing System Models

The remainder of the paper describes a set of object system models in which sensor agents detect the spatial/physical position of moving objects in a space. Examples of such object system models include monitoring for train collision avoidance, aircraft adherence to flight plans, monitoring prohibited zones for unlawful entry and controlling access to restricted parts of a building. Each model has been defined using both our domain modeling language [SUT94] and order-sorted, first-order logic [TAY94]. This section does not provide a complete formal definition of each object sensing system model. Instead, fundamental features are described using the domain modeling language, otherwise shorter, informal descriptions of the constructs are given to provide a basic understanding.

All object sensing system models have one or more sensor agents which monitor the physical location of moving objects. Object sensing system models are divided into two basic models according to differences in the domain structure, and hence behaviour with respect to that structure: (i) the problem has two spaces between which objects move, or: (ii) the problem has more than two spaces between which objects move, see Figure 3. These specialisations are called the simple (SOS) and complex object sensing (COS) models. Each is described in turn.



The requirements capturer has two approaches for acquiring new requirements and facts from the requirements engineer. First, guided requirements capture requests facts which are powerful discriminators between problem classes, thereby enabling matching and selection of abstractions. An intelligent dialogue which guided fact capture in this way was implemented in the first version of AIR, and evaluation of its use revealed moderate success with this strategy [MAI93b]. Second, unguided requirements capture is controlled by the requirements engineer. Requirements can be modeled using structured notations such as data flow diagrams [DEM78] or more informally using hypertext. Domain structure, behaviour, goals and constraints are inferred using a translation mechanism which exploits the knowledge types in the domain modeling language to infer fundamental domain features and functional requirements from existing notations. In both approaches however, the aim of the requirements capturer is to acquire facts which permit retrieval and explanation of problem abstractions.

2.2.3: Domain Matcher

The domain matcher is a computational analogical reasoning mechanism which retrieves object and information system models. The matcher infers mappings between abstractions and the new problem. This is achieved by first matching problem features to a high-level object system then refining this match to specialisations of the object system model until no further specialisation is possible. The domain matcher has two core mechanisms. Structure matching is used to establish an initial match with a high-level object system, therefore it is only called once during the matching process. Rule-based matching refines this initial match by mapping the requirement specification to lower-level specialisations of the initial object system. Rule-based matching also retrieves information system models linked to the retrieved object systems. Another tool, the problem notepad, exploits inferred object-pair and fact-pair mappings to aid detection of problem situations in the requirement specification [MAI93a]. The architecture and algorithms of the domain matcher have been implemented on a SparcStation IPX using Prolog by BIM, and are described in [MAI94c].

2.2.4: Requirements Critic

The requirements critic aids domain understanding and critiquing by explaining retrieved problem abstractions and detected problem situations [MAI94e]. Problems include incompleteness, inconsistencies and overspecification. The critic has five major features: (i) different guidance strategies which enable a mixed initiative dialogue with the requirements engineer; (ii) critiquing strategies to enable intelligent validation of requirements; (iii) strategies for explaining domain abstractions; (iv) a controller for strategies selection; (v) a problem notepad which reasons about mappings inferred from the domain matcher to detect and type problem situations [MEY85] for critiquing. Guiding and critiquing strategies

Specialisation is achieved through addition of the knowledge types using the domain modeling language. Detailed specialisation of the object sensing system model is shown in section 3.

2.2: The AIR Toolkit

AIR (Advisor for Intelligent Reuse) is an intelligent toolkit designed to enable reuse of problem abstractions and system specifications during requirements engineering. This paper discusses the components which enable reuse of problem abstractions prior to retrieval and transfer of analogical specifications [MAI92b]. AIR has four tools to capture key requirements, retrieve and explain problem abstractions and improve the requirement specification in light of this reuse, see Figure 2.

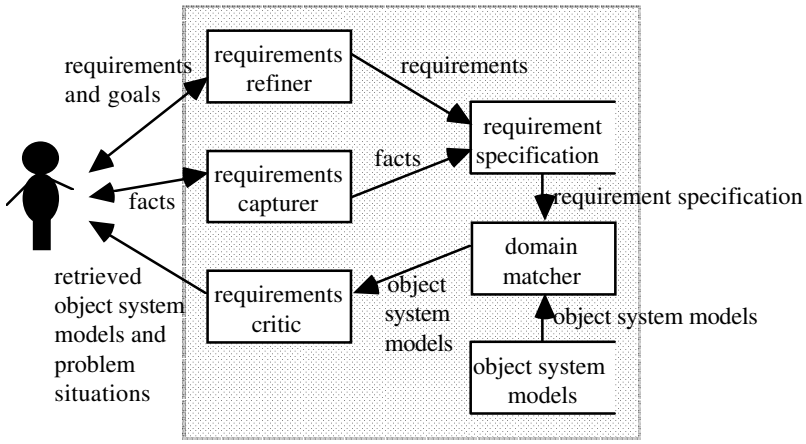


Figure 2. Overview of the AIR toolkit

2.2.1: Requirements Refiner

The requirements refiner is a hypertext tool which allows development of goal and requirement hierarchies which specialise and formalise high-level linguistic expressions of policy or need [SUT93]. The refiner is equiped with knowledge of goal types and heuristics to guide goal refinement. The requirements engineer is encouraged to type sub-goal nodes by picking goal types from menus. The refiner provides advice which is relevant to each goal type. This requirements capture and refinement is assisted by a second tool called the requirements capturer.

2.2.2: Requirements Capturer

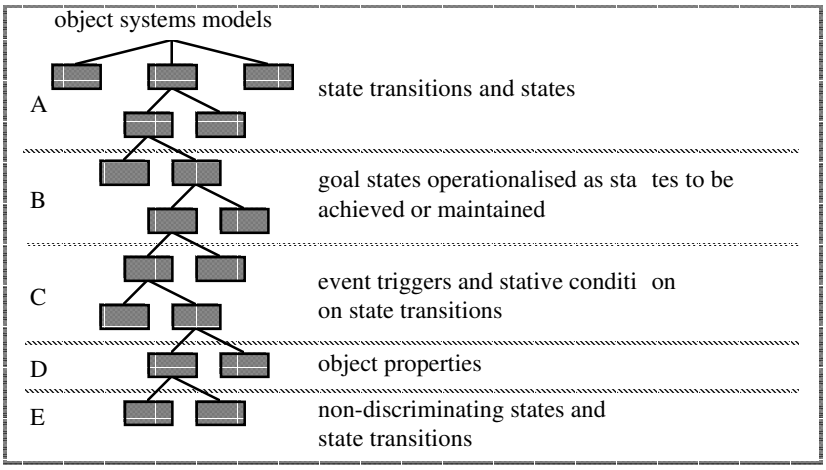


Figure 1. Hierarchical structure of object system models with addition of knowledge types for discrimination and matching

This hierarchical structure was derived from theoretical and example-based analysis and is claimed to be an effective categorisation of most software engineering problems for reuse. The 10 highest-level object system models indicate the breadth of the problem space [SUT94]:

- *resource usage* is characterised by the unidirectional transfer of resources from a holding to a client structure, for example sales order processing;
- *resource provision* is defined by the transfer of resources from a holding to a client structure, and their replenishment in the holding structure from a supplier structure, for example stock control;
- *resource returning* involves the transfer and return of resources between a holding and client structure, for example car hire;
- *item composition* aggregates objects to synthesise new ones, for example manufacturing domains which compose raw materials into a product;
- *item decomposition* is the opposite of composition;
- *resource allocation* models resource state changes prior to other transactions, for example booking domains;
- *object sensing* monitors object movement with respect to a spatial/physical structure, for example radar detection in air traffic control;
- *object messaging* communicates objects which have some information value between structures representing a communication topology;
- *agent object control* represents command and control applications in which agents pass messages which cause changes in object behaviour, for example ship control within a major port;
- *object manipulation* represents incremental changes in object states, for example computer based tools for word processing or computer-aided design.

object system models represent complex knowledge structures which can be transferred between problem instances.

At level-B, object system models representing the same domain behaviour and structure are discriminated by different goal states. These goal states define why domain behaviour with respect to structure takes place. Purpose is an important constraint on analogical inference (e.g. [GIC83, KED88]). Indeed, our empirical studies have shown the importance of goal states for discriminating between problems [MAI94d]. However, linguistic expression of purpose is often subjective, thus making retrieval of object system models difficult. Therefore, purpose in object system models is expressed as formal goal states, for example the most important goal state for collision avoidance is that no more than one aircraft is in any airspace at one time.

At level-C, events and stative conditions are added to discriminate between object system models. Event triggers and stative conditions on state transitions determine when and how the domain behaviour occurs to attain goal states. Events can be either domain or time events. Domain events model the result of some external activity in the real world while time events are similar phenomena emanating from a clock. Stative conditions constrain domain behaviour. Constraints are imposed by factors beyond the computerised system and are also used to define interaction between the system and domain. For aircraft collision avoidance, aircraft movement is continuous so no discrete events are defined, and no external factors control aircraft movement, so no stative conditions are needed.

At level-D, object properties discriminate between object system models. Each key object can either be physical, financial or conceptual. Object properties have been shown to be weak discriminators during analogical reasoning (e.g. [GEN83]). However, the theory claims that the fundamental properties of domain objects influences how these objects behave in the domain. Moving objects in collision avoidance problems tend to be physical rather than financial or conceptual, and are defined as such.

Object system models at level-E include non-discriminating states and state transitions. The structure of this domain space is illustrated in Figure 1.

models. Examples of such models are *report*, *count* and *sort*. The distinction between object and information system models is a development of Jackson's ideas [JAC83] on separating the model of the fundamental system entities from the functional processes which acquire information from that model.

Real-world requirements engineering domains are an aggregation of instantiations of several object and information system models. Object system models are linked by common objects and events which trigger behaviour manifest as state transitions in different models. Object system models also include agents to represent purposeful entities which undertake activities. Examples of domain aggregations include sales order processing, warehousing, allocation and air traffic control domains.

2.1: The Problem Space

The problem space is a hierarchical class structure of object system models and an orthogonal set of information system models. Object system models at different levels in this structure are distinguished using knowledge types with different powers of discrimination. The theoretical justification for this draws on assumptions from cognitive models of memory (e.g. [CHE85]), and in particular natural categories (Rosch 1983) and memory schemata [RIE89] which assert that human memory is organised as an informal hierarchy of classes. There is good evidence that memory for several different types of knowledge (e.g. objects, procedures and plans) is organised hierarchically. Applied theories of knowledge representation such as task knowledge structures [JOH88] embody classification structures. Mental representations of causally connected actions, properties and participants involved in common activities also exploit hierarchical classification of knowledge structures (e.g. [RIE89]). This generalisation, specialisation and inheritance of objects are familiar concepts in software engineering.

At the top level of the hierarchy (level-A), object system models are discriminated between using state transitions with respect to structure which represent causal domain behaviour. Empirical studies of the mental categorisation of problems by experienced software engineers has demonstrated the discriminating power of causal behaviour types [MAI94d]. One instance of the object sensing system model defined in this paper is aircraft collision avoidance. The fundamental movement of aircraft is expressed as state transitions with respect to airspaces which define the entire three-dimensional space in which aircraft move.

Structural and semantic relations have been added to improve the likelihood of analogical retrieval (e.g. [FAL89]) of object system models at level-A because computational and cognitive models of analogical reasoning emphasise the transfer of interrelated knowledge structures rather than isolated facts [GEN83]. Therefore,

deriving problem examples from software engineering textbooks (e.g. [PRE92]). The problem abstractions have also been validated against mental criteria and categories elicited from experienced software engineers using card sorting techniques [MAI94d].

This paper is in three parts. First, the theoretical basis and structure of the problem abstraction space are outlined, then the components of an intelligent requirements engineering toolkit are described and demonstrated. This is followed by an overview of the space of formal problem abstractions to indicate their breadth of coverage. Second, one segment of this problem space, in which sensors detect the spatial location of objects, is defined to demonstrate the hierarchical class structure, modeling language and nature of problem abstractions. The paper ends with a discussion of future research directions.

2: Main Components of the Domain Theory

The domain theory consists of four major components, the first two of which are described in this paper:

- a knowledge meta-schema or modeling language which defines the semantics for the set of formal problem abstractions [SUT94]. It is based on the Telos knowledge representation language [MYL90];
- a set of formal problem abstractions stored in a hierarchical structure [SUT94]);
- a computational analogical retrieval mechanism which retrieves problem abstractions for incomplete and inconsistent requirement models [MAI94c];
- a method which bridges the gap between informal linguistic expression of requirements to formal expression of semantics [SUT93].

Problem abstractions are composed of object and information system models. Object system models define the fundamental domain behaviour, structure, goals and constraints. These models introduce 'structures' as bridging abstractions which preserve some manifestation of the physical structure found in the real world. Key objects which undergo behaviour with respect to this structure, may be physical and subject to gravity or momentum, financial objects with some monetary value or conceptual objects without physical instantiation. Examples of object system models are resource hiring (which instantiates to *book lending*), item composition (*stock picking*), resource usage (*sales order processing*) and object messaging (*electronic mail systems*).

Information system models represent processes which feed off, report on and provide external representations of the object system model. Object system models define problem structure and behaviour while information system models describe processes forming solutions for requirements. Information system models are domain-independent and provide an orthogonal set of abstractions to object system

A Set of Formal Problem Abstractions for Reuse During Requirements Engineering

N.A.M. Maiden, A.G. Sutcliffe, C. Taylor & D. Till
Centre for Human-Computer Interface Design
City University, Northampton Square, London EC1V OHB

Abstract

This paper presents a reuse-oriented approach to requirements engineering. It argues for a set of formal problem abstractions which represent the fundamental behaviour, structure, goals, constraints and objects of different classes of software engineering problems. It describes an intelligent toolkit to exploit a data base of problem abstractions during a range of requirements engineering activities. This approach, and the nature of the problem abstractions, are demonstrated through a detailed description of one segment of the problem space for sensing the physical and spatial location of moving objects.

Keywords: Requirements Engineering, Software Reuse, Domain Modeling

1: Requirements Engineering

Requirements engineering is the complex technical, social and cognitive process which produces functional and non-functional requirements for a software system [JAR93a]. It is also the most difficult and expensive phase during construction of a computer system. Errors during requirement specification, such as incompleteness and inconsistencies, are harder to detect and more expensive to repair if undetected during requirements engineering. One solution advocated by the ESPRIT 6353 NATURE project [JAR93a] is to populate intelligent requirements engineering toolkits with problem abstractions to assist requirement specification [REU91, MAI92a]. Problem abstractions are formal models of the fundamental behaviour, structure, goals, constraints and functions of all instances of a problem class. This paper outlines the problem abstraction space and describes an intelligent toolkit for retrieval and reuse of problem abstractions during the requirements engineering process. It also includes a formal definition of one segment of the problem space to demonstrate the abstractions and provide templates for reuse.

Our approach claims that most software engineering problems belong to a tractable set of hierarchical classes which share general features [SUT94]. We define the semantic content, categorisation and structure of the set of problem abstractions to maximise their reuse. Problem abstractions are distinguished using a small set of knowledge types. The current set of 150 abstractions has been derived from extensive analyses of many software engineering problems. Strategies for this analysis include domain analysis [PRI90], case studies (e.g. [JAR93b]) and