Applying AI Techniques to Requirements Engineering: The NATURE Prototype*

Klaus Pohl[1], Petia Assenova[2], Ralf Doemges[1], Paul Johannesson[2], Neil Maiden[3], Veronique Plihon[4], Jean-Roch Schmitt[4], Giwrgos Spanoudakis[5]

Abstract: *Requirements Engineering (RE) is a critical part of software engineering. Within the NATURE (Novel Approaches to Theories Underlying Requirements Engineering) project we have devoloped and implemented& five theories which are based on AI techniques for supporting and improving the requirements engineering process. For making the results comparable we have used the well known library example [Wing 1990].*

Our contibution demonstrates that

- *requirements engineering can be essentially improved by applying AI techniques*
- *combining AI techniques has positive synergy effects on requirements engineering*

1. Introduction

In software engineering, the requirements engineering (RE) phase has traditionally been perceived as the fuzzy and somewhat dirty part in which a formal specification is gained from informal ideas. The process of developing a specification is poorly understood. In addition, as practise shows, the final specifications tend to be incomplete, ambiguous, error prone etc. To improve current RE practise, a better understanding of RE is needed.

What does it take to capture, maintain and use requirements information? The NATURE framework [Pohl 1994] derives three dimensions of interest from the major sources of problems in requirements engineering (cf. figure 1):

---

* This work is supported in part by ESPRIT Basic Research Project 6353 (NATURE).

1 Informatik V, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Germany; email: pohl@informatik.rwth-aachen.de

2 SISU-ISE, Isafjordsgatan 26, 1250 Kista, Sweden

3 Business Comp., City University, London ECIV OHB, UK

4 Universite Paris 1, rue de al Sorbonne 17, 75231 Paris, France

5 ICS-FORTH, Dedalou 36, 71110 Heraclion, Greece

& We like to thank all our students which helped us in implementing the first version of our prototype, namely: Michael Gebhardt, Fred Gibbels, Peter Haumer, Christof Lenzen, Klaus Weidenhaupt, Claudia Welter, S. Si-Said and L. Corre

- As a basis for formal system development, requirements engineering moves along a *representational axis* typically from informal to more formal representations, ideally ending with a formal specification that can be transformed to executable code. Modeling and relating these representations is mostly a technical problem albeit one whose solution needs good knowledge of human-computer interaction.
- A second obvious goal is to end up with a complete system *specification*. We see this as orthogonal to the first problem: it is perfectly possible to hide poor understanding behind a lot of formalism, as well as to describe a very deep understanding in quite informal terms. Moves along this axis mostly face cognitive and psychological problems of requirements engineering.
- The third axis is concerned with the *agreement* reached on the current specification. Requirements engineering is understood here as a negotiation process which should lead towards at least sufficient agreement to start building a system. Hence this axis mainly deals with the social aspects of requirements engineering.

At the level of process execution, RE can be understood as a movement in this space which, in addition to the technical, cognitive, and social barriers, is also influenced by economical factors and the methods used. For example, the traditional requirements capture process starts near the origin with different personal views, little system understanding and a completely informal representation; it should end with agreement on a well-understood and formally described specification.

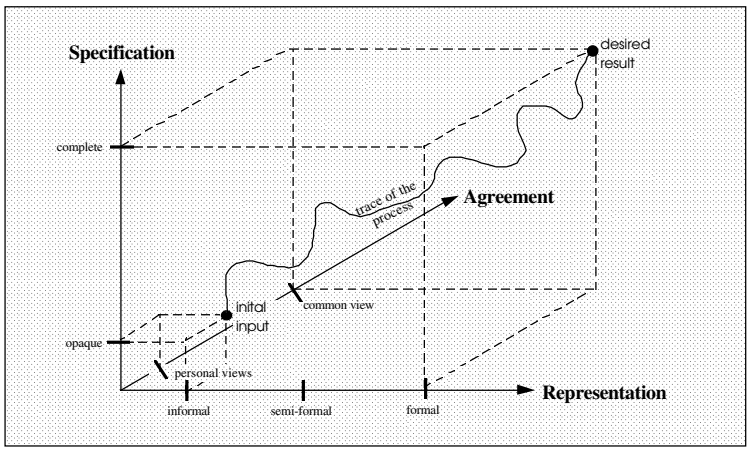

Figure 1. The three dimensions of requirements engineering.

For supporting the RE process we have developed five approaches which are heavily based on results gained by AI research.

First, process guidance is enabled by adopting results of decision-oriented process models and enhances them by representing the fact that developers react contextually according to the situation they are faced with (section 2).

Second, using knowlegde representation and reasoning techniques we capture knowledge about the RE process execution along the three dimension, thus enabling process traceability (section 3).

Third, knowledge about existing systems can be gained and reused by applying reverse engineering (section 4).

Fourth, reuse of specification is possible through similarity-based search. Our model for computing the similarity between software specifications is based on analogical reasoning techniques (section 5).

Fifth, we use domain abstractions for supporting the definition and critiquing of requirements. Two complex computational mechanisms, domain matcher and problem classifier, have been designed and implemented to maximise the leverage from retrieving domain abstractions (section 6).

2 Process-Theory (by J.R. Schmitt and V. Plihon)

2.1 Description of research

Software systems development in general and its Requirements Engineering phase in particular are characterized by their highly creative nature, making their process difficult to express [Wijers 1991]. This characteristic has two main consequences [NATURE 1992]. Firstly, the methods (ways-of-working) cannot be expressed precisely enough by using an algorithm-like planning approach. Secondly, although CASE tools are efficient in recording, retrieving and manipulating system specifications, they fail in actually supporting the developers in proceeding in the development.

The NATURE way of tackling these two interrelated problems is to concentrate on the development process itself and to propose a general framework that is powerful enough to allow the building of knowledge bases containing precise ways-of-working definitions and underlying more helpful CASE tools. This framework is in fact a model of the software systems development processes.

The process model we develop in NATURE inherits from the decision-oriented class of models [NATURE 1992] , and enhances them by representing the fact that developers react contextually according to the situation they are faced with. Our process modeling approach strongly couples the decisions with the situations they are made on, thus improving the representation of the "when" and "how" to decide on "what" aspects.

The five central concepts of our process model are the following: (1) a *situation* is most often a part of the product under development it makes sense to make a decision on; (2) a *decision* reflects a choice that a developer makes at a given point in time of the process; (3) a *context* consists of the association of a situation and a decision; (4) an *action* performs a transformation of the product; and (5) an *argument* is a statement which supports or objects to a decision within a given situation.

We further refine this model by representing the inherent granularity levels of development processes. Firstly, some contexts may be *compound* from more *atomic* ones. Secondly, some contexts are directly applicable by performing product transforming actions while others may be applied in several alternative ways, thus requiring a choice from the developer. Contexts of the former type are called *micro-contexts*, those of the latter type are called *macro-contexts*.

The process model we briefly sketched above (see [NATURE 1993] for a complete presentation) allows to precisely describe methods ways-of-working and serves as a structure for the Knowledge Bases underlying what we call "process-aware CASE tools".

2.2 Potential to address practical software problems

We believe that our process modeling approach can bring solutions to the following two problems, generally encountered in software systems development, i.e. the lack of :

- complete and precise understanding and representations of the methods ways-of-working,

- process-aware CASE tools capable of supporting most of the developers' tasks.

Firstly, to be really useful, ways-of-working should describe all their components, whatever their granularity level is (phases, decisions, actions, etc.). They should also describe the relationships between these components as completely as possible. In addition, this information should be provided for engineering activities (the development itself) as well as for project management activities.

Moreover, methods are rarely used as they are defined but are rather continuously adapted to development contingency factors.

These two characteristics of ways-of-working make their plan-based definition unrealistic and strongly argue for a knowledge-based approach [Rolland et al. 1990]. In NATURE, we follow this approach by considering ways-of-working as semantic networks whose nodes and links are instances of the process model. These semantic networks are expressed in the knowledge representation language Telos [Mylopoulos et al. 1990].

Secondly, besides process traces, developers can be provided with two other kinds of support from process-aware CASE tools: control and guidance.

A control facility ensures that the developers proceed in an allowed way. The guidance facility still goes one level higher by suggesting ways to proceed to the developers. In the terms of our process model, controlling means ensuring that correct decisions are made when they can be made, and guiding means suggesting treatable situations and applicable decisions that can be made on them. These two kinds of support require that treatable situations be recognized in the product and that decisions be retrieved from the process knowledge base.

Thus, an expert-system oriented architecture is surely more adapted to offer these two kinds of support than a procedural one. After the OICSI Expert System based prototype [Cauvet et al. 1988] [Grosz and Rolland 1990], we have experimented the guidance in NATURE by developing an active CASE tool prototype built around the ConceptBase KBMS [Jarke et al. 1992].

2.3 Example for Process Theory

To illustrate the use of our process model to define a way-of-working, stored in a Process Knowledge Base (PKB) and used by a guidance CASE tool, let's consider a process of refining a library management E/R schema.

In performing this refinement, the developer follows the way-of-working which defines all the contexts that are usable to proceed in the refinement.

As such, it contains for example the context associating a situation made of a couple entity-type/relation-type and a decision which is to "Improve" it. As there are several ways to improve this situation, the context is a macro-context. The PKB thus relates it to several other contexts that are its alternatives, along with arguments that support the choice between alternatives. One of these alternatives is for instance the context that "Partition" the entity-type (creation of mutually exclusive subtypes).

Furthermore, this context is a compound one as it is built from more atomic ones (some being themselves compound). The PKB then relates the "Partition" context to its component contexts. At the bottom of this decomposition, contexts are atomic-contexts and hence directly applicable to product transforming actions, also stored into the PKB.

Although very simple, the example shows how the concepts of our approach can be used to precisely define ways-of-working. This can be done both at a very low level of granularity (as in the example) or at a much more macroscopic one.

Once ways-of -working are defined in this way, they can serve as basements for process-aware CASE tools. For example, the prototype we developed in NATURE helps the developer in proceeding in the same refinement step by supporting this process in the following way :

- At the beginning, the developer is faced to a set of entity-types, relation-types, etc. , from which it is difficult to know what to treat and what to do (cf. appendix figure 2.1)
- The tool then recognizes treatable situations in this set and highlights them, thus easing the situation selection task. The developer selects the "User/loan" situation (instance the entity-type/relation-type situation of the way-of-working definition).
- Once the developer has selected the situation, the tool looks up all the contexts of the way-of-working definition and retrieves the ones that are applicable to the situation: it then lists the decisions set . For instance, it proposes the "Improve" decision and its alternatives with arguments that support the choice among them.
- After the choice of the "Partition" context, which is a micro-context, the tool is able to automatically performing its associated actions, possibly after a compound-context decomposition (the corresponding micro context is shown in appendix figure 2.2).

3 Knowledge Representation Theory

(by Ralf Doemges and Klaus Pohl)

Within the NATURE project knowledge representation plays a central role.

First, each approach presented in this paper uses the same knowledge representation language (Telos [Mylopoulos et al. 1990], [Jeusfeld 1992] and its implementation ConceptBase [Jarke 1992]) for formalizing and representing the meta-models as well as for recording the data, e.g., reusable process chunks or domain models.

Second, due to the conformable representation an integration of the approaches is enabled. By interrelating the different meta-models, each approach can use knowledge of others. For example, knowledge about product parts, which is part of reusable process chunks, can be adapted from generic domain model.

Third, by capturing knowledge about the RE process the process itself as well as the specification become traceable and therefore changes are easier to integrate [Jarke and Pohl 1993a, Jarke and Pohl 1993b].

In this paper we concentrate on the last aspect: enabling requirements traceability.

3.1 Description of Research

According to [Gotel and Finkelstein 1993, Ramesh and Edwards 1993] requirements traceability can be divided into *pre-traceability* and *post-traceability.* Pre-traceability

deals with the interrelation between the requirements and their origin (business goals, persons, user needs etc.). Post-traceability enables the trace from the requirements down to design objects as well as implementation nd documentation objects (and vise versa). Within NATURE we deal with pre-traceability.

For enabling requirements pre-traceability two questions must be answered.

3.1.1 Which kind of knowledge must be captured ?

Looking at the RE process using the three dimensional framework (cf. section 1) knowledge along the *representation, specification* and *agreement* dimension must be recorded and interrelated. For capturing this knowledge according to its content we have developed orthogonal meta models for each dimension: an IBIS-based model for structuring the knowledge about the agreements reached, a formal hypertext model for structuring the informal information, and a comprehensive specification model gained out of over 25 standards and guidelines for the specifications. The interrelation between the models are expressed using constraints and integrity rules [Jeusfeld and Jarke 1991].

These models are used for content-oriented trace capture. Relations between instances of the over 50 model classes are represented using multiple instanciation and dependency links, which are generated by our tool environment (cf. section 3.1.2).

3.1.2 How is this knowledge captured ?

Of course, manual recording of the information is too expensive. Therefore automated (at least partially) recording is necessary, which can only be provided by a suitable CASE environment.

Within the NATURE environment the execution of each action (not tool !!) as well as the input used and output produced is automatically recorded. Further, dependencies are created for representing relations between the inputs and outputs and therefore enable backtracking and trace. Since actions are often nested, an interoperable tool environment is needed. We have developed an interoperability approach which enables the communication between actions and control of the execution of nested action. Due to space limitation, we do not explain this in detail here.

3.2 Potential to address practical software problems

Capturing knowledge about the RE process along the three dimensions provides the basis for RE traceability. Requirements traceability itself is the basis for:

- the adaption of software systems; the enviroment the systems are embedded in continously changes. Hence, the software systems must be adapted to these changes and therefore requirements which were defined in the first place must be revised. This may lead to a more detailed definition or even redefinition of requirements. Finding the*right* requirements, which are affected by a given change, is only possible, if the requirements are related to their origin during the RE process.
- revising decisions; often there exist altenative sets of requirements for archiving a given goal. A revision of an decision (the choice of another set), e.g., caused by the need for change, is much more effective and consistent, if the alternatives are captured during RE and are therefore known.
- requirements freedoms [Feather and Fickas 1991]; the integration of informal and formal specification enables the requirements engineer to express in the first place his knowledge about the system in an natural way allowing inconsistency, incompleteness, different levels of abstraction, etc. Out of this information a formalization can be made later on. Due to the trace, the informal information is not lost during formalization.
- easing design and implementation decisions; the trace of the RE process provides background information, which may cause, that a decision is made in a better way.
- increasing the understanding of the specification and the software system; often requirements are defined out of informal statemants. Relating these statements to the derived (more formal) requirements can serve as an explanation later on. Further, if people have the possibility to trace the process, they are able to understand why certain decisions where made and what the alternatives had been. In addition, they can trace back to the reasons, explaining why a particular requirement must be fullfilled.
- process improvement; capturing data about process execution builds the basis for process improvement. The trace of the process enables the process engineering team to detect and understand process execution which does not conform to the

defined way of working. This enables process improvement as generalization out of experience.

3.3 Example for knowledge representation theory

We use the well known library example to illustrate our approach. Suppose, that out of the informal statements (cf. [Wing 1990]) a more formal specification was already gained. A new informal requirement is added indicating that *"proceedings should only be checked out for three days"*.

To integrate the new requirement, the engineer looks at the current ER diagram. Since *proceedings* seem to be a possible specialization of the entity *book* he wants to get more information about this entity, especially the decisions. Using the dependency links he retrieves all decisions dealing with the entity *book* , as well as the informal requirements on which the decisions are based.

He recognizes that there has been a decision **not** to specialze the entity *book*.

Looking at the decision, the reasons for declining the specialization of book can be gained. Further, browsing through the informal requirements - shown in the HT-editor - the requirements engineer recognizes, that there is a requirement stating *"journals should not be checked out"* (cf. appendix, figure 3.1).

The requirements engineer takes the whole information (informal requirement, decisions and their arguments) to the next group meeting, to discuss the integration of the "new" and the "old" requirement."*journals should not be checked out)*" - enabled through the use of process trace.

The RE team decides now that there should be a spezialisation of *book* into *proceedings* **and** *journals*, each with different lend durations.

After the meeting, the engineer revises the old decision (cf. appendix, figure 3.2). Thereby he adds an new argument and, in addition, he links the informal requirements to the revised decision.

The capture of the process information is supported by our tool environement.

4 Reverse Engineering

(by Petia Assenova and Paul Johannesson)

Interest in software reusability has increased in recent years. "Just as you rely on established theorems in a new mathematical proof, you should build new systems as much as possible with existing parts" [Fischer 1987]. The reusable components of a system can be code, databases, subsystems, algorithms and sometimes requirements specifications [Davis 1988]. Reusing system components requires an understanding of those parts that are to be reused. In order to obtain a better understanding, we need to view the system at a higher level. To attain this view we can use reverse engineering to raise a system from the level of implementation to the level of design and specification [Johannesson 1994].

4.1 Potential to address practical problems

The use of reverse engineering (RevEng) can be of benefit in the following situations:

- RevEng may be used as an aid to concept formation during the acquisition of initial requirements. This is particularly valuable in situations where the domain is large and its structure complex.
- RevEng can be used for validation. In order to check that the schema is reasonable and complete a conceptual schema, expressing requirements for a new system can be compared to that of an existing system. The conceptual schema of the existing application is derived through reverse engineering.
- RevEng may be used to identify reusable components of existing systems.
- RevEng may be used to find points of integration when two systems are to be integrated.
- RevEng may be used for comparing a system specification with a standard system. For this purpose a conceptual specification of the standard system may be derived by means of reverse engineering.
- RevEng may be used to derive conceptual schemes from existing systems to be included in a knowledge base of the domain.

In the context of information systems engineering, the process of reverse engineering starts with analysing an existing system (to a similar enterprise to the observed enterprise) in order to identify its components and their relationships. A description of the system at a higher level of abstraction is created. This description is often expressed in an object-based, conceptual modelling language in which it is possible to handle declarative enterprise rules. Reverse engineering usually includes a knowledge acquisition process, which adds (semantic) information not contained in the original system.

A part of the process of reverse engineering is translation a relational database schema into a conceptual schema. Works in this area have appeared since the beginning of the 1980s. Some important contributions are [Casanova 1983, Dumpala and Arora 1983, Briand 1987, Navathe and Avong 1987, Markowitz and Makowsky 1990, Castellanos and Saltor 1991, Shoval and Zohn 1993, Johannesson 1994]. There are also commercial products that transform relational schemas into ER schemas, e.g. Bachmans reverse engineering tool, [Bachman 1989]. [Briand 1987] describes a method that creates a schema in an extended ER model from a minimal cover of functional dependencies. [Dumpala and Arora 1983] gives a method for translating from the relational model to the original ER model, which not only considers the structure of the data but also its behaviour, i.e. how updates are performed. [Shoval and Zohn 1993] represents a method for transforming a relational model at any level of normalisation into a binary relationship model. The most complete method proposed for translating from the relational model into an extended ER model seems to be [Castellanos and Saltor 1991]. With the exception of [Markowitz and Makowsky 1990] and [Johannesson 1994] all methods proposed in the papers above are informal. One of the few who show, that the schemas produced have the same information capacity as the original relational schemas is [Johannesson 1994].

4.2 Example for Reverse Engineering

The reverse modelling method we have implemented is described in [Johannesson 1994]. A basic assumption in this method is that all relations given must be in third nor-

mal form, and that all keys and inclusion dependencies must be specified. The first steps of the method transform the relational schema into a form appropriate for identifying object structures. Roughly speaking, the transformations aim at obtaining a one-to-one relationship between relations and object types by splitting and collapsing relations. Normally, a relation corresponds to a single object type. However, if a relation contains several keys, it may correspond to more than one object type. For example, the relation COUNTRY[Name, Capital, Population, Capital_population] corresponds to two object types: COUNTRY and CAPITAL (we assume that Capital is a candidate key in COUNTRY).

If a user decides that a relation corresponds to several object types, it is split into several relations. Another case where a single relation may correspond to several object types is when an inclusion dependency exists, the right hand side of which is not a key. In this case, the right hand side of the inclusion dependency may correspond to an additional object type. In some cases, several relations may correspond to a single object type. This occurs when the relations are involved in a cycle of inclusion dependencies. In order to obtain a one-to-one relationship between relations and object types, the relations will in such a case be collapsed into a single one.

When the transformations outlined above have been applied, the resulting relational schema is translated into a conceptual one by mapping each relation into an object type and each inclusion dependency into either a generalisation relationship or an attribute. It should be noted that not in all cases does the method automatically determine if an inclusion dependency corresponds to a generalisation relationship or an attribute. This must sometimes be determined by a user with knowledge of the semantics of the relational schema.

5 Similarity Theory (by Giwrgos Spanoudakis)

5.1 Similarity for Analogical Software Reuse

Analogical reasoning is an appropriate paradigm for reuse to the extent that it contributes to solutions for its retrieval and comprehensibility problems [Biggerstaff and Richter 1987, Krueger 1992]. The identification of analogies between abstract knowledge structures(source descriptions stored in a repository) and structures reflecting directly the mental models of their inventors(target descriptions specified by software engineers) based on partial correspondences between their elements supports both the retrieval and the comprehension of the former.

In this problem area, we have developed a model computing the similarity between software artifacts of any type(i.e code, designs or specifications) [Spanoudakis and Constantopoulos 1993b], so as to promote their analogical reuse. Similarity is estimated on the basis of conceptual descriptions of artifacts, expressed in the Telos knowledge representation language [Mylopopoulos et al. 1990], as an aggregate function of partial metrics, measuring their distances with respect to distinct modeling abstractions. Telos supports three abstractions common in semantic and object oriented data models, namely the classification the generalization and the attribution. Since it allows multiple and meta classification as well as typed attribution, Telos can be used for specifying different models of describing artifacts and thus it is an appropriate representational framework for the similarity model. The estimation of similarity does not assume any domain specific representation constructs or other special forms of knowledge(e.g the user supplied distance measures between facet terms in [Pietro-Diaz and Freeman 1987, Ostertag 1990]). Moreover, the employed distance metrics are restricted by principles on how humans assess similarity, distilled from Psychology [Spanoudakis and Constantopoulos 1993a]. These features enhance the possibility of employing the model in tools supporting reuse of artifacts of any type and substance [Pietro-Diaz 1993], in cooperation with humans.

A first prototype of the similarity computation model has been implemented in C++ and integrated with the Semantic Index System [Constantopoulos and Doerr 1993], an outgrowth of the Software Information Base, developed in the ESPRIT project ITHACA [Constantopoulos et al. 1993]. This integration has resulted in a tool providing general knowledge retrieval(i.e querying and browsing) and viewing facilities(i.e text and graphical views, object cards with hypertext functionalities) as well as general types of similarity queries for analogical retrieval. This prototype(reffered to as the Software Requirements Analyzer in the following) has been initially tested through a scenario of reusing requirements specifications of existing software systems in developing a specification for a University Library System within the ESPRIT project NATURE [Jarke et al. 1993].

5.2 A Scenario of Specifying Requirements by Reuse

In the context of requirements engineering similarity is viewed as a basic technique that complements other forms of reasoning(e.g deduction, inheritance) and searching(e.g browsing) in providing intelligent aid to analogical reuse of specifications.

The detailed demonstration of the similarity analysis in the Software Requirements Analyzer prototype is based on a scenario of specifying a transaction returning books from their borrowers back to the University Library System, namely the ReturnLibrary–Item transaction. In this scenario similarity is utilized in different ways. In particular, it is used for assigning prototypical examples of using models for expressing specifications, for ranked retrieval, and for detecting analogies between specifications that may lead to reuse of their elements. Also, similarity may be used for reorganizing repositories of software specifications.

The assignement of prototypical examples of models is carried out through a pairwise similarity analysis between all the instances (i.e specifications) of a class that defines the relevant specification model. Prototypicality is estimated as the average similarity of some specification with the other specifications built according to the relevant model. As it is shown in appendix figure 5.1, the transaction *ResourceCheckIn* can be used as an example on how to specify transactions supporting the return of borrowed resources back to their holding systems.

Ranked retrieval may be carried out through the selection of a target specification and a class whose instances will be analyzed for their similarity to this target. These instances may be further inspected for reuse in a descending similarity order. For instance, we should examine the CheckInItem transaction before the ReturnMotorbike transaction in trying to reuse their elements for further specifying the ReturnLibraryTransaction.

The analysis of the similarity between two specifications, namely the target(i.e the one that is being developed) and the source(i.e the existing one) specification, results in a graph presenting their analogous and their unique elements. This distinction of elements is based on domain independent criteria of semantic homogeneity [Spanoudakis and Constantopoulos 1993b] and is estimated in a way ensuring the maximum possible coherency between the pairs of the analogous elements of the two specifications. Furthermore, it may accomplish the further development of the target specification in three distinct ways.

First, the elements which are unique to the source specification may be missing in the target due to incompleteness and as such they could be reused either as they are or by being modified. For instance, the "carriedOutby" element of the "CheckInItem" transaction that indicates the agent who is responsible for executing it, is missing from the partial specification of the "ReturnLibraryItem" transaction, as it is shown in appendix figure 5.2. Thus, we should specify a similar element for this transaction, too.

Also, the analogous elements may be further inspected through similarity analysis or browsing to check if they serve their roles in similar ways. Fine differences may give rise to changes in the target specification.

For instance the "borrowingIdentification" element of the "ReturnMotorbike" transaction serves the same role with the "hiringIdentification" element of the "ReturnLibraryItem" transaction but in a different way (cf. appendix figure 5.3). The identification of the past borrowing activity relies on some general information about the borrower in the case of the "ReturnLibraryItem" transaction and on a code(i.e the "MACode") assigned to the borrower in the case of the "ReturnMotorbike" transaction. Assuming that codes eliminate any sort of ambiguities in identifications, this latter solution could be adopted for the "ReturnLibraryItem" transaction, as well.

Finally, those elements that are unique to the target specification may be redundant and thus their necessity should be further investigated.

6 Domain Theory (by Neil Maiden)

6.1 Description of Research Approach

Requirements engineering is complex, error prone and in need of intelligent support for capturing complete, consistent requirement specifications and guiding the requirements engineering process. One solution advocated by the ESPRIT Nature project [Jarke et al. 1993a] is to populate tools with domain abstractions to assist requirement specification [Maiden and Sutcliffe 1992]. Domain abstractions represent the fundamental behaviour, structure and functions of a domain class. We propose that domain abstractions can aid modelling specification, critiquing of complex system requirements and describes computational mechanisms from an artificial intelligence perspective for that purpose. Two complex computational mechanisms have been designed and implemented to maximise the leverage from retrieving domain abstractions:

- the domain matcher [Maiden and Sutcliffe 1993] is a hybrid computational mechanism for analogical reasoning between a requirement specification and one or more domain abstractions. It integrates structure matching algorithms and heuristic-based reasoning with domain semantics to infer consistent fact-pair and object-pair mappings with abstractions. It is similiar to existing computational models of analogical reasoning, such as the Structure-Matching Engine [Falkenhainer et al. 1989] and the Analogical Constraint Matching Engine [Holyoak and Thagard 1989], and hybrid case-based reasoners such as Grebe [Branting 1991]. This computational mechanism is supported by an intelligent dialogue manager [Maiden and Sutcliffe 1994] which explains retrieved domain abstractions to requirements engineers. Explanation strategies are needed to ensure effective comprehension and adaptation of domain abstractions;
- the problem classifier uses mappings inferred by the domain matcher to detect possible problem situations in the requirement specification [Maiden 1993]. It detects four types of problem: incompleteness in the requirement specifications, overspecification, inconsistencies and noise. The classifier complements existing approaches such as propositional deduction and dependency maintenance (e.g. [Reubenstein and Waters 1991]). The problem classifier is integrated with a cooperative tool for explaining complex problem situations to requirements engineers.

The domain matcher and problem classifier are implemented using ConceptBase [Jarke 1992] with Telos [Mylopoulos et al. 1990] and BIM Prolog as part of Nature's coordinated demonstration.

6.2 Practical Problems Solved by the Domain Matcher & Problem Classifier

Retrieval of explanation of domain abstractions enables resolution of many important requirements engineering problems [Roman 1985, Meyer 1985]. Advantages from the domain matcher and problem classifier are five-fold:

- explaining domain abstractions to requirements engineers enables detection of incompleteness, overspecifications, inconsistencies, ambiguities and wishful thinking in requirement specification. The requirements engineer achieves this by inferring mappings between application and abstraction, assisted by explanation of domain abstractions;
- the problem classifier detects possible problem situations including incompleteness and inconsistencies which may not have been detected by the requirements engineer. The classifier differs from existing computational mechanisms because to retrieves domain knowledge to detect incomplete and incorrect specifications;
- domain abstractions also aid structuring and scoping of new requirements problems. Structuring and scoping complex software engineering problems has proved difficult, especially for less-experienced software engineers [Sutcliffe and Maiden 1992];
- retrieval of domain abstractions is a precursor to reuse of analogical specifications which instantiate the same domain abstraction. Reuse of analogical specifications has been shown to be successful despite the need for additional support tools [Maiden and Sutcliffe 1992];
- retrieved domain abstractions can aid communication between parties during requirements engineering by providing a common basis for understanding, similar to the role of program clichés in the Programmer's Apprentice.

Furthermore, the computational mechanisms can aid software engineers to retrieve design abstractions and detect design problems at other phases of the software development process.

6.3 Scenario Showing Benefits from the Domain Theory

This short scenario demonstrates active guidance and critiquing during requirements engineering from successful retrieval of domain abstractions. The requirements critic [Maiden and Sutcliffe 1994] explains retrieved domain abstractions to a requirements engineer with moderate experience to assist requirement specification for a computerised stock control system in a library. The requirements engineer has determined the need for a stock control system for a lending library. The system must identify missing or damaged books which are no longer in the library. It must also permit a stock take of books within the library. Books cannot be purchased without staff authorisation, see [Jarke et al. 1993b]. The requirements critic exploits the domain specialisation hierarchy reported in [Maiden and Sutcliffe 1994]. The high-level object system represents the concept of resource containment and is specialised by the OM object system to include resource repletion. This specialised abstraction can be instantiated to all stock control domains. This shown interaction between the tool and requirements engineer results from successful retrieval of an object system model by the domain matcher and problem detection by the problem classifier (cf. appendix figure 6.1). The OM object system and linked information systems defining functional requirements are explained to the requirements engineer:

- the domain abstraction is explained using visualisation and text-based descriptions of domain structure and behaviour, formal definitions and explanation of object-pair mappings. Further domain explanation and exploration is available by requesting animation of domain behaviour;
- the dialogue controller selects active critiquing because the problem classifier detects a large number of problems;
- guided explanation of information systems and prototypical examples takes place;
- information systems are explained using formal definitions, visualisation, animation and prototypical examples.

These explanations are possible due to retrieval of domain abstractions by the domain matcher and detection of problem situations by the problem classifier.

7 References

[Bachman 1989] Bachman C.W.: A Personal Chronicle: Creating Better Information Systems, with some Guiding Principle, *IEEE Transactions on Knowledge and Data Engineering*, 1(1), 1989, pp. 17-32

[Biggerstaff and Richter 1987] Biggerstaff T. and Richter C.: Reusability Framework, Assessment and Directions, *IEEE Software*, March 1987

[Branting 1991] Branting L.K.: Building Explanations from Rules and Structured Cases, *International Journal of Man-Machine Studies*, 34, 1991, pp. 797-837

[Briand 1987] Briand H.: From Minimal Cover to Entity-Relationship Diagram, in *Seventh International Conference on Entity-Relationship Approach,* 1987

[Casanova 1983] Casanova M.A.: Designing Entity-Relationship Schemas for Conventional Information Systems, in *Third International Conference on Entity-Relationship Approach,* 1983

[Castellanos and Saltor 1991] Castellanos M. and Saltor F.: Semantic Enrichment of Database Schemas: An Object Oriented Approach, in *First International Workshop on Interoperability in Multidatabase Systems*, eds. Y. Kambayashi et al., 1991, pp. 71-78

[Cauvet et al. 1988] Cauvet C., Rolland C., and Proix C.: Information Systems Design: An Expert System Approach, *Proc. of the Int. Conf. on Extending Database Technology,* Venice, Italy, March 1988

[Constantopoulos and Doerr 1993] Constantopoulos P. and Doerr M.: The Semantic Index System: A Brief Presentation, ICS-FORTH, 1993

[Constantopoulos et al. 1993] Constantopoulos P., Doerr M., and Vassiliou Y.: Repositories for Software Reuse: The Software Information Base, in *Proceedings of the IFIP Conference in the Software Development Process*, Como, Italy, 1993

[Davis 1988] Davis J.: CASE Viewed as a Solution to Backlog Problems, *CASE outlook*, 2(2), 1988

[Dumpala and Arora 1983] Dumpala S.R.and Arora S.K.: Schema Translation Using the Entity- Relationship Approach, in *Entity-Relationship Approach to Information Modeling and Analysis*, 1983

[Falkenhainer et al. 1989] Falkenhainer B., Forbus K.D., and Gentner D.: The Structure-Mapping Engine: Algorithm and Examples, *Artificial Intelligence* 41, 1989, pp. 1-63

[Feather and Fickas 1991] Feather M. and Fickas S.: Coping with Requirements freedom, Proc. Int. Workshop *Development of Intelligent Information Systems*, Canada, 1991, pp. 42-46

[Fischer 1987] Fischer G.: Cognitive View of Reuse and Redesign, *IEEE Software,* July, 1978

[Gotel and Finkelstein 1993] Gotel O. and Finkelstein A.: An Analysis of the Requirements Traceability Problem, Technical Report, Imperial College, Department of Computing, TR-93-41, 1993

[Grosz and Rolland 1990] Grosz G. and Rolland C.: Using Artificial Intelligence Techniques to Formalize the Information System Design Process, *Proc. of Database and Expert Systems Applications*, eds. A.M. Tjoa and R. Wagner, Vienna, Austria, August 1990, pp. 374-380

[Holyoak.and Thagard 1989] Holyoak K.J. and Thagard P.: Analogical Mapping by Constraint Satisfaction, *Cognitive Science* 13, 1989, pp.295-355

[Jarke 1992] The ConceptBase Manual, Version 3.1, Aachener Informatik-Berichte, Nr. 92-17, ed. Jarke M., RWTH-Aachen, Fachgruppe Informatik, 1992

[Jarke et al. 1993a] Jarke M., Bubenko Y., Rolland C., Sutcliffe A.G., and Vassiliou Y.: Theories underlying requirements engineering: an overview of NATURE at genesis, *Proceedings of IEEE Symposium on Requirements Engineering*, IEEE Computer Society Press, 1993, pp. 19-31

[Jarke et al. 1993b] Jarke M., Pohl K., Jacobs S., Bubenko J., Assenova P., Holm P., Wangler B., Rolland C., Plihon V., Schmitt J.R., Sutcliffe A.G., Jones S., Maiden N.A.M., Till D., Vassilou Y., Constantopoulos P., and Spandoudakis G.: Requirements Engineering: An Integrated View of Representation, to appear in *Proceedings 4th European Software Engineering Conference*, Garmisch-Partenkirchen, September 1993.

[Jarke and Pohl 1993a] Jarke, M. and Pohl K.: Vision Driven System Engineering, in *Information System Development Process* IFIP Transaction, eds. Prakash N., Rolland C. and Pernici B., Como, Italy, September 1.-3., 1993, North-Holland, pp. 3-22.

[Jarke and Pohl 1993b] Jarke, M. and Pohl K.: Establishing Visions in Context: Toward a Model of Requirements Engineering, Proc. of the 14th Int. *Conf. on Information Systems*, Orlando, Florida, December 5.-8., 1993, pp. 23-34

[Jeusfeld 1992] Jeusfeld, M.: Änderungskontrolle in deduktiven Objektbanken, DISKI Volume 17, Bad Honnef, Germany, INFIX Publ. (Diss. Univ. Passau, in German)

[Jeusfeld and Jarke 1991] Jeusfeld, M. and Jarke, M.: From relational to object oriented integrity constraint simplification,.in *Proc. 2nd Int. Conf. on Deductive and Object Oriented Data Bases,* Munich, Springer Verlag, 1991.

[Johannesson 1994] Johannesson P.: A Method for Translating Relational Schemas into Conceptual Schemas, to appear in *Tenth International Conference on Data Engineering,* Ed. M.Rusinkiewicz, Houston, 1994

[Krueger 1992] Krueger C.: Software Reuse, *ACM Computing Surveys*, 24(2), 1992

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

[Maiden and Sutcliffe 1994] Maiden N.A.M. and Sutcliffe A.G.: Requirements Critiquing Using Domain Abstractions, to appear in *Proceedings First International Conference on Reqiuirements Engineering*, IEEE Computer Society Press, 1994

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

[Maiden and Sutcliffe 1992] Maiden N.A.M. and Sutcliffe A.G.: Exploiting Reusable Specifications Through Analogy, *Communications of the ACM*, 34(5), April 1992, pp. 55-64, 1992

[Markowitz and Makowsky 1990] Markowitz V. and Makowsky J.: Identifying Extended Entity-Relationship Object Structures in Relational Schemas, *IEEE Transactions on Software Engineering*, 16(8), 1990, pp. 777-790

[Meyer 1985] Meyer B.: On Formalism in Specifications, *IEEE Software*, January 1985, pp. 6-26

[Mylopopoulos et al. 1990] Mylopopoulos J., Borgida A., Jarke M., and Koubarakis M.: Telos: Representing Knowledge about Information Systems, *ACM Transactions on Office Information Systems*, 8(4), 1990, p. 325

[NATURE 1992] Jarke M., Pohl K., Jacobs S., Bubenko J., Assenova P., Holm P., Wangler B., Rolland C., Plihon V., Schmitt J.R., Sutcliffe A.G., Jones S., Maiden N.A.M., Till D., Vassilou Y., Constantopoulos P., and Spandoudakis G.: Novel Approaches to Theories Underlying Requirements Engineering: NATURE Initial Integration Report, Deliverable NATURE.D-I, December, 1992

[NATURE 1993] Deliverable D-P-1: Formal Definition and Evaluation of Requirements Engineering Process Meta Model, eds. Jarke M., Sutcliffe A.G., Vassiliou Y., Rolland C., and Bubenko J, August 1993

[Navathe and Avong 1987] Navathe S. B. and Avong A. M.: Abstracting Relational and Hierarchical Data with a Semantic Data Model, in *Seventh International Conference on Entity-Relationship Approach,* 1987.

[Ostertag et al. 1992] Ostertag E.: Computing Similarity in a Reuse Library System: An AI-Based Approach, *ACM Transactions on Software Engineering and Methodology*, 1(3), July 1992

[Pietro-Diaz 1993] Pietro-Diaz R.: Report: Software Reusability, *IEEE Software*, May 1993

[Pietro-Diaz and Freeman 1987] Pietro-Diaz R. and Freeman P.: Classifying Software for Reusability, *IEEE Software*, January 1987

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

[Pohl '94] Pohl, K.: The Three Dimension of Requirements Engineering: A Framework and its Application, *Information Systems*, 19(2), 1994.

[Ramesh and Edwards 1993] Ramesh B. and Edwards M.: Issues in the Development of a Requirements Traceability Model, in *IEEE Int. Symposium On Requirements Engineering*, San Diego, California, January 4.-6., 1993

[Rolland et al. 1990] Rolland C., Cauvet C., and Proix C.: The role of Artificial Intelligence in Information System Design, in Volume on *Intelligent Systems: state of the art and future directions*, eds. W.R. Zbigniew and M. Zemankova, Ellis Horwood Limited Pub, 1990

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

[Shoval and Zohn 1993] Shoval P. and Zohn S.: Database reverse engineering: From the Relational to the Binary Relationship model, *Data & Knowledge Engineering*, 6(3), May 1993

[Spanoudakis and Constantopoulos 1993a] Spanoudakis G. and Constantopoulos P.: Similarity for Analogical Software Reuse: A Conceptual Modeling Approach, *Proceedings of CAiSE '93*, LNCS 685, June 1993

[Spanoudakis and Constantopoulos 1993b] Spanoudakis G. and Constantopoulos P.: Similarity for Analogical Software Reuse: A Computational Model, submitted for publication, 1993

[Sutcliffe and Maiden 1992] Sutcliffe A.G. and Maiden N.A.M.: Analysing the novice analyst: cognitive models in software engineering, *International Journal of Man-Machine Studies*, 36, 1992, pp. 719-740

[Wijers 1991] Wijers G.M.: Modeling Support in Information Systems Development, PhD Thesis, Thesis Publishers Amsterdam, 1991

[Wing 1990] Wing J.: A specifier´s introduction to formal methods, *IEEE Computer,* 23(9), 1990, pp. 8-26.

Appendix (screen dumps of prototypes)

Process Theory

Figure 2.1: Dedection of a situation

Figure 2.2: Micro context for "partitioning"

Knowledge Representation Theory

Figure 3.1: Decisions and informal requirements for entity book

Figure 3.2: Revision of a decision

Similarity

Figure 5.1: Selection of Prototypical Examples of Specification Models

Figure 5.2: Similarity Between Transactions

Figure 5.3: Similarity Between Transactions (Case 2)

Domain Theory

Figure 6.1: Result of the match between the current specification and the domain abstraction