

How Knowledge Representation meets Software Engineering (and often Databases)

Alex Borgida

Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08904, USA
`borgida@cs.rutgers.edu`

Abstract. This paper surveys a selection of personal research projects which addressed problems related to Software Engineering, and whose solution was inspired by ideas from the field of Knowledge Representation and Reasoning. Surprisingly frequently the research was also related to problems in Databases. We discuss, in part, to what extent did the KR ideas provide ready-made solutions to SE and DB problems, and how frequently we had to invent new KR techniques.

1 Introduction

The readers of this journal do not need to be convinced of the viability and interest in the enterprise of applying knowledge representation and reasoning (KR) research to the problems of software engineering (SE). After all, this journal's statement of intent includes "Knowledge representations and artificial intelligence techniques applicable to automated software engineering are of interest." It would be hopeless to attempt a review of this entire field, since it is too broad. I will therefore consider only work directly related to my own research, and in doing so, observe that in my case most advances in Software Engineering problems were accompanied by, or were in fact, advances on problems related to Databases, or, more generally, Information System software development.

To give some shape to my presentation, I will follow a list of topics that have been the focus of considerable research in the KR field:

1. Semantic Networks
2. Description Logics
3. Rules and exceptions to them
4. Nonmonotonic Inheritance Hierarchies
5. The Frame Problem in the Situation Calculus
6. Goals and Problem Solving

In each case, I will briefly review the relevant notions in KR¹ and, discuss how they inspired us to find new solutions to SE (and DB) problems. I will also take

¹ Given the nature of the audience, I will assume at least some familiarity with the academic field of Artificial Intelligence.

a retrospective look on the positive and negative aspects of the pre-existing KR technology we started from, and the ramifications of our research to the field of KR itself.

2 Semantic Networks

2.1 KR Foundations

Motivated by research in Natural Language Understanding and Cognitive Psychology, the most popular formalism for representing knowledge in the 1970's were semantic networks: labeled directed graphs, which in our case had an open-ended label set for nodes, but only employing a fixed, pre-determined set of possible edge labels.

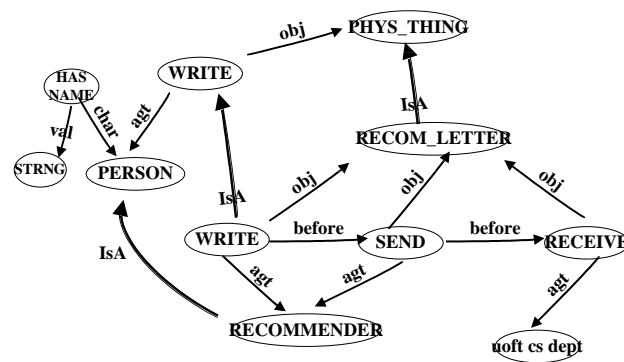


Fig. 1. Part of a Torus Semantic Network

Fig.1, based on [25], illustrates part of a semantic network graph describing the process of writing, sending and receiving recommendation letters. The picture illustrates the use of so-called “case-frames” to describe actions as nodes with participants linked to it via roles such as **agt**. The most remarked-on feature of semantic networks was the subconcept/IsA relationship that organized the nodes, and that supported a privileged kind of inference called “inheritance”. In this case, the **hasName** property of **PERSON**s is inherited by the subclass **RECOMMENDER**.

The following subsections discuss 4 projects led by John Mylopoulos in which I participated, and that applied (and often extended) semantic network ideas to problems in SE and DB. Since this work is probably best known (for an earlier survey, see [20]) the presentation below will be rather cursory.

Torus: natural language access to databases. In the context of answering natural language questions about databases, such as “Did we receive any recommendation letters for Jimbo?”, it is necessary to connect the semantic network representation of the sentence to the database schema. Supposing that there is a database table

```
RecLetterTable(Name,Source,Address,Text,DateReceived)
```

This is accomplished by connecting every column to a node in the graph (e.g., `Name` to `Applicant`, `Source` to `Recommender`, ...). The result is that part of a software system (the table) is related to the conceptual model of the domain (the semantic net). This is a general pattern we will see below, and in this special case is one of the earliest cases of representing the semantics of databases – a topic that became of considerable interest in data integration [23, 12].

Taxis: semantic data model and DB programming language. The Taxis language [26] was meant to specify or even program Information Systems at a higher, conceptual level (that of objects grouped into classes, linked by properties), which could then be compiled or transformed by hand to lower-level code dealing with standard data structures.

Taxis provided a conceptual modeling language not just for objects but also for procedures, exceptions, exception handlers, and eventually workflows [2]. So, for example, after specifying `STUDENTS`, `COURSES`, and how to `ENROLL` students into courses, one could describe `PART_TIME_STUDENTS`, `GRADUATE_COURSES`, and additional details concerning the procedure to enroll for these specialized arguments.

In fact, this led to the proposal for a methodology of *programming by step-wise refinement by specialization* [3], which prefigured similar methodologies suggested in object-oriented programming.

RML: requirements modeling. RML [19] took up the cause advocated by a few farsighted computer scientists, that *all* software (not just databases) should be connected to a description of the domain, and proposed a language for presenting such descriptions. RML can be arrived at by removing from Taxis the procedural aspects of transaction specifications (e.g., loops, assignment), introducing a temporal view of the world, and making assertions be first-class citizens that can also be grouped into classes, etc. By proposing SADT as a graphical notation that abstracts in a first pass from the details required by RML, and by offering a formal semantics for RML through translation to First Order Predicate Calculus (FOPC), together with the earlier methodology of iterative specialization, we layed out an approach for gradually going from informal to precise, logical descriptions, that could be checked for consistency.

Telos/Daida: meta-model specification and IS development environment. While Taxis pushed to the limits the use of inheritance, *meta-classes* become essential when moving to the creation of Information Systems software development environments. In the language Telos [27], just as individual **Michelle** can be an instance of class **ACCOUNTANT**, which in turn can be an instance of meta-class **PERSON_CLASS**, so property “Michelle has age 25 in 2007” (thought of as a tuple $\langle Michelle, age, 25, 2007 \rangle$) can be an instance of preproperty class “PERSONS have an age whose value is an INTEGER”, which in turn can be an instance of a meta-class that might say “This constraint must hold at all times”. Telos thus allows one to move from a situation (as in Taxis and RML) where one can say that

```
PERSON
  initial condition
    age : 0
```

with **initial condition** being *built-in to the language*, to one where arbitrary new property categories, like **initial condition**, can be defined by (well-qualified) individuals in Telos itself. Therefore Telos is useful to extend modeling languages by creating new meta-classes and allowing their semantics to be represented as formal assertions. As such, Telos meets the first desideratum of supporting Model-Driven Engineering: allowing the development of domain specific modeling languages.

In addition, the ability to treat links as first class citizens, attaching to them meta-information (e.g., “this link was asserted by x at time y for reason r”) provided the perfect substrate for building a KBSE environment. The DAIDA project [21] proposed to build an environment for developing Information Systems, and within it, an implementation of Telos, called ConceptBase, was used for a variety of tasks, including to represent requirements, design, implementations, describe tools and software processes, can capture design rationale.

2.2 KR Revisited

In the process of working on the above applications, we ended up making a number of extensions to the basic ideas of Semantic Networks, by successively extending the notion of what is an object, what properties it has, and how its classes are organized into subclasses, with property inheritance or explicit specialization:

- *procedure hierarchies*: parameters, pre- and post-conditions were treated as properties of a procedure class; instances were particular invocations. This allowed us to discuss for the first time the notion of subclasses for procedures in AI [3].
- *scripts/workflows*: Taxis scripts were Petri-nets, with condition-action rules on transitions; the places, transitions, conditions, actions on transitions, as well as local variables were all properties of script objects. The organization of workflows into subclass hierarchies echoes somewhat the recent application of description logics to workflow fragment management [18].

- *logical formulae/assertions*: in RML, free variables and subformulae are properties of an assertion class; instances of it corresponded to bindings of the free variables that make it true.
- *properties*: in Telos, properties are themselves objects, with associated fields: \langle source, identifier, destination, time-interval \rangle . For example,
 - p1 with \langle Jimbo,instanceOf,PERSON,2/feb/1992 - 31/dec/2067 \rangle
 - p2 with \langle PERSON,age,{0},all_time \rangle
 - p3 with \langle p2,instanceOf,INITIAL_PROP_CLASS,all_time \rangle
 The treatment of properties as objects (which can have their own properties) echoes the recent RDF(S) proposal for web ontology language [29], which is also based on \langle subject, predicate, object \rangle triples, that can have properties.

3 Description Logics

3.1 KR Foundations

Description Logics start with the same basic ontology as Semantic Networks: individuals, related by binary relationships (here called roles and features), and grouped into categories (called concepts). However, one distinguishing aspect of DLs is the ability to describe not just primitive/atomic concepts, but also defined ones, which are composite terms built up from simpler ones using so-called *concept and role constructors*. For example, assuming that we have atomic binary relationship `hasParents`, and atomic concepts `RICH` and `PERSON`, then the concept/noun phrase “something all of whose parents are rich” could be represented by the term **(all hasParents RICH)**, using the concept constructor **all**, which selects out those objects that are related by its first argument only to objects of the second kind. Using the role constructor **inverse**, one can define `hasChildren` as **(inverse hasParents)**, and then specify the more complex concept “a person with at least 3 children, some of whom are rich” as **PERSON and (at-least 3 hasChildren) and (some hasChildren RICH)**. Such terms, whose semantic specification closely resembles the denotation of record structure/object types in programming languages, can then be used in several kinds of assertions, most importantly *subsumption* \sqsubseteq (which resembles subtyping): One can *deduce* relationships such as **(at-least 3 hasChildren) \sqsubseteq (at-least 2 hasChildren)**; or one can assert

```
PERSON  $\sqsubseteq$  (and
  (all hasParents PERSON)
  (at-least 2 hasParents)
  (at-most 2 hasParents))
```

to state that a person has exactly two parents, both of whom are persons. Note that this is a necessary property of persons, not a definition of them. On the other hand, `MOTHER` is *defined* to be a female person (one whose `gender` feature is filled by value `'Female`) with at least one child

```
MOTHER =def (and PERSON (fills sex 'Female)
  (at-least 1 hasChildren))
```

As a result of the fact that subsumption is supported by a logic, composite concepts, hence definitions, can be automatically classified by a computer system into a subclass hierarchy.

The final important aspect of Description Logics is that there is a 20 year history of investigating the computational complexity of various combinations of concept and role constructors, because the ultimate aim is to provide implementations for languages that, unlike FOPC, come with some guarantees of termination for reasoning tasks. For example, the Classic DL [9] was proven to have a reasoner that worked in low-order polynomial time.

3.2 SE applications

One significant software engineering problem, tackled by Prem Devanbu as part of his PhD thesis with the use of DLs (and embodied in the LaSSIE Software Information System [16]) is the invisibility of the structure of software, and the need to correlate multiple views of it. Among others, software maintainers need to see operations that make sense at (i) the domain of application level (e.g., terminate a call), (ii) the architecture level (e.g., processes, layers, messages) and (iii) the code level (e.g., functions, variables, files). The following are partial examples from [14]

```

/* architecture view */
TRUNK_CONNECT_ACTION =def (and
  NETWORK_ACTION
  RESOURCE_ACTION
  (fills actor TrunkMgr)
  (all operand TRUNK)
  (all cause ATTEND_ACTION)
  ...

```

```

/* code view */
FUNCTION ⊆ (all calls FUNCTION)
  (all has_args EXPRESSION)
  ...

```

This then allows the description of specific actions, as individual objects with specific property value fillers

```

International-ISAN-Grab : (and
  ACTION
  (fills actor TrunkMgr)
  (fills operand IntlISDN-Trunk)
  ...

```

In addition, the knowledge base may contain rules such as

```

IF INTL-TRUNK-ACTION
  THEN (all cause (and ACTION (all actor ATTENDANT))))

```

representing domain knowledge that the actors of international trunk actions are attendants (a subclass of humans). These would be automatically added by

the DL reasoner to queries about INTL-TRUNK-ACTIONS, and then returned to enlighten Software Information Systems users. The SIS would also record how architecture-view slots (of an action for example) relate to code-view slots (of the function implementing it):

(operand **same-as** secondarg)

where **same-as** is a concept constructors satisfied by objects whose feature (chain) value on the left is equal to the feature (chain) value on the right.

A second SE problem we attempted to solve with the use of DLs is the specification of the *semantics of e-services* [10], whose syntax may be specified by traditional techniques. Such techniques provide a strictly code-level view of services that can be invoked on the web, usually in terms of features provided by programming languages, such as types, exceptions, etc. For example, CORBA requires the description of a service using an object oriented approach, where a class has associated attributes and methods, and the formal parameters of the methods are typed using the same identifiers as used to type the attributes. For example, the description of a service for tracking automobile ownership appears in Fig.2. Unfortunately, such specifications do not provide any aspect of

```
interface CAR{
  attrib CAR-MODEL model;
  attrib OWNER ownedBy;
  attrib MANUFACT madeBy;
  ...

  deliver( in MANUFACT src,
           in DEALER dest,
           in DATE time
           ) signals (BadDealer);
  sell(...);
  destroy(...);
}
```

Fig. 2. CORBA syntactic specification of CAR Interface with attributes and methods

the *semantics* of the service, which is necessary both for discovering appropriate services, and for verifying that they are invoked properly. In SE, such semantic specifications for methods would normally include pre/post-conditions associated with their invocation, as well as conditions under which exceptions can be expected to be raised.

A DL description of the static aspects of the CAR interface can easily be constructed from the above, by treating CORBA interfaces as DL concepts, attributes as DL features, and then asserting the appropriate subsumption:

```
CAR  $\sqsubseteq$  (and
         (the model CAR-MODELS)
```

```
(the ownedBY OWNER)
(the madeBy MANUFACT))
```

To model methods, we consider them subconcepts of a special concept **ACTION**, with parameters as features in DL. In the case of an OO specification, we need to model explicitly the implicit first argument, **this**, leading to the following:

```
DELIVER  $\sqsubseteq$  (and
  ACTION
  (the this CAR)
  (the src MANUFACT)
  (the dest DEALER)
  (the time DATE))
```

We then attach the method to the interface, by adding the term (**the deliver DELIVER**) to the necessary conditions of **CAR**. But what about the semantics of the **deliver** method? In particular, suppose its (unchecked) preconditions include that the source of the delivery must be the manufacturer of the car, its postconditions include that the owner of the car is the destination of the delivery, and that exception **BadDealer** will be signalled when the dealer is not a representative of the manufacturer. Each of these conditions can be specified as a concept in a DL:

```
//each car has associated a possible Deliver action
  (the deliver DELIVER)
//preconds include
  (madeBy same-as deliver.src)
//postconds include
  (ownedBy same-as deliver.dest)
//exception conditions for BadDealer signaling include
  not( (src overlaps dest.representativeOf) )
```

For the purposes of semantic service matching, one would then use some external program to separate the pre-/post-/exception conditions, and reason with them appropriately.

3.3 DB applications

Classic was in fact first presented at a SIGMOD database conference, and in later applications it was found that DLs provide in certain circumstances a number of advantages over traditional databases.

The special strengths of description logics became apparent when viewing them as languages used for interacting with a database-like system. Specifically, one can adopt an SQL-inspired view of a information management system as a black box with

- *create* operator to declare new identifiers, including ones associated with definitionse
- *constrain* operator to express integrity constraints on valid states of the KB
- *update* operator to manage facts about a specific world
- *inquire* operator to ask about the state of the world

Now, each of the first three operators x above involves a language L_x , while *inquire* involves two languages: one for stating questions, L_{query} , and one for expressing answers, L_{answer} . By considering what happens when each of these languages is a DL such as Classic, one gets the following insights:

- L_{create} : DLs provide the opportunity to add not just primitive but also defined concepts (“views”), to verify them for consistency, and to automatically organize them in subsumption hierarchies. In general, by providing conceptual level constructs (in contrast to the relational model, which provides more implementation-level constructs), DLs support the creation of conceptual ontologies that have been argued to be the cornerstone of efforts for *information integration*. For example, following the work of several influential papers [23, 12], the *semantics* of a relational table T :Employee(ssn, name, dept, proj) would be expressed in terms of DL ontology concepts like \mathcal{O} :Employee (treated as unary predicates), and ontology roles like \mathcal{O} :works_for (treated like binary predicates) as the following Horn formula:
 T :Employee(ssn, name, dept, proj) :-
 \mathcal{O} :Employee(x_1), \mathcal{O} :hasSsn(x_1 ,ssn), \mathcal{O} :hasName(x_1 ,name), \mathcal{O} :Department(x_2),
 \mathcal{O} :works_for(x_1 , x_2), \mathcal{O} :hasDeptNumber(x_2 ,dept), \mathcal{O} :Worksite(x_3),
 \mathcal{O} :works_on(x_1 , x_3), \mathcal{O} :hasNumber(x_3 ,proj).

- For a large class of such specifications it is possible to provide a sanity check by using a DL reasoner to verify that formula on the right is satisfiable.
- $L_{constrain}$: DLs allow necessary conditions to be stated on primitive concepts, which are essentially (complex) Integrity Constraints. (These are the first crucial ingredient in configuration management applications.)
- L_{update} : By asserting a description such as **all friends (fills gender Female)** about some individual, **DonJuan**, one is able to give properties of an indefinite number of objects – the current and any yet to be specified friends of Don Juan. This is the source of considerable expressive power for DL-based knowledge/data management, in contrast to null values in relational databases, and the second key to the success of Classic in configuration management. Note that this benefit of dealing with incomplete information accrues even when the language does not support disjunction, and has polynomial time reasoning (unlike marked null values, saay, studied in databases).
- L_{query} : DL concepts are well suited to retrieve sets of values — their instances; a benefit here is that queries can themselves be organized in subsumption hierarchies, which facilitates re-use and query refinement. (On the negative side, DL concepts cannot express even some very simple conjunctive queries [7] — the bread and butter of database research.)
- L_{answer} : When using DL concepts as part of answers, one gets the benefit of *descriptive*, rather than just enumerated, answers. For example, in response to the query “Who is female?” one could get not just Eve, but also a concept like **(fills (inverse friendOf) DonJuan)** — “the friends of Don Juan”.

3.4 KR Revisited

The development of the Classic DL was driven in part by anticipated use in KBSE. It is for this reason that the **same-as** construct was added to the language, and its use was illustrated in both the applications above. Without getting into technical details, suffice it to say that **same-as** reasoning with feature chains in Classic is efficient, and we conjecture that its absence from current web ontology languages such as OWL, will be noted once real semantic web service compositions will be attempted.

On the other hand, fairness requires us to point out that one cannot expect to encode many possible pre/post-conditions associated with methods, given the known restrictions on the expressive power of *all* DLs considered so far (see [7]).

Finally, the need to represent program control structures (iteration, conditionals, etc) as well as plans underlying software led Devanbu and Littman to introduce the Clasp DL for such notions[15]. Though CLASP was built on top of Classic, in [8] we showed that it is possible to build an *extensible architecture for normalize-and-compare DL reasoners* that allowed one to systematically add most of Classic’s constructors as well as those of Clasp to a small kernel. Such an architecture could not however handle case-based reasoning of the kind needed for disjunction. It is this aspect that current tableaux-based reasoners can handle, albeit at the cost of losing guarantees of good computational complexity.

4 General Rules and Individual Exceptions

4.1 KR Foundations

Since the early days of Semantic Networks, it was recognized that general statements asserted about concepts (e.g., “all birds fly”), may need to allow exceptional instances (e.g., “Tweety is a bird. Tweety does not fly (because she is afraid of heights, or has a broken wing, or simply does not want to).”) Such contradictions between formulas of the form $(\forall x)P(x)$ and $\neg P(a)$ were the first “crack” in the façade of statements like “*Semantic Nets are just an alternative notation to First Order Logic*”, and eventually led to the development of so-called *non-monotonic logics*, where one could make default assertions, such as “For all birds, deduce they can fly, unless this leads to a contradiction”. For example, in Reiter’s notation this would be written as $\frac{Bird(x):Fly(x)}{Fly(x)}$.

4.2 DB applications

In databases, *integrity constraints* are general rules that are not intended to deduce new information, but instead to detect inconsistencies after database updates. When dealing with the natural world, integrity constraints are almost always over-generalizations. The spirit of freedom implied by the KR modeling above suggests that even in databases one would want to allow the co-existence of general constraints such as “An employee’s salary is over 1000, but less than

their manager's”

$$\forall e. e \in EMP \implies (e.salary > 1000) \wedge (e.salary < e.manager.salary)$$

and occasional exceptions, such as

```
calvin.salary = 20000
calvin.manager = hobbes
hobbes.salary = 15000
```

where the problem might be blamed on the fact that `hobbes` is only temporarily assigned to be `calvin`'s manager. (It could also have been that `hobbes` earns too little, or `calvin` too much, or some combination of facts.) But this leads to a difficulty: once even *one* exception is allowed to persist, this IC will always evaluate to false, and can no longer detect errors in future updates (e.g., when `judy`'s salary is changed to 900, which is “a new reason” for it to be false). Therefore, we want to modify the constraint to restore its error detection capability. In [4], I propose that one first rewrite the constraint in FOPC without function symbols:

$$\begin{aligned} \forall e, se. (e \in EMP \wedge sal(e, se)) \implies (se > 1000) \wedge \\ \forall me, sm. (mgr(e, me) \wedge sal(me, sm)) \implies se < me \end{aligned}$$

Then, rather than going for the obvious

$$\begin{aligned} \forall e. (e \neq hobbes \wedge e \in EMP \wedge sal(e, se)) \implies (se > 1000) \wedge \\ \forall me, sm. (mgr(e, me) \wedge sal(me, sm)) \implies se < me \end{aligned}$$

(which is not good enough because in this constraint there are two conditions that are being checked at the same time), I suggest to use the the more subtle formula

$$\begin{aligned} \forall e. (e \in EMP \wedge sal(e, se)) \implies (se > 1000) \wedge \\ ([mgr(e, me) \wedge \neg(e = calvin \wedge me = hobbes)] \wedge sal(me, sm)) \implies se < me \end{aligned}$$

Interestingly, this corresponds to a *model theoretic* specification: minimally mutilate all models of the original IC, so that the exceptional fact — $mgr(calvin, hobbes)$ in this case, is counter-factual. In other words, take the set M of all models of the original IC, and modify each member μ so that if $mgr(calvin, hobbes)$ was true in it, it will no longer be so, and if it was false, than it will now be true; the resulting set of structures are exactly the models of the modified formula above! (Note that the form of the modified formula depends on which facts are blamed for the exception.) In fact, one can prove that this holds in general: there is a provably correct syntatic transformation on any FOPC formula which corresponds to the minimal mutilation. A nice consequence of this model-theoretic characterization is that the actual syntactic form of the IC does not matter.

The paper [4] also proposed that persisting exceptional facts, blamed by users for constraint violations, be guarded by programming language exceptions, which would be raised whenever database queries or transactions would touch them, because they would likely require special handling that cannot be anticipated ahead of time. (The proposal allowed for on-line, dynamic exception handling by users.)

4.3 SE applications

Inspired in part by the general idea above, though not the technical details, Robert Balzer wrote a well-known paper [1] (winner of the ICSE Most Influential Paper Award) that investigates the occurrence of transient exceptions in software systems, including software development environments, where problems of inconsistency are likely to arise, especially in asynchronous distributed environments. In contrast to persistent exceptional data, problems here are seen to be due to (temporarily) inaccurate data or partial updates.

The solution is based on guards (“pollution markers”) being placed alongside data that cause constraint violations. The technical solution relies on automatically softening arbitrary FOPC constraints by inserting/removing guards or rules which derive guards.

A pollution marker for constraint $\forall \mathbf{x}.\Psi(\mathbf{x})$ records the n-tuple of values \mathbf{a} instantiating the leading quantified variables for which $\Psi(\mathbf{a})$ is false. Such a pollution marker is added to the database as a new atom $\Psi_{Pollution}(\mathbf{a})$. In one implementation of this idea, markers are added and removed by pairs of database triggers. In a second one, guards are derived from a modified version of the constraint, of the form $\forall x.\Psi(x) \oplus \Psi_{Pollution}(x)$, where \oplus is “exclusive-or”. Since inconsistencies are considered to be undesirable, the paper also investigates automatic repair techniques, which remove inconsistencies (i.e., what we would call automatic exception handlers for the original constraints).

Note that both the SE and DB approaches to living with inconsistency do so by modifying the original constraint Ψ . It may be revealing to briefly consider the technical differences between the two approaches. The one in [4] relies on external agents to identify one or more literals to be “blamed” for the inconsistency, and then minimally modifies the formula Ψ to accommodate these. (If the choice of facts to be blamed is wrong, then the modified constraint will still not be consistent with the database!) The second one automatically chooses to blame all the individuals involved in the constraint, thus ensuring the consistency of the new constraint with the data, but at the cost of painting a picture with a broad brush: For example, for the constraint

$$\forall x, m, f. Person(x) \implies (hasMother(x, m) \implies \neg Working(m)) \wedge (hasFather(x, f) \implies Working(f))$$

when a violation is detected for $x=jay$, the pollution marker for (jay, jay’s mother, jay’s father) cannot differentiate between the problem being that the mother works, the father doesn’t or both. In fact, pollution markers guard/soften constraints, rather than individual facts that need to be treated circumspectly because they may be incorrect.

4.4 KR Revisited

As we have seen, neither the DB nor the SE applications of exceptional individuals adopted the default logic solution proposed in KR, because the general rules in our special domains are for *constraint checking*, rather than *deduction*, and

default rules would not raise any red flags whenever they were contradicted — they would simply “back off”, even if this was an error. Moreover, the syntactic solutions proposed in both case fell within the well-understood framework of FOPC.

As an interesting KR aside, my student, Mukesh Dalal used the idea of minimal mutilation of models (introduced to accommodate blamed exceptional facts) to obtain an operator $update(kb, u)$ for revising beliefs in propositional knowledge-bases [13]. This was (almost) the first belief revision operator that was insensitive to the syntactic form of the theory kb or the update u . (To get the models of $update(kb, u)$, it iteratively mutilated single atoms of kb models till at least one model consistent with u was found.)

5 Subclass Hierarchies and Nonstrict Inheritance

5.1 KR Foundations

Another famous conundrum for logicians in the KR community was the problem of representing information such as *All birds fly*, *All penguins are birds*, and *No penguins fly*. These statements, though not inconsistent by themselves, imply that there can be no penguins, which is not the original intent of the modelers: they actually want that when a Penguin instance is found (e.g, opus), then from $Penguin(opus)$ one deduces $\neg Fly(opus)$, and not deduce $Fly(opus)$.² Such theories can again be expressed using default logic, but because of their ubiquity, they were studied under the special rubric of “default inheritance networks” [31, 17], consisting of concepts/nodes connected only by IsA and IsNotA edges. In the above case, from {Penguin IsA Bird, Bird IsA Fly, Penguin IsNotA Fly}, one obtains the desired inference using a shortest path heuristic. In more complex cases, when the graph is not a tree, one needs more careful algorithms.

5.2 SE/DB applications

One of the standard features of *object-oriented* data models, as well as OO programming languages, is *inheritance*: the ability to avoid having to repeat restrictions on super-classes, to all their subclasses. For example, having stated that

```
PERSONS with
  age : 0..120
```

one does not have to repeat the constraint that ages are integers between 0 and 120, for any subclasses, such as ATHLETES, CANADIANS, etc. It was noted already

² The difference between this problem, and the one explored in the previous section, is not just that in this case we have exceptional subclasses, while in that case we had exceptional individuals; as our example showed, it was the atom $mgr(calvin, hobbes)$ that was exceptional, not just an individual.

in the Taxis language that frequently it is desirable to further specialize the constraints expressed for certain subclasses. For example, for subclass `EMPLOYEE` of `PERSON` one may want `age: 16..65`.

This provides several software engineering advantages:

1. since repetition is a potential source of errors, inheritance avoids them;
2. when changes are made to a constraint (e.g., longevity of humans increases in general), the change on `PERSON` is also inherited to all subclasses, thus avoiding the problem of forgetting to make changes in some places;
3. when constraints are stated on a subclass, the system can verify that they are more restrictive than those inherited, thus warning the programmer in case of an error.

As in AI, there are however cases where the constraint specified on the class is too strict. We have already seen in the previous section a technique for dealing with individual exceptions. But what if an entire subclass is exceptional? For example, one may want to say that `GEORGIANS` have a longer life span in general, by allowing their age to range from 0 to 135 say.

```
GEORGIAN is a PERSON with
  age : 0..135
```

As noted above, this would normally lead to a compiler error. To avoid this, one could generalize the constraint on `PERSON` to allow `age: 0..135`, but then the constraint would be less useful for detecting errors for all instances of `PERSON`, or we would have to repeat the constraint `age : 0..120` for all other immediate subclasses of `PERSON`, such as `ATHLETE`, `CANADIAN`, etc., which used to inherit it, thus impeding advantages 1 and 2 above. The other alternative, “default inheritance” as defined in AI, eliminates advantage 3.

For this reason, we proposed [5] that programmers express conflicting specialization with a syntax like

```
GEORGIAN is a PERSON with
  age : 0..135 excuses age on PERSON
```

which explicitly acknowledges the improper specialization, and proposes to override it. Rather than using a default logic to describe it, it turns out to be possible to capture the desired semantics using FOPC:

$$\begin{aligned} \forall x.GEORGIAN(x) &\implies (0 \leq age(x) \leq 135) \\ \forall x.PERSON(x) &\implies (0 \leq age(x) \leq 120) \vee \\ &[GEORGIAN(x) \wedge (0 \leq age(x) \leq 135)] \end{aligned}$$

Excuses provide a way of explicitly adjudicating between contradictory constraints. This can in fact be used to deal not just with problems due to inheritance, but also the case of individuals belonging to multiple classes. (See [5] for details.)

It is well known that subclasses provide an additional opportunity for improved software engineering of OO systems: a program or database query that

is type-correct when it involves objects of class C is type correct for objects of type D, for all subclasses D of C (the famous substitution principle), *as long as subclasses are also subtypes*.

In the presence of exceptional subclasses this is no longer the case, as illustrated by the following example, involving a large company with many subsidiaries, some of which (say web-site designers) are located outside the USA. For such a company the address and phone number of employees should be modeled as

```
EMPLOYEE with
  phone: [areaCode :INT(3);
          localPhone :INT(7)]
  name: ...
```

```
WEBSITE_DESIGNER is a EMPLOYEE with
  phone: [countryCode : INT(2);
          cityCode :INT(2);
          localPhone :INT(8)]
  name: ...
```

where we use the notation [p:T; ...] to describe, as usual, record types with field p of type T.

Now, the problem is that while the following query looking for Italian website designers

```
select x from WEBSITE_DESIGNER where x.phone.countryCode =33
```

is type correct, the next query, looking for certain New Jersey employees in general

```
select x from EMPLOYEE where x.phone.areaCode = 732
```

is not so, because some employees do not have `areaCode`. But in OODB, the set of instances of a class is available at run-time, and so testing for membership in this class extent is allowed. So, suppose we introduce a `WHEN-THEN-ELSE` construct, resembling `IF-THEN-ELSE`, which can be used to test for class membership and hence guard against the evaluation of inappropriate attributes:

```
select x from EMPLOYEE where
  when (x in WEBSITE_DESIGNER)
  then false
  else x.phone.areaCode = 732
```

It is possible to develop a type theory that allows type checking even in the presence of non-strict inheritance. In particular, we introduce record structures with union types and “guards” $C\Delta$ on record fields [6]:

```
[phone: [areaCode:INT; localPhone:INT]
  ⊔
  WEBSITE_DESIGNERΔ[countryCode:INT; cityCode:INT; localPhone:INT];
  ...
```

]

The language type inference rules then incorporate conditional tests in the evaluation of subexpressions,

$$\begin{array}{l}
 \mathcal{T}, \Sigma \vdash e : ENTITY, \\
 \mathcal{T}, \Sigma, e : C \vdash g1 : s, \\
 \mathcal{T}, \Sigma, e \notin C \vdash g2 : s, \\
 \hline
 \mathcal{T}, \Sigma \vdash (\text{when } (e \text{ in } C) \text{ then } g1 \text{ else } g2) : s
 \end{array}$$

and correlate guards in types with class membership, e.g.,

$$\begin{array}{l}
 \mathcal{T}, \Sigma \vdash e : C, \\
 \mathcal{T}, \Sigma \vdash e.p : s \\
 \hline
 \mathcal{T}, \Sigma \vdash e : [p : C \Delta s]
 \end{array}$$

Note that in order to achieve completeness of reasoning, we have had to introduce judgments about *non-membership* in classes ($e \notin C$), and in fact these play an essential role.

5.3 KR Revisited

This is another case when, although the real world phenomena observed by both AI and SE are ubiquitous, the AI solutions are not suitable for SE and DB applications, because of different goals: inference in AI, and safe software development practice in SE/DB. Moreover, the theory of subclasses with exceptions that we developed is based once again on FOPC, and, perhaps surprisingly, includes rules for subsumption reasoning (see [5] for *subtyping* rules), familiar in Description Logics, but not in default inheritance hierarchies.

Once again, the solutions we pursued are based on standard FOPC rather than esoteric non-monotonic logics, which normally have poor computational properties.

6 The Frame Problem in Specifying Actions

6.1 KR Foundations

The *situation calculus* is one of the earliest and most elegant logical formulations of the problem of reasoning about actions and their effects. As all such theories, the situation calculus needs to propose a way to distinguish assertions describing the state of the world before and after actions take place. Rather than using an additional temporal argument for each predicate/function, the situation calculus reifies states, and uses expressions to denote new states resulting from actions carried out in prior states. For example, to say that *Anna* is enrolled

in *math101* (normally written as *enrolledIn(Anna, math101)*) after the operation *signUp(Anna, math101)* was carried out in state *s0*, we assert the atomic formula:

$$\text{enrolledIn}(\text{Anna}, \text{math101}, \text{do}(\text{signUp}(\text{Anna}, \text{math101}), \text{s0}))$$

and to say that class size increased by one as a result of the same action, we write

$$\text{size}(\text{math101}, \text{do}(\text{signUp}(\text{Anna}, \text{math101}), \text{s0})) = \text{size}(\text{math101}, \text{s0}) + 1$$

One of the early problems noted in representing and reasoning about actions, whether in the situation calculus or other notations, is that in describing the effect of actions, it seems natural to only describe what has changed, but then no formal reasoning is possible about some completely unrelated predicate, say *presidentOf(Dubya, usa, do(signUp(Anna, math101), s0))* unless there is also an explicit axiom relating this to *presidentOf(Dubya, usa, s0)*. At the very least, this causes an $O(n*m)$ increase in the number of axioms, where *n* is the number of actions, and *m* is the number of “fluents” (predicates and functions that may change with state).

An elegant solution to this problem was proposed by Ray Reiter. Essentially, rather than focusing individually on each action, it focuses in turn on each fluent, and what actions might have changed it. For example, one might have

$$\neg \text{enrolledIn}(x, y, s) \wedge \text{enrolledIn}(x, y, \text{do}(a, s)) \implies a = \text{signUp}(x, y)$$

to indicate that only the *signUp(-, -)* operation adds new tuples to *enrolledIn*. For a more complicated example, we might have that if *x* drops a course *c*, then they also drop the co-requisites of *c* that they are signed up for.

$$\begin{aligned} \text{enrolledIn}(x, y, s) \wedge \neg \text{enrolledIn}(x, y, \text{do}(a, s)) \implies a = \text{drop}(x, w) \wedge \\ (w = y \vee \text{coRequisiteOf}(w, y)) \end{aligned}$$

The secret of solving the frame problem is then to make the above the *only* reasons why the predicate might change:

$$\begin{aligned} \text{enrolledIn}(x, y, s) \wedge \neg \text{enrolledIn}(x, y, t) \iff \\ t = \text{do}(\text{drop}(x, w), s) \wedge (w = y \vee \text{coRequisiteOf}(w, y)) \\ \vee s = \text{do}(\text{signUp}(x, y, t)). \end{aligned}$$

The representation of actions must also include statements about their preconditions and postconditions being satisfied in the appropriate states. This is done using a predicate *Occur*, as in

$$\begin{aligned} \text{Occur}(\text{signUp}(st, crs), s) \implies \\ \neg \text{EnrolledIn}(st, crs, s) \wedge \text{size}(crs, s) < \text{classLimit}(crs, s) \end{aligned}$$

6.2 SE applications

The frame problem also arises in the formal specification of procedures using Floyd-Hoare-style pre-/post-conditions. For example, one might describe the enrolling method on class COURSE as

```
SIGN_UP(st : STUDENT, crs : COURSE)
pre
  size(crs) < classLimit(crs) ∧ crs ∉ enrolledIn(st)
post
  enrolledIn' = enrolledIn ∪ {(st, crs)} ∧
  size' = size + { crs ↦ 1 + size(crs) }
```

while dropping a course is specified as

```
DROP(st : STUDENT, crs : COURSE)
pre
  (st, crs) ∈ enrolledIn
post
  enrolledIn' = enrolledIn - {(st, crs)} ∧
  size' = size ⊖ { crs ↦ size(crs) - 1 }
```

Here we use the standard convention that predicates evaluated in the ending state of the procedure are primed versions of the names used in the initial state.

It is usual to associate with classes certain invariant assertions I , such as $size(crs) \leq classLimit(crs)$, and then a standard proof obligation for every method is to show

$$I \wedge pre \wedge post \implies primed(I)$$

Of course, since nothing is said above in the post condition of DROP() about $classLimit'$, the frame problem strikes again and such a proof is not possible in this case. One would have had to explicitly assert for both the above methods that $classLimit' = classLimit$.

All specification languages recognize this difficulty, and provide some mechanism for saying “...and nothing else changes”. So, for example, Z provides an operator Ξ , such that ΞW expresses that the primed and unprimed identifiers in W are identical. Larch and other specification languages have a construct “modifies at most M ”, which allows the system to implicitly assert that all other names not in W will not change.

In [11], we provide a number of example showing that these solutions are not adequate for program specifications, because the explicit/implicit frame assertions contradict each other when we try to compose local specifications to create larger ones. For example, it would seem natural to define $transfer(st, from, to)$ as the conjunction of the specifications for $drop(st, from)$ and $signUp(st, to)$. But the frame assertions of the two procedures conflict, resulting in inconsistency. This is particularly problematic for object-oriented specification languages, where one would like to specialize the method on a subclass, but inher-

itance of the super-class' specification is just conjunction, and hence leads to contradiction.

We therefore adapted Reiter's proposal to this notation as follows: The *Occur* predicate is used to capture the usual pre and post conditions, but with the primed notation, as in

$$\begin{aligned} Occur(drop_{COURSE}(st, crs)) \implies \\ EnrolledIn(st, crs) \wedge \\ size'(crs) = size(crs) - 1 \wedge \neg EnrolledIn'(st, crs) \end{aligned}$$

We now need two explanation closure axioms for *EnrolledIn*:

$$\neg EnrolledIn(x, y) \wedge EnrolledIn'(x, y) \wedge Occur(\alpha) \implies (\alpha = signUp(x, y))$$

$$\begin{aligned} EnrolledIn(x, y) \wedge \neg EnrolledIn'(x, y) \wedge Occur(\alpha) \implies \\ (\alpha = drop_{COURSE}(x, y)) \\ \vee (\alpha = drop_{COURSE_WITH_COREQ}(x, w) \wedge (w = y \vee hasCoreqs(y, w))) \end{aligned}$$

and one for each function, *classLimit* and *size*. Finally, note that the original proof obligations now become showing that

$$I \wedge Occur(\alpha) \implies primed(I)$$

for every possible action α .

6.3 KR Revisited

Although once again we did not adopt the fully general AI solution to the frame problem, based on circumscription, in this case the FOPC-based solution provided by Reiter and others turned out to be entirely adequate for our task: all we had to do is simplify the Situation Calculus notation, eliminating fluents in favor of the primed predicate notation, since there was no need to reason about sequences of actions. This is then one of the few cases where we were able to take off-the-shelf KR technology and apply it directly to SE problems.

7 Goals and problem-solving

This section is included to show that we are even at this point pursuing similar tactics in solving Software Engineering and Database problems.

7.1 KR Foundations

In the early days, AI was seen as mainly concerned with search through a space of alternatives. A particular version of this, problem (goal) reduction search, recursively broke up more difficult goals into simpler ones, till these could be achieved by operators/actions available to the system. The above process naturally leads to an AND/OR graph of alternative decompositions, which is usually

searched heuristically for a solution. GPS [28], one of the classic early AI system could be viewed as implementing just such a model.

Herbert Simon also noted that certain kinds of goals (e.g., to be happy) cannot be said to be definitively achieved/satisfied, and argued that a weaker notion of *satisficing*, was necessary for such “soft goals”.

7.2 SE applications

In the past decades, research in Requirements Engineering for software has undergone a revolution, whereby the standard functional specification stage is now preceded by a new phase of *early requirements*, dealing with the intentions of the agents and organizations in the environment where the software is to be used. Because of its focus on goals, and how they are to be achieved, this is known as Goal-Oriented Requirements Engineering (GORE). We point the reader to van Lamsweerde’s excellent review of the field [32], and note that AND/OR decomposition of goals, followed by specification of actions that achieve leaf-level goals plays a central role in this enterprise.

A second, important aspects of GORE is dealing with so-called non-functional goals, such as efficiency, accuracy, etc. which are exactly the kind of soft goals that Simon suggested should be satisfied.

For example, GORE methodology suggests that we start with various stakeholders, and their general hard and soft goals; decompose these into subgoals using AND/OR graphs; then apply means/ends analysis to find tasks that achieve them. See Fig. 3 for a small example, which uses the i^* notation [33]. One novelty is that, in diagrams, we use *contribution edges* labeled with + or - (or even ++ and --) to indicate how goals influence each other. Thus, in Fig.3, the softgoal (peanut-shape) of “Evaluating students accurately” contributes positively towards the “Do a good job” softgoal, but performing the task (hexagon) of giving a single exam contributes negatively towards accurate evaluation. One important advantage of goal-oriented approaches is that they provide consideration of design alternatives, and traceability for decisions based on such contribution dependencies.

7.3 DB applications

The field of database specification, on the other hand, has remained pretty-well unchanged since the mid-70’s: one still starts by constructing a conceptual schema as an ER or maybe UML diagram. In recent joint work with Jiang et al. [22], we have started to look at a *goal-oriented approach to database schema design*. Assuming that the example above would require a solution with a database of students, grades, etc., the methodology suggests analyzing the textual description of goals, and the participants in the tasks, to obtain a list of *relevant concepts*, which is then organized into a *domain model* (expressed in some conceptual modeling language); its purpose is to provide a shared understanding of the domain for database designers and end-users. The *conceptual schema* of

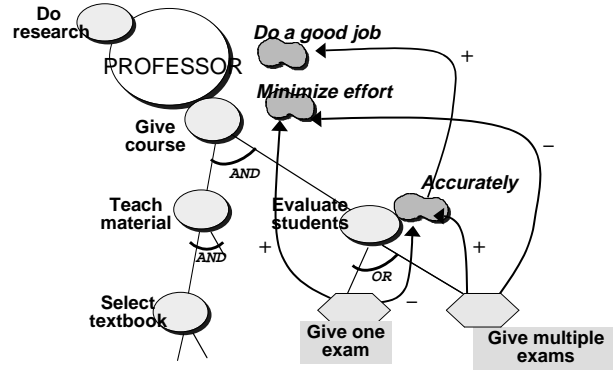


Fig. 3. A Goal Model in i^*

the database is then derived from this by addressing a list of questions concerning issues such as persistence, time, data quality, etc. For each such category, we have a number of alternative schema manipulation operators which can be applied to derive the final conceptual schema from the domain model. We are also currently investigating the influence of *data quality* considerations on the schema design.

7.4 KR Revisited

Although it is clear how to formalize AND/OR goal decomposition even in Horn propositional logic, notions like softgoals, their satisficing, and contribution edges would seem to be inherently “soft” — hard to reason with. Sebastiani et al [30] however show how to formalize even this aspect: Since there could be conflicting evidence concerning any goal g , the secret is to replace proposition g by 4 propositions

fully_satisfied_g, partially_satisfied_g, partially_denied_g, fully_denied_g.

The reasoning task is then set up by, for example, replacing an edge $g \xrightarrow{+} h$ by axiom

$$partially_satisfied_g \implies partially_denied_h$$

while edge $g \xrightarrow{-} h$ also adds axiom

$$fully_satisfied_g \implies fully_denied_h$$

By using a suitable extension of this set of axioms, and a min-sat solver, it is then possible to find minimal sets of “input/bottom” goals that guarantee desired top-level goals.

8 Conclusions

I have briefly reviewed a sample of software engineering/database-inspired projects which had connections to KR&R topics:

KR topic	SE application	DB application
semantic networks	requirements modeling; methodology of iterative refinement by specialization; KBSE environment for developing Info Systems;	semantic data model; database programming; meta-model;
description logics	Software Information Systems; e-service semantic specification	managing incomplete data (e.g., design databases)
rules with exceptions	transient inconsistent states in software kbs	persistent exceptions to database integrity constraints
inheritance networks with defaults	supporting exceptional subclasses; type checking in their presence;	application to object-oriented databases;
frame problem in situation calculus	solution to frame problem in object-oriented program specification languages;	
goals and problem solving	GORE	Goal-oriented database schema design

In retrospect, in several of these cases, but by no means all, we started with KR technology (semantic networks, description logics, situation calculus) and looked for applications, usually in the field of database but sometimes directly in Software Engineering. Interestingly, it often turned out that the database topics led to software engineering issues (methodologies, CASE tools). Moreover, except for Reiter’s solution to the frame problem, the actual solutions either did not use the AI solutions to the similar problems (e.g., default logics for exceptions), or we had to extend them considerably, leading to sufficiently novel ideas that they were published in the KR literature.

Acknowledgments

I am most grateful to all my collaborators over the years for making this such an enjoyable adventure. I am particularly grateful to John Mylopoulos and Ron Brachman, who are not just exemplary scientists but wonderful human beings, whose support has made much of this work possible. I owe Enrico Franconi thanks for suggesting “KR meets DB” as the topic of a prior talk, and Rick Salay for excellent comments on short notice.

This work was supported in part by the U.S. DHS under ONR grant N00014-07-1-0150.

References

1. Robert Balzer, "Tolerating Inconsistency", *ICSE 1991*: 158-165
2. J.L. Barron, "Dialogue and Process Design for Interactive Information Systems using Taxis", *Proc. ACM SIGOA*: 12-20, Philadelphia, June 1982
3. Alexander Borgida: "On the Definition of Specialization Hierarchies for Procedures." *IJCAI 1981*: 254-256
4. Alexander Borgida: "Language Features for Flexible Handling of Exceptions in Information Systems." *ACM Trans. Database Syst.* 10(4): 565-603 (1985)
5. Alexander Borgida: "Modeling Class Hierarchies with Contradictions." *Proc. ACM SIGMOD Conference 1988*: 434-443
6. Alexander Borgida: "Type Systems for Querying Class Hierarchies with Non-strict Inheritance." *Proc. ACM PODS 1989*: 394-400
7. Alexander Borgida: "On the Relative Expressiveness of Description Logics and Predicate Logics." *Artif. Intell.* 82(1-2): 353-367 (1996)
8. Alexander Borgida: "Extensible Knowledge Representation: the Case of Description Reasoners." *J. Artif. Intell. Res. (JAIR)* 10: 399-434 (1999)
9. Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, Lori Alperin Resnick: "CLASSIC: A Structural Data Model for Objects.", *Proc. ACM SIGMOD Conference 1989*: 58-67
10. Alexander Borgida, Premkumar T. Devanbu: "Adding more "DL" to IDL: Towards More Knowledgeable Component Inter-Operability." *ICSE 1999*: 378-387
11. Alexander Borgida, John Mylopoulos, Raymond Reiter: "On the Frame Problem in Procedure Specifications." *IEEE Trans. Software Eng.* 21(10): 785-798 (1995)
12. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Daniele Nardi, Riccardo Rosati: "Data Integration in Data Warehousing." *Int. J. Cooperative Inf. Syst.* 10(3): 237-271 (2001)
13. Mukesh Dalal: "Investigations into a Theory of Knowledge Base Revision." *AAAI 1988*: 475-479
14. Premkumar Devanbu, Mark Jones: "The Use of Description Logics in KBSE Systems", *ACM. Trans. on Software Engineering Methodology* 6(2) 141-172 (1997)
15. Premkumar T. Devanbu, Diane J. Litman: "Taxonomic Plan Reasoning." *Artif. Intell.* 84(1-2): 1-35 (1996)
16. Premkumar T. Devanbu, Ronald J. Brachman, Peter G. Selfridge: "LaSSIE: A Knowledge-Based Software Information System". *Commun. ACM* 34(5): 34-49 (1991)
17. David W. Etherington, Raymond Reiter: "On Inheritance Hierarchies With Exceptions." *AAAI 1983*: 104-108
18. Antoon Goderis, Ulrike Sattler, Carole A. Goble: "Applying Description Logics for Workflow Reuse and Repurposing." *Description Logics 2005*
19. Sol J. Greenspan, Alexander Borgida, John Mylopoulos: "A requirements modeling language and its logic." *Information Systems* 11(1): 9-23 (1986)
20. Sol J. Greenspan, John Mylopoulos, Alexander Borgida: "On Formal Requirements Modeling Languages: RML Revisited." *ICSE 1994*: 135-147
21. Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou: "DAIDA: An Environment for Evolving Information Systems." *ACM Trans. Information Systems* 10(1): 1-50 (1992)
22. Lei Jiang, Thodoros Topaloglou, Alexander Borgida, John Mylopoulos: "Incorporating Goal Analysis in Database Design: A Case Study from Biological Data Management." *RE 2006*: 196-204

23. Alon Y. Levy, Anand Rajaraman, Joann J. Ordille: "Querying Heterogeneous Information Sources Using Source Descriptions." *VLDB 1996*: 251-262
24. John Mylopoulos, Alexander Borgida, P. Cohen, Nick Roussopoulos, John K. Tsotsos, Harry K. T. Wong: TORUS - A Natural Language Understanding System For Data Management. *IJCAI 1975*: 414-421
25. John Mylopoulos, P. Cohen, Alexander Borgida, L. Sugar: "Semantic Networks and the Generation of Context." *IJCAI 1975*: 134-142
26. John Mylopoulos, Philip A. Bernstein, Harry K. T. Wong: "A Language Facility for Designing Database-Intensive Applications." *SIGMOD 1978 (Abstract)*. *ACM Trans. Database Syst.* 5(2): 185-207 (1980)
27. John Mylopoulos, Alexander Borgida, Matthias Jarke, Manolis Koubarakis: "Telos: Representing Knowledge About Information Systems." *ACM Trans. Inf. Syst.* 8(4): 325-362 (1990)
28. Allen Newell, Herbert A. Simon: "GPS, a program that simulates human thought", *Computation & Intelligence: collected readings*:415 - 428. MIT Press, 1995 (Original RAND Report: 1961)
29. RDF Vocabulary Description Language 1.0: RDF Schema, <http://www.w3.org/TR/rdf-schema/>
30. Roberto Sebastiani, Paolo Giorgini, John Mylopoulos: "Simple and Minimum-Cost Satisfiability for Goal Models." *CAISE 2004*: 20-35
31. David S. Touretzky, John F. Horty, Richmond H. Thomason: "A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems." *IJCAI 1987*: 476-482
32. Axel van Lamsweerde: "Requirements engineering in the year 00: a research perspective." *ICSE 2000*: 5-19
33. Eric S. K. Yu, John Mylopoulos: "From E-R to A-R — Modelling Strategic Actor Relationships for Business Process Reengineering." *Int. J. Cooperative Inf. Syst.* 4(2-3): 125-144 (1995)