Database Application Development as an Object Modeling Activity

**Manfred A. Jeusfeld*, Michael Mertikas+, Ingrid Wetzel+,
Matthias Jarke*, Joachim W. Schmidt+**

* FORWISS Research Center, Universität Passau, P.O. Box 2540, 8390 Passau, W. Germany
+ Fachbereich Informatik, Universität Hamburg, Schlüterstr. 70, 2000 Hamburg 13, W. Germany

Abstract. The DAIDA project has made an attempt of formally defining and relating declarative and computational entities considered relevant to the process of database application development. Such entities range from object-oriented specifications to executable modules of database programs. To bridge the gap between semantics and computation, they also include abstract machine-based formal specifications and transformational theories. In an second contribution, selected characteristics of such entities and relationships are modeled uniformly in a software information system. Emphasis is placed on those properties that may become relevant when applications have to be modified or adjusted. Besides discussing the interaction of these aspects of the DAIDA methodology, the paper outlines an operational project prototype and reports first experiences.

1 Introduction

A short historical consideration of the database area points out that the first data models, which were supposed to satisfy arising database needs in computing environments, were not sufficient to make the distinction between *semantic* and *computational* aspects of database applications [BM86]. Programmers tried to come up with database application development by implicit and naive modeling of the application area in terms of available data types -- trees, networks, relations -- and to capture all application semantics within computational units -- transactions, programs. Research effort focused on providing the database programming community with tools -- code generators, language-sensitive editors, interfaces, etc. This certainly increased programming productivity but did not solve the well-known inefficiencies of database software development.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

The development of the relational model, with its well defined abstraction features, was probably the starting point for suspecting that there is something more to data models than just symbol manipulation. This suspicion caused the exploitation of *semantic data models* as a first dimension of activities towards better database application engineering [HK87]. These models emphasized the specification of application semantics -- with their own environment, tools, support, etc. -- but caused a divergence between technologies dealing with semantic and computational aspects. Database applications can now be thought of as *resting on two pillars without a bridge* in between. This statement is well illustrated by the current debate on semantic-oriented knowledge representations versus computationally motivated object-oriented databases. Generic attempts to combine both, for example deductive databases which offer a declarative as well as a procedural interpretation, seem to cover only a limited number of application domains.

For more general database programming tasks, the two pillars are too far apart to be easily connected by any kind of automated translation technology. The conceptual distance between their goals and functionalities -- expressive power on the one side, computational efficiency on the other -- therefore adds an additional factor of complexity to database application development.

The second dimension of research activities related to database application development has been attacking this additional complexity by developing *bridging technologies* -- expert systems guiding implementations, mapping assistants supporting transformation of one formalism to another, or proof assistants supporting refinement of specifications. Extending the similarities between database application development and the bridge construction we may assume that *adding a third pillar in the right place* interprets this activity in a good way.

Work on DAIDA (= Development Assistance for Interactive Database Applications) was supported in part by the European Commission under ESPRIT Contract 892, including the software houses BIM, Belgium; GFI, France; SCS, Germany; the FORTH Computer Research Center, Greece; and the Universities of Frankfurt/Hamburg and Passau, Germany.

In section 3, we present part of the second dimension activities: how to bridge the semantic and the implementation layer. We start from predicative object-oriented descriptions, a technology supporting attractive abstraction principles and concise notation. We transform these specifications into an intermediate structure of Abstract Machines and proceed with formal refinement down to efficient transaction-oriented database programs.

In section 4, we enter the third dimension of activities by explaining how to describe abstractly system components and their dependencies; also, how to make use of this information. Section 5 describes the actual tools the DAIDA environment offers for mapping support and software information management, and presents a more general process model based on this experience.

2 Two DAIDA Views of a Data-Intensive Application

Taking for granted the necessity of a semantic layer addressing expressive power and reasoning capabilities and of a computational layer addressing computational efficiency, we briefly describe the languages TDL and DBPL which are appropriate for the semantic and the implementation layers. Both languages have been defined and used in the DAIDA project [BMMS88, SEM88].

Extending ideas from the Taxis project at the University of Toronto [MBW80], the Taxis Design Language TDL [BMMS88] introduces a predicative and object-oriented style of specifications. TDL supports object manipulation through instantiation in a data class called `EntityClass`.

Objects have intrinsic identity. This supports the ability to distinguish object identity from object equality, to share objects among other objects, to modify objects (which is not the same as deletion and insertion in value- based formalisms), and to support referential constraints naturally.

The structure of values in TDL can be complex. Simple values are modeled as instances of `BasicClass` and `EnumeratedClass`. Values built as tuples of other values are modeled as instances of `AggregateClass`, and their identity is specified by their components, which are named by attributes. Entities in the application domain are modeled as object instances of `EntityClass`; specific integrity constraints are assigned to their attributes by means of attribute categories and range constraints.

Operations are modeled as instances of `TransactionClass`, where the input/ output of a transaction and its actions is described by appropriate attributes and logical formulas.

Most experience has been gained with design assistance for database structuring. Much less work has addressed procedural aspects of information systems development and their interaction with data design [BR84, EGL84, SS89]. This is despite the fact that object-oriented databases emphasize the integration, even encapsulation, of structural and procedural components [SSE87].

Finally, the third dimension of activities is concerned with *the maintenance monster*, something that is a routine task in bridge construction, but software engineers are terrified of. Once the overall product is in a fairly stable situation, bridge constructors look for the next bridge to apply their knowledge to while software engineers never stop maintaining the database application. The application area evolves, specifications get modified, implementations must reflect design updates in a consistent way. Consequently, the problem we face is the abstract representation of what we constructed, the design decisions we made, the relationships across the layers, so that we are able to reason on our product and reconstruct parts of it in a consistent way with respect to the whole product.

Of course, a uniform representation is an indispensable prerequisite for using this knowledge efficiently. Current software CAD databases [RW89] emphasize the storage of software objects [LK86] and the integration of active components such as attribute evaluators [HK89], but tend to neglect the management of design decisions [PB88]. Integration and schema evolution problems have been dealt with only at the programming level [BKKK87].

When the DAIDA project started in 1985 to investigate a comprehensive environment for information systems engineering, all three classes of activities had to be considered: semantic modeling and database programming languages, suitable intermediate representations and mappings, and design decision support [BJM*87]. Meanwhile, a set of coherent concepts have evolved and their feasibility has been shown through the realization of an integrated DAIDA prototype [DAIDA89].

In this paper, we present a substantial portion of this work, excluding, however, the requirements engineering and rapid prototyping components of the DAIDA environment. In section 2, we propose two specific languages dealing with semantics and computations of database applications, respectively. The construction principles of these languages have the advantage of keeping the conceptual distance between the semantical and computational aspects manageable but cannot remove the impedance mismatch between their features completely.

Inheritance is supported for data classes as well as for transaction classes, allowing the organization of objects and the reuse of transaction specifications. This abstraction and structuring of the statics and dynamics of the database application, and the expression of specific classes of integrity constraints in a short, convenient way in terms of attribute categories represent the main TDL features useful for representing database-intensive applications.

Integrity constraints which cannot be represented by attribute categories can be described with the assertional language supported by TDL, a first-order predicate language with equality. The pre-/post condition style is adopted to express specifications of transactions in terms of the states before and after the execution of the transaction. We only need to express the changes that are caused by a transaction and implicitly accept that everything else remains unchanged -- the "frame assumption". This decision greatly facilitates the presentation of specifications but makes the semantic layer less explicit.

We give the TDL description for a small database example dealing with project management:

```
TDLDESIGN  ResearchCompanies IS

 ENUMERATED  CLASS
   Agencies = {'ESPRIT, 'DFG, 'NSF };

 ENTITY CLASS Companies WITH
   UNIQUE, UNCHANGING name : Strings;
   CHANGING engagedIn : SetOf Projects;
 END Companies;

 ENTITY CLASS Employees WITH
   UNCHANGING  name : Strings;
   CHANGING  belongsTo : Companies;
   worksOn: SetOf Projects;
   INVARIANTS onEmpComp: True IS
       (THIS.worksOn SubSetOf
        THIS.belongsTo.engagedIn);
 END Employees;

 ENTITY CLASS Projects WITH
   UNIQUE, UNCHANGING
       name : Strings;
       getsGrantFrom : Agencies;
   CHANGING
       consortium : SetOf Companies;
   INVARIANTS onProjComp: True IS
     (THIS.consortium =
        {EACH x IN Companies:
         THIS IsIn x.engagedIn});
 END Projects;

TRANSACTION CLASS HireEmployee WITH
    IN  name : Strings;
        belongs : Companies;
        works : SetOf Project;
    OUT, PRODUCES e : Employee;
    GIVEN
        THIS.works SubSetOf
        THIS.belongs.engagedIn;
    GOALS
      (e.name' = name) AND
        (e.worksOn' = works) AND
        (e.belongsTo' = belongs)
  END HireEmployee ;
END ResearchCompanies;
```

The language DBPL [SEM88] emphasizes the concept of transaction-oriented database programming. The major modeling constructs are sets and predicates. Sets are used as a bulk data structure and as an orthogonal type constructor. Like in NF^2 relations [SP82], the underlying data model is a value-based one.

The inherent constraints supported by such a data model include the necessity of values (no null values) within structured data and the uniqueness of values of specific attributes of structured sets (relations), characterized as keys. Additionally, first-order predicates over implicitly declared set element variables are integrated into DBPL expressions. Orthogonal persistence is provided by encapsulating data objects and transactions in so-called database modules.

DBPL's transaction orientation and its rich typing system (though without inheritance) reduce the conceptual distance to TDL, making it a promising "second pillar" for application development. Full database functionality including, e.g., concurrency control is provided by the DBPL system.

We give a DBPL program of our small database example:

```
DEFINITION  MODULE
  ResearchCompaniesTypes;
 IMPORT Identifier,String;
 TYPE
  Agencies = (ESPRIT, DFG, NSF);
  CompNames, EmpNames,ProjNames = String.Type;
  EmpIds = Identifier.Type;
  ProjIdRecType = RECORD name : ProjNames;
                  getsGrantFrom : Agencies END;

  ProjIdRelType = RELATION OF ProjIdRecType;
  CompRelType = RELATION name OF
                  RECORD
                   name : CompNames;
                   engagedIn : ProjIdRelType
                  END;
```

```
  EmpRelType = RELATION employee OF
                 RECORD
                  employee : EmpIds;
                  name : EmpNames;
                  belongsTo : CompNames;
                  worksOn : ProjIdRelType
                 END;
  ProjRelType = RELATION projId OF
               RECORD
                projId : ProjIdRecType;
                consortium:
                   RELATION OF CompNames
               END;
END ResCompaniesTypes.

DATABASE DEFINITION MODULE
   ResearchCompaniesOps;
  FROM ResearchCompaniesTypes
  IMPORT EmpNames, CompNames,
        ProjIdRelType, EmpIds;
TRANSACTION hireEmployee(name:EmpNames;
  belongs:CompNames; works:ProjIdRelType)
                                  : EmpIds;
END ResearchCompaniesOps.

DATABASE IMPLEMENTATION MODULE
   ResearchCompaniesOps;
  FROM ResearchCompaniesTypes
  IMPORT CompRelType; EmpRelType; ProjRelType;
  IMPORT Identifier;

 VAR
  compRel : CompRelType;
  empRel : EmpRelType;
  projRel : ProjRelType;

 TRANSACTION hireEmployee (name : EmpNames;
  belongs : CompNames;  works : ProjIdRelType)
                                  : EmpIds;
 VAR tEmpId : EmpIds;
 BEGIN
  IF SOME c IN compRel (c.name = belongs) AND
   ALL w IN works
   SOME p IN compRel[belongs].engagedIn (w = p)
  THEN  tEmpId := Identifier.New; empRel :+
        EmpRelType{<tEmpId,name,belongs,works>};
        RETURN tEmpId
  ELSE RETURN Identifier.Nil END
 END hireEmployee;
END ResearchCompaniesOps.
```

3 Mapping the Semantic Layer to Computations: Expressiveness vs. Efficiency

The predicative style of TDL, combined with object orientation, provides reasoning capabilities and abstraction principles that are extremely valuable within the semantic layer. Nevertheless, this framework also causes a mismatch with the imperative and value-based style of DBPL. We enter the second dimension of activities where we have to bridge the gap between the technologies. The approach we took was to build a translator that explicitly writes out all assumptions and inherent constraints underlying the TDL model, and then re-expresses them in the specification formalism we use [BMSW90].

Among the different styles of software specification that have been proposed (logic-based, functional, algebraic, ...), we had to choose one which fits well with TDL descriptions. The model-based style -- also present in VDM or Z [SPIV89] -- turns out to be the appropriate candidate. At the same time, a formalism for our needs must be supported by a reasonably developed methodology and by a supporting technology suitable for a database-oriented target language such as DBPL.

The third pillar of our "bridge" construction is therefore based on Abstract Machines (AMs), a notation using basic set theory, first-order predicates and generalized theories [AGMS88]. Appropriate theories defined on Abstract Machines support the transformation of TDL models into AM descriptions, the formal structural and operational refinement of Abstract Machines down to our computational model and the final transformation into transaction-oriented database programs. These transformations are formally represented as so-called Generalized Substitutions [AGMS88, BMSW90].

The AM description of our small TDL example is:

MACHINE researchCompanies.initialversion
 BASIC SETS
  Agencies, Companies, Employees, Projects,
  CompNames, EmNames, ProjNames
 CONTEXT
  Agencies = { ESPRIT, DFG, NSF };
  CompNames, EmpNames, ProjNames = Strings
 VARIABLES
  companies, compName, engagedIn, employees,
  empName, belongsTo, worksOn,
  projects, projName, getsGrantFrom, consortium
 INVARIANTS
  companies IN *POW*(Companies);
  compName IN (employees -> CompNames);
  engagedIn IN (companies -> *POW*(Projects));
  employees IN *POW*(Employees);
  empName IN (employees -> empNames);
  belongsTo IN (employees->companies);
  worksOn IN (employees -> *POW*(projects);
  projects IN *POW*(Projects);
  projName IN (projects -> ProjNames);
  getsGrantFrom IN (projects->Agencies);
  consortium IN (projects -> *POW*(companies));

```
∀ x, y. x, y IN companies ==>
    (compName(x) = compName(y) ==> x = y);
∀ x. x IN employees ==>
    (worksOn(x) SUBSET engagedIn(belongsTo(x));
∀ x, y. x, y IN projects ==>
    (projName(x) = projName(y) ==> x = y);
∀ x. x IN projects ==>
    (consortium(x) = { y | y IN companies AND
                           x IN engagedIn(y) })
OPERATIONS
  HireEmployee (name, belongs, works) =
  PRE     name IN EmpNames AND
          belongs IN companies AND
          works IN POW(engagedIn(belongs))
  THEN    ANY e IN (Employees - employees)
     THEN  (empName(e), worksOn(e), belongsTo(e))
                    |<- (name,works,belongs) ||
          employees |<- employees UNION {e} ||
          HireEmployee |<- e  END
  END HireEmployee;
END researchCompanies.initialversion
```

To explain the perhaps unfamiliar notation, consider its last part, `OPERATIONS`. `HireEmployee` defines three attribute functions, one variable `e`, and the results function by parallel (||) textual substitution (|<-). These parallel substitutions are part of an unbounded non-deterministic `ANY`-substitution which chooses an arbitrary fresh member from the set of elements considered for employee representation (basic set `Employees` minus existing `employees`). The entire substitution is preconditioned by "type conditions" on the input parameters (`PRE`).

The reader may have noticed a rough correspondence between the constructs of the TDL and AM formalisms: classes become sets, attributes become mappings between sets, integrity constraints become invariants, and pre/post statements become generalized substitutions.

A proof assistant, the B-Tool [ABRI86], has been used to encode the knowledge of TDL-AM transformation. It supports the software engineer in proving the consistency of an Abstract Machine. It also supports the consistent refinement of an Abstract Machine by creating proof obligations, applying proof tactics, and keeping track of proven or still open lemmas. In the case of transactions, this refinement gradually leads from predicative specifications to imperative code; each such operational refinement is based on a corresponding data refinement.

The crucial factor, however, during the overall refinement process are the *design decisions* taken by the software engineer in order to come up with application descriptions that can be executed.

The necessity of introducing computational concepts leads to design decisions for *data identification, data structuring, data typing,* and *operational refinement.* Specific toolkits containing theories and tactics for each of these activities support the execution of design decisions. In DAIDA, full automation has only been possible for certain substeps of this abstract-machine based methodology. Further research is needed to determine how far automation can be carried and at what points human decisions remain necessary; the current status of this work is presented in [BMSW90].

The following Abstract Machine represents a consistent refinement of the previous one, where data structuring (e.g., `EmpClass` as a cross-product) and operational refinement (e.g., assignment to `EmpClass`) have been introduced:

```
MACHINE  researchCompanies.refinedVersion
IMPLY researchCompanies.initialversion
VARIABLES
  compClass, empClass, projClass, tEmpId
INVARIANTS
  compClass IN (companies  ->  POW (projects));
  empClass IN (employees ->
      EmpNames x  companies x POW (projects));
  projClass IN (projects IN POW(companies));
  tEmpId IN EmpIds;
  ...
DEFINITIONS
  engagedIn =  λ x. (x IN companies | compClass(x));
  (empName, belongsTo, worksOn) =
      λ x. (x  IN employees | empClass(x));
  consortium =  λ x. (x IN projects | projClass(x));
OPERATIONS
  HireEmployee (name, belongs, works) =
     PRE      name  IN EmpNames AND
              belongs IN companies AND
              works IN POW(engagedIn(belongs))
     THEN     tEmpId |<- newEmpId;
              empClass  |<- empClass UNION
                   {(tEmpId, name, belongs, works)};
              HireEmployee  |<- tEmpId
     END
  END HireEmployee;
OTHER
  newEmpId  IN ( -> (EmpIds - employees));
END researchCompany.refinedversion;
```

4 Software Object and Dependency Modeling

The modeling of software development is of outstanding importance during the third dimension of activities, the maintainance of the system. The structure and interrelationships of evolving system components have to be captured in order to support system designers in modifying parts of the software system in a consistent way.

We claim that this is essentially a typical database problem. We have to manage a large set of objects, their relationships, and their evolution caused by particular development and maintenance activities. This domain is well-known in the database community since the E-R epoch; recent E-R extensions dealing with activity modeling include RML [GBM86] and ERAE [HAGE88].

This view does not only allow us to model the software objects, relationships and software engineering activities composing the software development domain. Going further, we can now define *what are the consistent means by which we should proceed* and reason *whether our system has been evolved in a consistent way*. This is because such a model need not define a methodology in terms of fixed predefined steps but can describe it in terms of design goals and constraints whose satisfaction is then accomplished and verified by particular steps taken during development and maintainance.

What we need, consequently, is a semantic model capturing the statics (software objects, relationships) and the dynamics (development and maintainance activities) of the software modeling activity in a form appropriate for reasoning. Extending earlier work on RML [GBM86], the knowledge representation language *Telos* was elaborated in DAIDA with exactly these goals in mind [MBJK90]. It is a structurally object-oriented language into which a time calculus (not used in this paper) and the rules and integrity constraints of deductive databases have been integrated.

In the following, we give a Telos description of some TDL and AM language constructs as well as the description of the dependencies that can be established between them. All these descriptions can be thought of as uniformly represented metalevel knowledge about the two layers, allowing us to establish and maintain dependencies across objects of different languages; these objects and dependencies can be further restricted by Telos´ predicative rules and constraints. We start with the Telos description of some TDL classes modeling static aspects of application domains (the reader can easily establish the relationship to our TDL example in section 2):

```
IndividualClass TDL_Design
  in MetaClass with
  attribute
     entities: TDL_EntityClass;
     transactions: TDL_TransactionClass;
     enumerated: TDL_EnumeratedClass;
     aggregates: TDL:AggregateClass;
     basicclasses: TDL_BasicClass
end TDL_Design
```

```
IndividualClass TDL_EnumeratedClass
  in MetaClass isA TDL_DataClass
end TDL_EnumeratedClass

IndividualClass TDL_EntityClass
  in MetaClass isA TDL_DataClass with
  attribute
     CHANGING,UNCHANGING,UNIQUE:
                        TDL_DataClass;
     INVARIANTS: TDL_AssertionClass
end TDL_EntityClass

IndividualClass TDL_TransactionClass
  in MetaClass isA TDL_Class with
  attribute
     IN,OUT,PRODUCES: TDL_DataClass;
     GIVEN,GOALS: TDL_AssertionClass
end TDL_TransactionClass

IndividualClass TDL_AssertionClass
  in MetaClass isA TDL_Class, String
end TDL_TransactionClass
```

Note that the abstract description of TDL assertions is simply a string. Next, we describe Abstract Machines:

```
IndividualClass AbstractMachine
  in MetaClass with
  necessary
    basicsets: AM_BasicSet;
    initializations: AM_Initialization;
    variables: AM_Variable;
    invariants: AM_Invariant;
    operations: AM_Operation
  attribute
    definitions : AM_Definition;
    contexts: AM_Context;
    others: AM_Other
end AbstractMachine

IndividualClass AM_Operation
  in MetaClass with
  attribute
    PRE,THEN: String
end AM_Operation
```

Since we have now modeled both languages within a common formalism, we can formally specify possible dependencies between TDL and AM objects. Dependencies are represented as classes of attributes (with category `dependson`) attached to derived objects. In our example, the most general dependency `InitialAbstractMachine!dependsonTdl` establishes that an Initial Abstract Machine is derived from a single TDL design. The constraint demands that attributes of the Initial Abstract Machine are derived from attributes of the same TDL design. The model can easily be semantically enriched by further constraints.

```
IndividualClass InitialAbstractMachine
  isA AbstractMachine with
  dependson, single
    dependsonTdl: TDL_Design
   constraint
    completeMapping:
    $ forall y/TDL_Design, a/Attribute
      (THIS.dependsonTdl = {y} and From(a,THIS)
        ==>exists d/Class!dependson
           exists b/Attribute
        From(d,a) and To(d,b) and From(b,y) ) $
end InitialAbstractMachine
```

In Telos notation, x.a gives the *value* of an attribute of category a while x!a stands for the attribute link a of x itself. THIS denotes an arbitrary instance of the class object being defined. The component dependencies below establish some (example) constraints on the derivation of an Abstract Machine's basic sets, contexts, operations, and invariants.

```
AttributeClass
  InitialAbstractMachine!basicsets with
  dependson, single
    dependsonEnum: TDL_Design!enumeratedclasses
end InitialAbstractMachine!basicsets

AttributeClass
  InitialAbstractMachine!contexts with
  dependson, single
    dependson: TDL_Design!enumeratedclasses
  constraint
    correspondingBasicsetExists :
    $ exists x/ InitialAbstractMachine
        ib/ InitialAbstractMachine!basicsets,
           (From(ib,x) and From(THIS,x)) $
end InitialAbstractMachine!contexts

AttributeClass
  InitialAbstractMachine!operations with
  dependson
    dependsonTransaction:TDL_Design!transactions
end InitialAbstractMachine!operations

AttributeClass
  InitialAbstractMachine!invariants with
  dependson
    tdlInvariant: TDL_EntityClass!INVARIANT;
    uniqueTdlAttr: TDL_EntityClass!UNIQUE
  constraint
    correctEntities: $ forall x/TDL_Design,
        am/InitialAbstractMachine
        (From(THIS,am) and am.dependsonTdl = {x}
        ==>
        (exists ea/Attribute, e/TDL_EntityClass
         From(ea,e) and  e outOf x.entities and
         (ea outOf THIS.tdlInvariants or
         ea outOf THIS.uniqueTdlAttr)) ) $
end InitialAbstractMachine!invariants
```

Going on this way all possible dependencies between software objects can be specified. Particular dependencies for a specific design would be represented as instances of the above dependency classes. It is quite easy now to guess how we support the third dimension maintenance activities. A software information system keeps track of the dependencies during the development and maintainance of a software system component. It can identify the consequences of modifications not only within the language layer where the modification took place, but also across the layers.

5 From Dependency Modeling to Design Decision Support: ConceptBase

In the DAIDA prototype environment, the interaction of bridging activities (sec. 3) and object-dependency (sec. 4) modeling has been implemented by coupling knowledge-based mapping tools with the Telos-based software information system, *ConceptBase* [EJJ*89]. A mapping assistant documents its activities and their results in ConceptBase, and retrieves information. The ConceptBase usage environment offers a window-based and graphics-oriented set of tools for browsing, zooming, and editing hypertext-like views of the software knowledge that may guide the software engineer (the user of the mapping assistant) in further activities.

In the following subsections, we first sketch the implementation of the TDL-to-DBPL mapping methodology via AMs, and then exploit this experience to identify several important model extensions we have incorporated in the ConceptBase system.

5.1 Implementation of Mapping Methodology

We support the particular mapping methodology introduced in sections 3 and 4 as illustrated in figure 1. In this figure, intermediate results are denoted by rectangles, design steps or decisions by ovals, and design tools by rounded boxes. Links establish the input-output relationships.

A first step translates a TDL design into an Initial Abstract Machine which is verified for consistency. After the consistency proof, the AM is subjected to a series of verified refinements. The last refined machine (baseline) is automatically translated into DBPL code. The formal properties of Abstract Machines and refinements are assured and organized with the help of Abrial's interactive proof assistant, the B-Tool [ABRI86]. Another tool, the language-sensitive editor DBPL-USE [NS89], provides syntactic and some semantic support for correct programming and program interconnection of DBPL modules that come from outside the DAIDA environment.

The refinement process is directed interactively by the user and controlled formally by the mapping assistant. The mapping assistant itself is organized as a toolkit from which the developer must choose problem-adequate theories (i.e., sets of previously proven theorems) and tactics (sequencing rules for the application of theories) to be employed by the B tool.

Additionally, there is usually a large set of open proof obligations, called lemmas. The generation of refinements and their correctness proofs is quite complicated and requires a lot of knowledge about available theories and useful proof tactics for specific proving tasks. This experience led to a first extension of the mapping model managed by ConceptBase: we needed to model not only the refinement steps but also their associated correctness proofs, and even individual steps of those proofs. The proof model can again be seen as an object-dependency structure.

Fig. 2 illustrates this claim for one particular refinement step. Dependencies (here drawn as vertical arrows, e.g., `dep-opn_1`) are used to document individual operational refinements. Each dependency creates a proof obligation (e.g., the box pointed to by `opn_1_tobeproven`) which in turn requires a complex hierarchy of proof steps; only if all of these proof steps succeed for all operations (either formally or because the user signs them off as correct), the `WholeProof` object for the refinement will be created in the knowledge base.

5.2 Extensions to Object-Dependency Model

The two figures reveal some deficiencies in the object-dependency model of section 4. For example, besides the dependencies which are directly related to the specific methodology at hand, figure 1 also illustrates more generic kinds of activities which are associated with software project management. The picture shows the distinction between *initial* versions, *refinements* within a particular language context, *release* decisions for temporarily frozen object versions, and *mapping* between different language contexts. We may also need to represent the relationships between different *variants* or the discussion of *design goals*.

In ConceptBase, we have decided to *separate the semantic descriptions* given by the models discussed in section 4, *from the administrative aspects* of a software information system. This has two advantages. Firstly, each administrative object can be associated with both, a semantic description and the actual software object. Secondly, version and configuration management problems can be addressed at both a conceptual and a storage level [RJ90] so that we can combine the advantages of a software database with those of a knowledge-based mapping assistant.

A related observation that is not adequately addressed in the above model is that the dependencies are created by the execution of *human design decisions*. Such design decisions are free within a prescribed methodology, they can be driven by goals and can be argued about in design teams. ConceptBase makes the notion of design decisions *explicit* and provides tools for multi-objective decision-making and argumentation support [HJR90]; another part of the DAIDA environment has explored the idea of goal satisficing for non-functional requirements [CKM*90]. The idea of separating administration and semantics of objects is now applied to design decisions: the dependencies defined in section 4 become the semantic description of their underlying design decision.

Finally, figure 1 illustrates that we have neglected the existence of multiple interacting *tools* in our initial model. To evaluate or replay a design history, we have to know which tools were used to create the version we are looking at. Moreover, the whole approach proposed here is so documentation-intensive that it becomes economically feasible only in a *CASE* (*computer-aided* software engineering) environment. The *formal modeling and technical interconnection of an open toolset* of mapping assistants, layer-specific editors, compilers, proof assistants, etc. is therefore a necessity, albeit one not addressed in most existing software information systems.

ConceptBase models *tools as reusable software objects* that are specified in a TDL-like style and implemented in any programming language. Technically, such tools can be connected to ConceptBase by interprocess communication in a client-server architecture. A trigger concept added to the Telos language controls the activation of such tools via their specification [JJR89]. The way how this is implemented is closely related to active databases [DBM88].

Summarizing, we have identified three extensions. Taken together, they generalize the object-dependency approach of section 4 to a *Decision-Object- Tool or D.O.T. model*:

- separation of administrative and semantic aspects of *object* management
- *decision* support instead of just dependency recording
- *tool* modeling and technical tool integration.

A final point is *extensibility*. The full DAIDA environment does not only support the specification-to-implementation mapping discussed in this paper. It also includes requirements engineering and prototyping sub-environments and a mapping assistant for the derivation of TDL descriptions from requirements [CKM*90]. Although these subtasks address rather different problems, we have been able to model their execution within the same framework [DAIDA89].

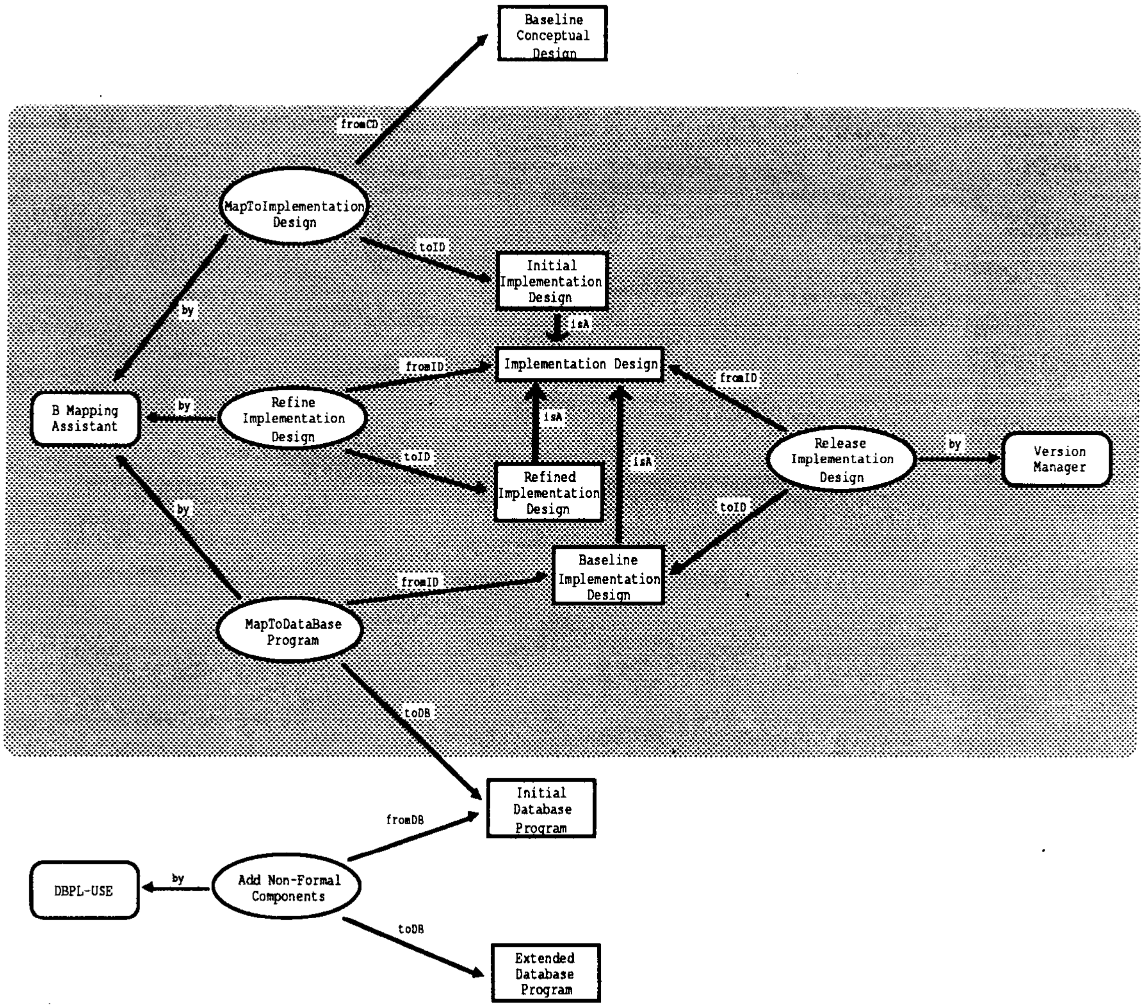

Fig. 1: Mapping TDL specifications to DBPL programs in the DAIDA environment

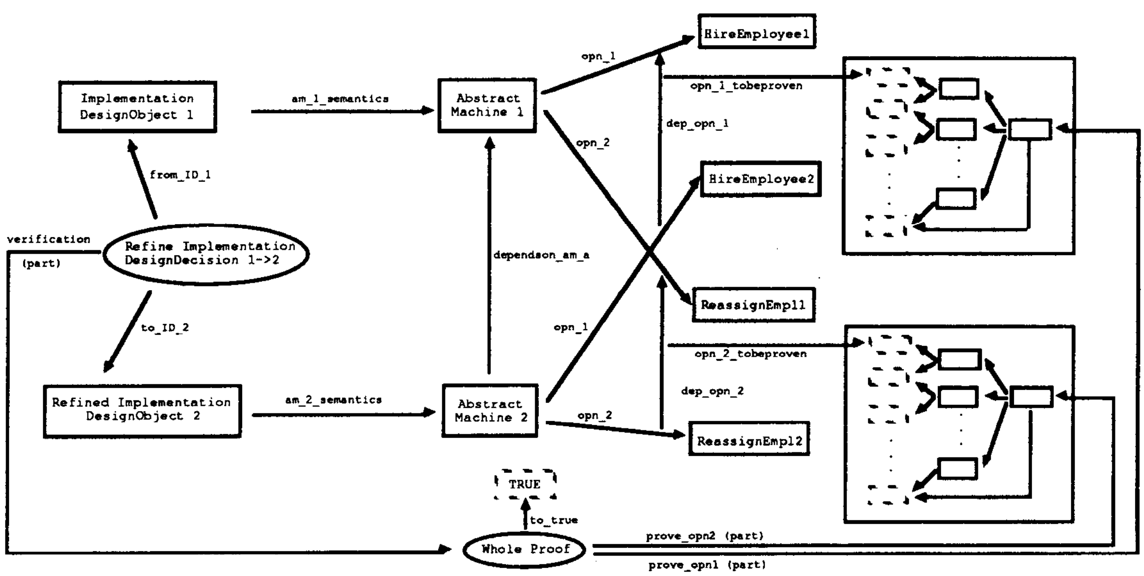

Fig. 2: Example of interaction between refinement and proofs

For this purpose, we have defined a *generic D.O.T. model of tool-assisted information systems processes* at the next-higher level of the Telos metaclass hierarchy which has all the classes defined in section 4 as well as others as its instances [JJR90]. This metamodel consists of the following Telos classes:

```
IndividualClass Class with
  attribute
        attribute : Class;
        dependson : Class;
        trigger   : Behavior
end Class

IndividualClass DesignObject
  in MetametaClass with
  attribute
        objsemantic : Class;
        objsource   : ExternalReference
end DesignObject

IndividualClass DesignGoal isA DesignObject
end DesignGoal

IndividualClass DesignDecision
  isA DesignObject with
  attribute
        from, to    : DesignObject;
        goals       : DesignGoal;
        decsemantic : DecisionDescription
end DesignDecision

IndividualClass DecisionDescription
  in MetametaClass with
  attribute
        dependencies : Class!dependson
end DecisionDescription

IndividualClass DesignTool
  isA DesignDecision with
  attribute
        from : DesignDecision;
        to   : Behavior
end DesignTool
```

Figure 1 from section 5.1 can be understood as a semantic network view of our mapping knowledge base that directly reflects the D.O.T. structure: software object classes are denoted by rectangles, decision classes by ovals, and tools by rounded rectangles. For example, the objects, decisions, and tools involved in the initial translation from the TDL formalism to Abstract Machines is represented as an instance of the D.O.T. metamodel as follows (the classes defined in section 4 form the semantic descriptions):

```
IndividualClass BaselineConceptualDesign
  in DesignObject with
  objsemantic
       : TDL_Design
  objsource
       : TDL_Directory
end BaselineConceptualDesign

IndividualClass
  InitialImplementationDesign
  in DesignObject with
  objsemantic
       : InitialAbstractMachine
  objsource
       : B_Directory
end InitialImplementationDesign

IndividualClass MapToImplementationDesign
  in DesignDecision with
  from
       fromCD : BaselineConceptualDesign
  to
       toID   : InitialImplementationDesign
  decsemantic
       : TDL_AM_Description
end MapToImplementationDesign

IndividualClass TDL_AM_Description
  in DecisionDescription with
  dependencies
       : InitialAbstractMachine!dependsonTdl
end TDL_AM_Description

IndividualClass B_Mapping_Assistant
  in DesignTool with
  from
       suppDecision : MapToImplementationDesign
  to
       b_tool_call : "/private/daida/gobee"
end B_Mapping_Assistant
```

Figure 3 is a ConceptBase screendump which illustrates the above model and its instantiation. The left side shows part of the model in a graphical editor/browser. The editor on the upper right shows an instance of the `AbstractMachine` class defined in section 4, for the example information system used in sections 2 and 3; the names differ slightly since in reality our example is embedded in a longer history of system versions and configurations [DAIDA89].

Without going into details of Telos syntax, figure 4 illustrates the instantiation of D.O.T. by another example: the handling of proofs as in figure 2. Proof obligations and theories are modeled as instances of class `DesignObject`, and proofs as hierarchically nested design decisions. Each step is supported by a prover tool which consists of B enhanced by specific theories and tactics.

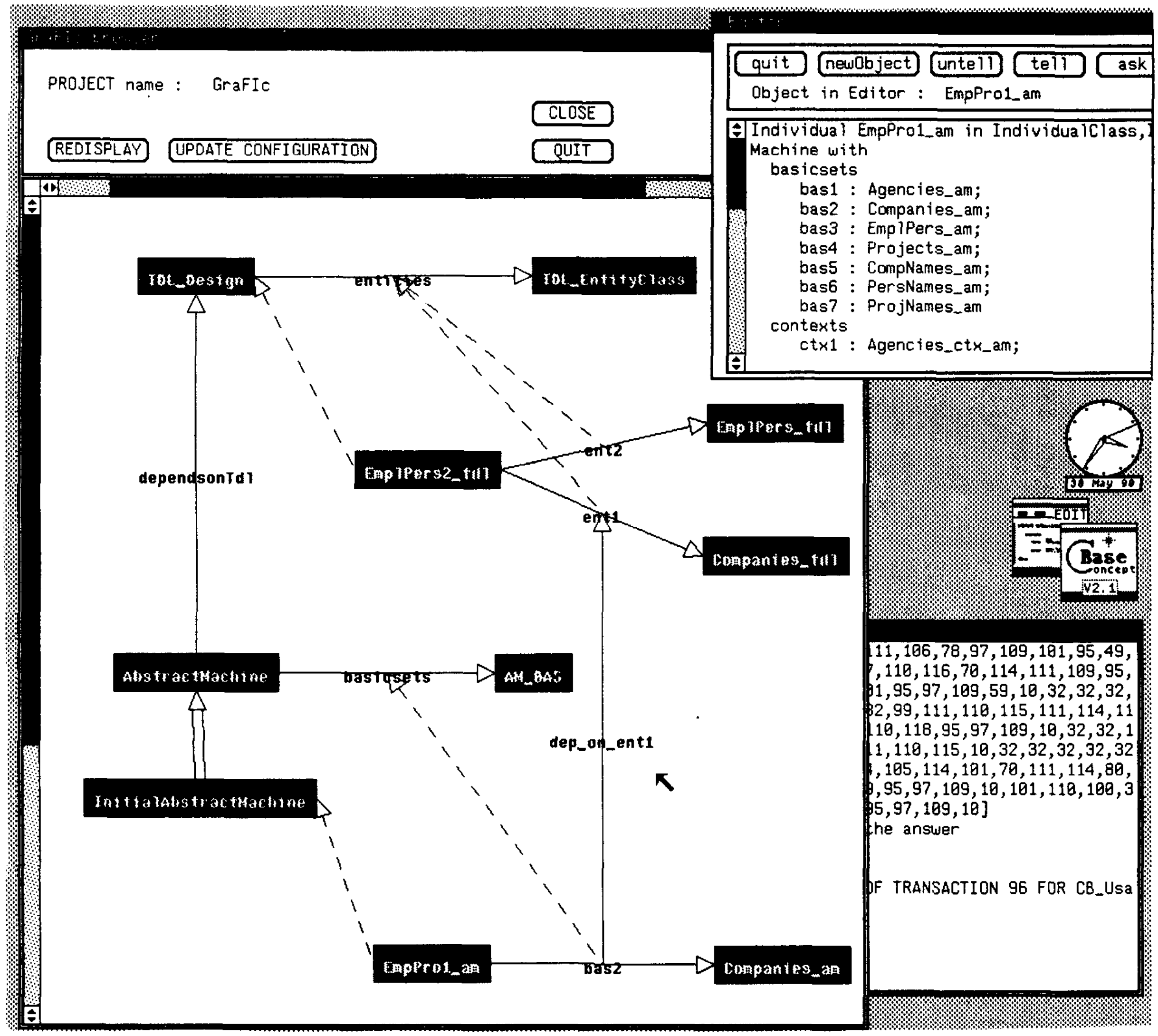

Fig. 3: ConceptBase screendump with mapping model and example object instance

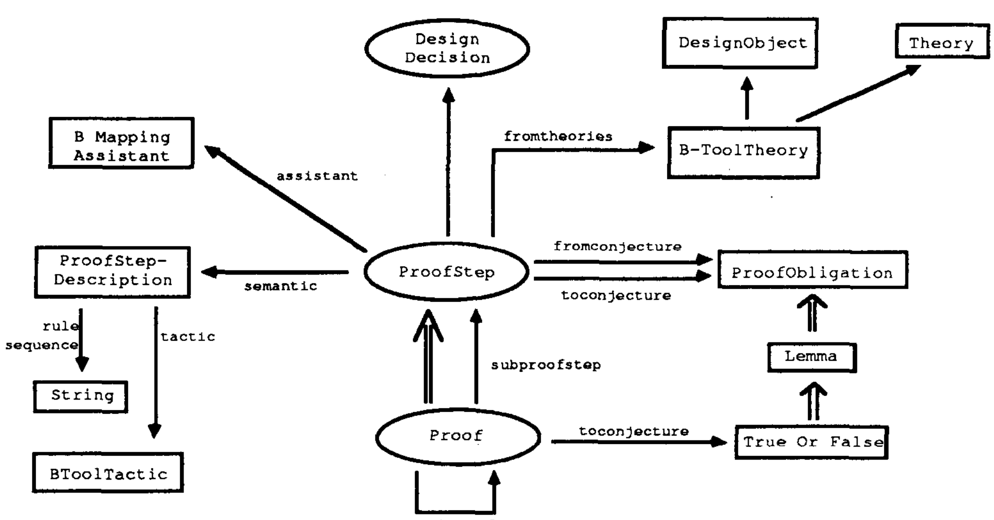

Fig. 4: Class-level network representation of proof management model based on D.O.T.

6 Discussion and Conclusion

We started this paper with drawing an analogy between database application development and bridge construction, and with asking three questions: What are the basic pillars -- the semantic and computational modeling languages -- we need? How can we bridge the gap between them by a suitable specification formalism (a third pillar in the right place) and mapping methodology? Finally, how can we support maintenance by abstract modeling of software objects and design decisions?

Our answer to these questions has been the integration of

- a *semantic modeling language* (TDL) which supports an object-oriented and predicative style of conceptual specification,
- a *transaction-oriented database programming language* with sets and predicates (DBPL),
- an appropriate *formal method and tool* (Abstract Machine refinements and database-specific proof theories and tactics using the B tool) for individual software development tasks, and
- a *knowledge base management system* (based on Telos, the D.O.T. model and ConceptBase) to keep track of information about a system's underlying design decisions across multiple representational levels, and with method-dependent precision.

Aspects of this approach have also been addressed by a number of other projects on transformational and knowledge-based software engineering [PS83, BARS87], software hypertext systems [GS89], software databases [RW89], and project support environments [BROW88]. Our solution differs from these by its integration of database-specific languages, its integration of programming-in-the-small with version, configuration, and cooperation management (not discussed here [RJ90, HJR90]), and, for some aspects, simply by the fact that they have been implemented and experimented with.

Currently, our practical experience does not go beyond a few medium-scale applications. These applications share the need for a wide variety of structurally constrained objects but relatively simple operations that can be understood intuitively in terms of conditioned state transitions. Both, objects and operations, are constrained and interrelated by relatively high numbers of *general* first-order invariants and pre- and postconditions. Applications with such characteristics seem to be well served by the reported base technology, i.e., by the assertion language of TDL, the typing and querying mechanisms of DBPL, the Abstract Machine/ Generalized Substitution approach of the B Tool, and the corresponding D.O.T.-based software information schema. In a more restricted context, our implementation of the deductive and object-oriented KBMS ConceptBase has followed the same approach [JJR89].

Up to now we did not exploit the *specific* generalization/ specialization predicates of the TDL language, provided symmetrically for data class as well as transaction design. As a next step in applying the DAIDA framework, we are interested in gaining experience about the consequences of changes in Information Systems requirements, in particular of those changes that are *incremental* due to the nature of inheritance. This is intended to lead to formal support for a new object-oriented software lifecycle heavily based on reusability through inheritance, i.e., by specialization or generalization of existing components. The gain in productivity by re-utilizing a previous effort in proofs and refinements is expected to be considerable, in particular, when the need for future generalizations and specializations is foreseen and respected in the initial design and development.

References

[ABRI86] Abrial, J.R. (1986). An informal introduction to B. Manuscript, Paris, France.

[AGMS88] Abrial, J.R., Gardiner, P., Morgan, C., Spivey, M. (1988). Abstract machines, part I-IV. Manuscript, Oxford University, UK.

[BARS87] Barstow, D. (1987). Artificial intelligence and software engineering. *Proc. 9th Intl. Conf. Software Eng.*, Monterey, Ca, 200-211.

[BJM*87] Borgida, A., Jarke, M., Mylopoulos, J., Schmidt, J.W., Vassiliou, Y. (1987). The software engineering environment as a knowledge base management system. In Schmidt/ Thanos (eds.): *Foundations of Knowledge Base Management*, Springer-Verlag, 411-440.

[BKKK87] Banerjee, K., Kim, W., Kim, H.-J., Korth, H.F. (1987). Semantics and implementation of schema evolution in databases. *Proc. ACM-SIGMOD Conf.*, San Francisco, Ca, 311-322.

[BM86] Brodie, M.L., Mylopoulos, J. (1986). Knowledge bases and databases: semantic vs. computational theories of information. In Ariav/ Clifford (eds.): *New Directions for Database Systems*, Ablex, 186-218.

[BMMS88] Borgida, A., Meirlaen, E., Mylopoulos, J., Schmidt, J.W. (1988). Report on the Taxis Design Language TDL. Report, ESPRIT 892 (DAIDA), FORTH-CRC, Iraklion, Greece.

[BMSW90] Borgida, A., Mertikas, J., Schmidt, J.W., Wetzel, I. (1990). Specification and refinement of databases and transactions. Report, ESPRIT project 892 (DAIDA), Universität Hamburg.

[BR84] Brodie, M.L., Ridjanovich, D. (1984). On the design and specification of database transactions. In Brodie/ Mylopoulos/ Schmidt (eds.): *On Conceptual Modeling*. Springer, 277-306.

[BROW88] Brown, A.W. (1988). Integrated project support environments. *Information & Management 15*, 2, 125-134.

[CKM*90] Chung, L., Katalagarianos, P., Marakakis, M., Mertikas, M., Mylopoulos, J., Vassiliou, Y. (1990). From information systems requirements to designs: a mapping framework. To appear, *Information Systems.*

[DAIDA89] Jarke, M., DAIDA Team (1989). The DAIDA demonstrator: development assistance for database applications. *Proc. ESPRIT Conf.*, Brussels, Belgium, 459-474.

[DBM88] Dayal, U., Buchmann, A., McCarthy, D.R. (1988). Rules are objects too: a knowledge model for active, object-oriented databases. In Dittrich, K. (ed.): *Advances in Object-Oriented Databases*, Springer-Verlag, 129-143.

[EGL84] Ehrich, H., Lipeck, U., Gogolla, M. (1984). Specification, semantic, and enforce- ment of dynamic integrity constraints. *Proc. 10th VLDB Conf.*, Singapore, 301-308.

[EJJ*89] Eherer, S., Jarke, M., Jeusfeld, M., Miethsam, A., Rose, T. (1989). ConceptBase V2.1 user manual. Report MIP-8936, Universität Passau,.

[GBM86] Greenspan, S., Borgida, A., Mylopoulos, J. (1986). A requirements modeling language and its logic. In Brodie, M.L., Mylopoulos, J. (eds.): *On Knowledge Base Management Systems*, New York, Springer-Verlag, 471-502.

[GS89] Garg, P.K., Scacchi, W. (1989). ISHYS -- designing an intelligent software hypertext system. *IEEE Expert 4*, 4, 52-63.

[HAGE88] Hagelstein, J. (1988). Declarative approach to information systems requirements. *Knowledge-Based System 1*, 4, 211-220.

[HJR90] Hahn, U., Jarke, M., Rose, T. (1990). Group work in software projects. *IFIP Conf. Multi-User Interfaces & Applications*, Iraklion, Greece

[HK87] Hull, R., King, R. (1987). Semantic database modeling: survey, applications, and research issues. *ACM Comp. Surveys 19*, 3, 201-260.

[HK89] Hudson, S.E., King, R. (1989). Cactis: a self-adaptive, concurrent implementation of an object-oriented database management system. *ACM TODS 14*, 3, 291-321.

[JJR89] Jarke, M., Jeusfeld, M., Rose, T. (1989). Software process modeling as a strategy for KBMS implementation. *Proc. First Intl. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan, 496-515.

[JJR90] Jarke, M., Jeusfeld, M., Rose, T. (1989). A software process data model for knowledge engineering in information systems. *Information Systems 15*, 1, 85-116.

[LK86] Lyngbaek, P., Kent, W. (1986). A data modeling facility for the design and implementation of information systems. *Proc. Intl. Workshop Object-Oriented Databases*, Pacific Grove, Ca.

[MBJK90] Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M. (1990). Telos: a language for representing knowledge about information systems. To appear, *ACM Trans. Information Systems*.

[MBW80] Mylopoulos, J., Bernstein, P.A., Wong, H.K.T. (1980). A language facility for designing interactive, data-intensive applications. *ACM TODS 5*, 2, 185-207.

[NS89] Niebergall, P., Schmidt, J.W. (1989). DBPL-USE: a tool for language-sensitive database programming. Report, ESPRIT 892 (DAIDA), Universität Frankfurt, FRG.

[PB88] Potts, C., Bruns, G. (1988). Recording the reasons for design decisions. *Proc. 10th Intl. Conf. Software Eng.*, Singapore, 418-427.

[PS83] Partsch, H., Steinbrüggen, R. (1983). Program transformation systems. *ACM Computing Surveys 15*, 3, 199-236.

[RJ90] Rose, T., Jarke, M. (1990). A decision-based configuration process model. *Proc. 12th Intl. Conf. Software Eng.*, Nice, France, 316-325.

[RW89] Rowe, L.A., Wensel, S. (eds.): *Proceedings of the ACM-SIGMOD Workshop on Software CAD Databases.* Napa Valley, Ca.

[SEM88] Schmidt, J.W., Eckhardt, H., Matthes, F. (1988). Extensions to DBPL: towards a type-complete database programming language. ESPRIT 892 (DAIDA), Universität Frankfurt.

[SP82] Schek, H.-J., Pistor, P. (1982). Data structures for an integrated data base manage- ment and information retrieval system. *Proc. 8th VLDB*, Mexico City, 197-207.

[SPIV89] Spivey, J.M. (1989). An introduction to Z and formal specifications. *Software Engineering Journal 4*, 1, 40-50.

[SS89] Sheard, T., Stemple, D. (1986). Automatic verification of database transaction safety. *ACM Trans. Database Systems 14*, 3, 322-368.

[SSE87] Sernadas, A., Sernadas, C., Ehrich, H.-D. (1987). Object-oriented specification of databases: an algebraic approach. *Proc. 13th VLDB Conf.*, Brighton, UK, 107-116.