

Support for Data-Intensive Applications: Conceptual Design and Software Development¹

Alexander Borgida
Dept. Comp. Science
Rutgers University
New Brunswick, NJ08904
USA

John Mylopoulos
Dept. Comp. Science
University of Toronto
Toronto M5S 1A4, Ont.
Canada

Joachim W. Schmidt, Ingrid Wetzel
Fachbereich Informatik
Universität Frankfurt
D-6000 Frankfurt a.M.
F.R. Germany

Abstract

In the process of developing an Information System, one passes through stages that include requirements gathering, design specification, and software implementation. The purpose of the TDL language is to express the conceptual design of an information system; it is the intermediate language in a triad that includes the language Telos, which captures an evolutionary view of the application domain and requirements, and DBPL, a procedural programming language that has persistent values and transactions supporting the development of databases. We consider TDL's features for specifying the data and eventual procedural components of the system, and discuss how these are related to its companions. We also survey several tools for manipulating TDL descriptions which are currently under development, and give a detailed example of the iterative refinement of TDL designs into DBPL programs.

1 Motivation and setting

The authors and their collaborators, have been involved in project DAIDA, whose goal is to build a novel software engineering environment for developing and maintaining Information Systems [7]. It is generally agreed that the development of software systems, including Information Systems, involves stages such as requirements definition, design, implementation/testing.

The key features of the DAIDA project are

- The use of three specific (kinds of) languages for the description of the software at each stage: i) a knowledge representation language for domain analysis, ii) a semantic data model with specifications of procedures for design, and iii) an imperative programming language with efficient management of relational data.
- The use of a Knowledge Base Management System to keep track of
 - application-dependent knowledge about the problem domain, gathered as part of the requirements definition process;
 - meta-knowledge about the current system's design decisions and evolutionary history.
 - application-independent knowledge such as useful strategies for designing and implementing database-intensive software;

Among the expected contributions of the project will be a better understanding of the problems and tools/techniques for moving from a problem specification to the implementation of an Information System with significant data management component and a relatively straight-forward procedural component.

¹This is a PRELIMINARY REPORT on results from DAIDA, a project supported in part by the European Commission under ESPRIT contract #892 (DAIDA).

1.1 Telos – recording requirements

In order to determine the user requirements, we have argued that it is important to arrive at, and capture, an understanding of the application domain [8]. For this purpose, we have progressed through a series of languages, the latest called Telos, which allows one to represent a temporal model of the application domain in an assertional knowledge representation language [10]. Telos uses the paradigm of objects related by temporally qualified relationships to describe the entities and activities (both generic and specific) that occur in a particular world. The following example gives a flavor of Telos specifications. It describes a Project activity, which uses up budgeted money, producing progress and research reports, as well as associated meetings along the way.

IndividualClass Project in ActivityClass with

single input

budg : Budget

single output

finalReport : ResearchReport

single control

proposal : ProjectProposal

nm : Name

startDate, endDate : Date

part

meets: HaveMeeting

genProgressReport : GenerateReport

...

finalCondition

budg.amountLeft = 0

necessary

*{ * one progress report is produced every 6 months during the project * }*

(ForEach x/Time)[x during this \Rightarrow (Exists 1..1 y/genProgressReport)

[(y startsAfter x) and (y endsBefore x+6month)]]

Although not necessarily apparent from this example, the crucial fact about Telos is that *every aspect* of the description is associated with *time intervals*: the occurrence of activities, the membership of objects in classes, the relationships between objects, activities, etc.; in fact, in a recursive process, not only is the domain viewed temporally, but also the description itself is temporally indexed.

1.2 DBPL – a systems implementation language

At the other end – implementation – we have available the database programming language DBPL [19], a successor of Pascal/R [18]. DBPL integrates a set- and predicate-oriented view of database modelling into the system programming language Modula-2 [23]. Based on integrated programming language and database technology, it offers a uniform framework for the efficiency-oriented implementation of data-intensive applications.

A central design issue of DBPL was the development of abstraction mechanisms for database application programming [20], [15]. DBPL's bulk data types are based on the notion of (nested) sets and first-order predicates are provided for set evaluation and program control. Particular emphasis had been put on the interaction between these extensions and the type system of Modula-2. DBPL extends Modula-2 *orthogonally* into three dimensions:

- bulk data management through a data type *set* (relation);

- abstraction from bulk iteration through *access expressions*;
- *persistent modules* and *transactions* for sharing, recovery, and concurrency control.

1.3 TDL – the design language

There are several problems with jumping from the user’s conceptual requirements directly to DBPL code: first, not everything seen by the user in the world needs to be part of the final software system – much of it is contextual information; secondly, Telos is meant to capture the ontology of the user domain (so a project is viewed as an activity) while a database usually captures information *about* the domain (so there may be a *data* class corresponding to some activities in the world); third, in most environments there are multiple needs/goals that are being achieved, and these need to be integrated; finally, the descriptions in Telos and DBPL are just too widely different in style. TDL therefore is concerned with the intermediate stage of *software design*: identifying and specifying the data and procedural *components* of the system, which themselves are to be implemented in DBPL.

Briefly summarized, TDL is a language for describing the management of data maintained as *classes of objects related by attributes*. The state of the database is reflected by the extents of the classes, and the values of objects’ attributes, which are subject to certain integrity constraints. *Transactions* are atomic state changes. TDL also introduces *functions* to aid in the expression of assertions, *scripts* to aid the description of global control constraints and *communication* facilities to support the frequent needs of Information Systems for message passing, coordination and timing constraints.

Speaking comparatively, the significant difference between the requirements language Telos and the design language TDL is that Telos supports a *descriptive* view of the world through time, which emphasizes the relationship of the values at different points of time; on the other hand, TDL changes over to the state-based view of computation which is prevalent in most programming languages, especially ones supporting database access: the computer system has a *state*, and the design specifies the components of the state and state transitions. However, TDL preserves the object-oriented nature of Telos descriptions. Therefore the principal activities in moving from a Telos description to a TDL design is view integration, the choice of cases when historical information will be preserved (and its form), and the operations available to users.

DBPL, in turn, differs from TDL in having a relational view of the data, without the notion of object identity or subclass hierarchies, and, of course, describing functions and transactions in deterministic procedural rather than assertional terms.

As shall be seen, the basic ideas in TDL have been derived from our experience with the programming language Taxis [17]. The novelties include multi-valued attributes, a set-oriented expression language, and the predicative techniques for specifying the “dynamic” parts of the system: transactions, functions, scripts, and derived classes and attributes.

We shall briefly discuss in this paper several classes of software tools associated with the TDL language. Some of these tools are intended to detect errors before going to the expense of implementing the design. One obvious techniques for early detection of errors is providing for redundancy in the specification, and then verifying the consistency of the result. Another proposal is to permit the rapid prototyping of the software on small sets of data, without going through the implementation phase.

Second, we shall also outline certain innovative facilities and techniques that can be used to iteratively map a TDL specification into a DBPL program, through successive refinements supported by Abrial’s Abstract Machine approach[3][2].

2 Data description facilities in TDL

A number of different kinds of data classes can be defined in TDL:

1. Base classes: integer, real, string, boolean;
2. Enumerated classes. e.g., *ExpenseKinds* = {*travel*, *equipment*, *salary*};
3. Aggregate classes, which are essentially labeled cartesian products, and whose instances have equality decided structurally, e.g.,

```
AGGREGATE CLASS Money WITH
  amount : PositiveInteger;
  currency : { 'francs', 'ecu', ... };
END Money;
```

4. Entity classes have an associated extent: a set of objects with intrinsic, unchanging identity. The definition of entity classes specifies, among others, the attributes applicable to their instances, as well as their ranges. For example, the following definition describes the class of company entities, which have attributes *name*, *engagedIn* and *budget*:

```
ENTITY CLASS Companies WITH
  UNCHANGING, UNIQUE
  name : String;
  CHANGING
  engagedIn : SetOf Projects;
  budget : Money;
END Companies;
```

As the example shows, an attribute is specified to be either single-valued or set valued. Moreover, there is a small set of built-in primitive attribute *qualifiers*, which allow constraints to be placed on attribute values, and their evolution:

- UNCHANGING attributes cannot be modified, once assigned a value;
- CHANGING attributes, in contrast, may get different values as states change;
- UNIQUE attributes are required to have different values for all instances of the class (i.e., can act as “keys” in relational terms).

These qualifiers correspond to constraints that are frequently useful in the practice of data base system design. (Note also the contrast with Telos, where a user of the language may create arbitrary new attribute qualifiers in order to abbreviate constraints – this, because attributes are first class citizens in Telos.)

In addition, there exists an assertion language, to be described below, that permits the expression of predicates about objects and attribute values. These assertions can be used in three ways as part of describing a class.

- An assertion can be used to *define* a class intensionally through a predicate involving the special variable **this**, which is considered to range over the intersection of all the superclasses specified for the class. The following two examples define the integers between 1900 and 2000, and the subset of Companies with budget less than 100000 respectively:

*ORDINAL CLASS Years IS-A Integer WHERE (1900 < **this**) and (**this**< 2000)*
*ENTITY CLASS SmallCompanies IS-A Companies WHERE (**this**.budget.amount < 100000).*

TDL has the standard Pascal subrange notation as a convenient short form for the former: 1901..1999.

- Assertions can be used to specify sets, which in turn may be used to indicate how a single attribute value is related to the state of the data base. For example, the class *Projects* might have the *consortium* attribute specified as the inverse of the *Companies*' *engagedIn* attribute:

```
ENTITY CLASS Projects WITH
  UNCHANGING UNIQUE
    name : String;
    getsGrantFrom : Agencies;
  CHANGING
    consortium : SetOf Companies IS { each x in Companies : this isIn x.engagedIn };
END Projects;
```

Here, the special variable **this** refers to the (prototypical) instance of the class being defined – *Projects* in this case.

- The assertion language can also be used to specify more general integrity constraints, in the form of *Boolean-valued attributes* which are constrained to be true. For example, the following definition of class *Employees* requires that they work only on projects engaged in by their companies:

```
ENTITY CLASS Employees WITH
  UNCHANGING
    name : String;
  CHANGING
    belongsTo : Companies;
    worksOn : SetOf Projects;
  INVARIANT
    onEmpComp: True IS (this.worksOn subsetOf this.belongsTo.engagedIn);
END Employees;
```

Assertions about the “derivation” of attributes can also be expressed as declarative invariants. For example, the earlier description of the attribute *consortium* on class *Projects* is equivalent to

```
ENTITY CLASS Projects
  CHANGING
    consortium : SetOf Companies;
  INVARIANT
    onProjComp: True IS ( this.consortium = { each x in Companies : this isIn x.engagedIn } );
```

Boolean assertions may be qualified by 3 special attribute categories, referring to the time when the assertions are required to hold

- *INITIAL* assertions hold when an object becomes an instance of the class;
- *FINAL* assertions hold when an object ceases to be an instance of the class;
- *INVARIANT* assertions hold throughout the period when an object is a member of the class.

For example, the following attribute, attached to class *Projects*, might be used to require that every project have at least five participants at the beginning:

INITIAL

atLeastFive: True IS (*Count*(**this**.*consortium*) > 5) ;

Finally, TDL allows the specification of attributes which act as maps, in the sense that they take additional arguments before identifying a value. For example, Projects may break down their expenditures into sub-categories such as travel, equipment, and salary. This could be captured by having the *expenditures* attribute be a mapping which takes as argument an *ExpenseKind*, and produces a *MoneyValue*.

expenditures(*k*) : *ExpenseKinds* → *MoneyValue*

Note that as a specification language, no commitment is made here on how such attributes are to be stored (arrays, tables, relations) in the final system, nor in fact whether the information provided by the attribute will be obtained by look-up or by a function computing its answer.

It should come as no surprise that subclass hierarchies are supported and their use is encouraged throughout the design, according to the methodology described in [9]. In particular, subclasses (with multiple parents) can be defined, and subclasses can introduce new attributes, or refine conditions on old ones. In this paper we de-emphasize these by now familiar features of Taxis and its descendants.

3 The assertion language

The underlying data base state and description of a TDL database is captured by five predicates

InstanceOf(*<object>*, *<class>*)
HasAttr(*<object>*, *<attribId>*, *<value>*)
IsA(*<class>*, *<class>*)
HasDefAttr(*<class>*, *<attribId>*, *<class>*)
AttrCategory(*<class>*, *<attribId>*, *<category>*)

The first two predicates describe the database state – what are the instances of a class, and what values are related to some individual by an attribute. The last three provide access to the class descriptions themselves, including the subclass hierarchy, and the range classes specified for attributes on each class. We could have chosen as the assertion language for TDL the many-sorted First Order Logic based on these predicates, with class names being the sorts, and built-in operations on the base types, such as arithmetic, strings, etc.

We have experimented instead with a more “sugared” version of the language, which is based on set expressions. This was motivated by the several facts: attributes can now be multi-valued (unlike Taxis); DBPL has a powerful predicative language for manipulating sets of objects; and the methodology for refining specifications is based on Abrial’s rich repertoire of set-manipulation expressions [1][24]. The current expression language is then an amalgam of the functional notation for queries and DBPL’s quantifier structure, where attributes are treated as functions and classes as sets, denoting the extent of the class. The following examples illustrate the resulting language.

x.consortium – participants in project *x*

x.consortium.name – bag of names of all participants

(As in Daplex, sets are flattened before further attribute evaluation.)

{*x.consortium.name* **of each** *x* **in** *Projects*: *x.budget.amount* < 100000 **and** *x.budget.cur* = ‘ECU’}

– the name of every company participating in some project with a small budget

Count({**each** *z* **in** *Employee* : **all** *y* **in** *Projects* (*y* **isIn** *z.worksOn*)})

– the count of employees involved in every project.

4 Transactions and functions

As mentioned above, the state of the database is captured by the values of attributes, and the extents of classes. TDL provides for the definition of two kinds of procedures: *functions*, which are evaluated with respect to a single database state, and leave it unchanged while returning a value; and *transactions*, that specify atomic state transitions, where the final state is supposed to be free of inconsistencies. By their nature, functions may be used anywhere expressions may occur, including the definition of attributes, sets, etc. for data classes. On the other hand, as we shall see, transactions are normally expected to be associated with transitions of scripts, including possibly on-line invocation by users.

4.1 Functions

Functions may take several parameters as arguments and return a value. In keeping with Taxis' original view, most things are viewed as attributes, possibly qualified by attribute categories. Thus function parameters and the return value are seen as attributes of the “function object”, and are typed in the same manner as class attributes:

```
FUNCTION Plus(x:Integer; y:Integer) WITH  
  RETURNS answer : Integer; ...
```

is short form for

```
FUNCTION Plus WITH  
  IN  
    x, y : Integer;  
  RETURNS  
    answer : Integer; ...
```

The return attribute value can be specified either by the use of a “derivation expression”, as in

```
RETURNS  
  answer : Integer IS x + y;
```

or through a general boolean assertion under the category GOALS:

```
RETURNS  
  answer : Integer;  
GOALS  
  (answer * answer = x + y);
```

The former is essentially an executable specification, while the latter provides a non-effective description of the value to be returned. In general, we believe that in the design of Information Systems there is little need for such complex specifications, and have restricted our attention to the former kind.

Functions may specify “local variables”, which of course are listed as attributes under the attribute category LOCAL. As an illustration of some of the features of TDL introduced so far, we consider a useful library of class and function definitions – those involving time.

4.2 Some useful examples involving time

The following special classes are assumed to be predefined in TDL in order to facilitate the expression of conditions involving time. Generally, there are two distinct types of temporal expressions: ones involving time points, the others involving time intervals.

The following special classes are assumed to be available:

```
AGGREGATE CLASS $TimePoint.
```

```
AGGREGATE CLASS $Date IS-A $TimePoint WITH
  year : 1900 .. 2000;
  month: 0..12;
  day: 0..31;
  WHERE not (month=2 and day>29) and not (month=4 and day=31)
  and ... and not (month=0 and day ≠ 0)
END Date;
```

```
AGGREGATE CLASS $Daytime IS-A $Date WITH
  hour : 1..24;
  minute : 0 ..60;
  second : 0 .. 60;
  WHERE not (minute=0 and second ≠ 0)
END $Daytime;
```

Dates are the standard calendar dates, and are available in two versions, where the first one requires less precision: `$Date[year: 1987, month:7, day: 21]`, and `$Daytime[year: 1987, month:7, day: 21, hour:12, minute:30, second:0]`. The special variables **\$today** and **\$now** are instances of `$Date` and `$Daytime` respectively, and are assumed to represent the current date and clock time at all times.

Values of 0 for *month*, *day*, *minute* and *second* will be used to signify "don't care", and are used by the *Before* and *After* predicates (boolean functions) which compare two time points. Thus `$Date` is really just `$Daytime` with hour, minute and second set to 0. The functions *Before* and *After* can be specified in TDL; here is the beginning of one such specification:

```
FUNCTION Before (t1: Date, t2: Date) WITH
  RETURNS
  answer : Boolean IS (t1.year < t2.year) or ((t1.year=t2.year) and
    ((t1.month < t2.month and t1.month ≠ 0 and t2.month ≠ 0) or
    (t1.month = t2.month and ... )))
```

Although *After* is similarly defined, note that it is possible for some time b to be neither before nor after some time c, thus signifying that the time points are specified to different degrees of precision and one includes the other.

Temporal durations can be specified in many ways, one common technique being temporal lengths ("floating intervals"), which are subclasses of general measures:

```
AGGREGATE CLASS $TemporalDuration;
```

```
AGGREGATE CLASS $Measure WITH
  value : PositiveInteger;
  unit: Anything;
```

END \$Measure;

AGGREGATE CLASS \$TimeLength IS-A \$TemporalDuration, \$Measure WITH
unit: {'years, 'months, 'days, 'hours, 'minutes, 'seconds};
END \$TimeLength;

An interval of two hours would then normally be expressed as $\$TimeLength[value:2, unit: 'hour]$.² Similarly, one can define functions such as *Plus* and *Minus*, overloaded to add time lengths to time points (e.g., *Plus(\$now, 2'minutes)*).

In general, a designer is free to extend the temporal system provided in the standard library. One might thus obtain biblical calendars, or temporal durations defined by time points.

4.3 Transactions

Transactions are state transitions, and their parameters appear as attributes with the property categories IN and OUT indicating whether the parameter is used to communicate information into or out of the transaction. Local variables can be introduced to abbreviate expressions, using the same style of attribute specification:

TRANSACTION HireEmployee WITH
IN
name : String;
belongs : Companies;
works : SetOf Project;
OUT
e : Employee; ...

A specification is intended to associate with every transaction a set of acceptable state transitions. This can be accomplished in a variety of ways. One standard approach is to assert constraints on the values of the state variables (attribute functions, parameters and class extents) relating the initial and final states. This is the familiar precondition/postcondition notation introduced by Hoare, and widely used in such specification languages as Z [21][14] and the Larch language families [13].

The assertions are expressed in the usual Taxis-like notation, under two attribute categories

- GIVEN: conditions required to hold in the initial state when the transaction is invoked;
- GOALS: conditions required to hold in the final state.

Both conditions are logical assertions, with goals able to relate values in both the initial and final state. For this purpose, we use the standard technique of distinguishing initial and final state values of variables by marking with the suffix ' the values to be evaluated in the final state. For example

GOALS $x.budget' = x.budget + Count(y.participants) * 1000$

The insertion and removal of objects from the extents of classes will be expressed through special predicates **Added** and **Removed** that can appear in GOAL conditions: e.g., if p is a Person, then **Added** ($p, Employees$) holds in the final state if p is added to the initial set of instances of Employees, and its

²If there is sufficient demand, *all measures* could be syntactically sugared to the form *value'unit* (e.g., *2'hours* for 2 hours, *2200'FF* for 2200 French Francs).

superclasses. In order to indicate that a new object needs to be created, we must use the special predicate **New**: for example, **New**(x) AND **Added** ($x, Employees$). To indicate that an object is to be deleted, we simply remove it from the topmost class, Anything, in the hierarchy: **Removed** ($e, Anything$). This removes it from all subclasses as well.

Since many simple transactions result in the creation and deletion of new objects, an alternative shorter syntax is provided for this: in the spirit of the TDL notation of attribute specifications, two attribute categories, PRODUCES and CONSUMES, assert that the attribute value is (is not, resp.) an instance of the class specifying its range. Thus

CONSUMES e : Employee

asserts that e will not be an instance of class Employee in the final state. The value of e might be specified in several ways: i) as an IN parameter of the transaction; ii) as a derivation expression of the kind previously encountered

CONSUMES e : Employee IS the {Employee: name= "Aristotle"}

iii) through goal assertions about the properties of the object.

Therefore, the full specification of the HireEmployee transaction could be

TRANSACTION HireEmployee WITH
IN
name : String; belongs : Companies; works : SetOf Projects;
OUT , PRODUCES
e : Employees;
GOAL
(e.name' = name) and (e.worksOn' = works) and (e.belongsTo' = belongs);

In addition, we have chosen to distinguish certain pre- and post-conditions by calling them INITIAL and FINAL conditions. These conditions prevent the transaction from completing unless they are satisfied, and they play two roles:

- they allow the early detection of cases where updates would result in the violation of integrity constraints stated on classes (e.g., spending more money than the budget allows);
- they allow the specification and checking of *dynamic integrity constraints* (e.g., ages do not decrease).

INITIAL/FINAL conditions are distinguished from GIVEN/GOAL conditions by the following criteria:

- INITIAL and FINAL conditions are “run-time” test obligations to be met at transaction entry and commit time; the failure of such a condition corresponds to an exception condition in TDL, and some additional error or special case handling code is specified to deal with this. These conditions therefore support the “normal-case first” abstraction [6].
- GIVENS are conditions which are expected to be satisfied by the caller of the transaction, so there is no need to have code checking such conditions.
- GOALS are conditions which are expected to be established by the body of the eventual procedure implementing this transaction³.

³In a formal view of program derivation, GIVENS and GOALS are therefore proof obligations.

- All class identifiers need to be defined.
- All enumerated values need to appear in the definition of some enumerated class.
- The subclass hierarchy needs to be acyclic.
- If a subclass B of class C, specifies that attribute p has range R, then if C (or its superclasses) also mention p, then whatever range S is specified for p on C, must be a superclass of R (i.e., we have “proper/strict” specialization).
- In computing the “subclass” relationship, we need to consider that there is an implicit such relationship between subranges like 3..5 and 2..6, and enumerations such as {‘FF’} and {‘FF’, ‘DM’}.

We have also implemented in Prolog a type checker which considers TDL expressions and issues warnings wherever it cannot verify that no type errors occur in it: i.e., no function is applied to arguments outside its specified domain, and no attribute is applied to an object which does not belong to a class that assures that it has those attributes. The functions recognized include the built in arithmetic, set and boolean operations. As usual, the type checker essentially carries along an ‘environment’ describing the type of the variables introduced either as parameters/locals or by the quantifiers in expressions. This, together with the class definitions available, is used to deduce type(s) for attribute expressions involving the dot operator: *x.budget*. The type checker also considers the complications introduced by the fact that some attributes/expressions are set valued, while others are single valued.

5.2 Executing TDL specifications

Another important software engineering capability is the ability to prototype an information system at the design level. This means that TDL design specifications should be executable with a minimum of additional effort. In turn, this means that there must be ways to execute a transaction or script for particular arguments by finding ways to satisfy the goals of a transaction or to execute the transitions of a script. There seem to be three (compatible) alternatives to make transactions executable in the context of a design environment:

1. The TDL interpreter finds on its own ways to achieve the goals of a transaction.
2. The user provides his/her own temporary code for a transaction, thus giving his own way of achieving the goals of that transaction or transition. This code need not be used in an efficient implementation of the transaction or transition.
3. The user is asked to effect interactively changes that achieve the goals of a transaction or transition. Once the user is done with such changes, the executor checks that the goal assertions have indeed been achieved.

The most attractive alternative is of course the first one. Unfortunately, it can only be pursued in a subclass of all specifications, since in its most general form, it would require full automatic programming. Our current efforts are directed towards translating those TDL specifications where GOAL assertions are of the form $e' = f$, where all values in f are from the initial state, and e' is a simple attribute chain expression concerning the final value of some attribute (e.g., $x.salary' = x.salary * 1.2$). Such expressions are being translated into a special language called Probe, developed at BIM, Belgium – one of our partners in the Daida project. Probe is a language which can be executed through the use of a meta-interpreter in Prolog, and it contains constructs that resemble a domain-relational calculus expression, with primitive predicates involving class/set membership, subset relationships, arithmetic operations and comparators. As a result, a user could populate a Prolog database with a small number of individuals, and then execute one of the transactions which has been translated into Probe. This interaction takes place through a spread-sheet style/menu driven interface, rather than direct Prolog function invocations [16].

5.3 Refining TDL designs

Within the DAIDA project, a major effort concentrates on the production of high-quality database application software. In DAIDA we make the following assumptions about this software production process:

- for a given TDL-design there are many substantially different DBPL-implementations, and it needs human interaction to make and justify decisions that lead to efficient implementations;
- the decisions are too complex to be made all at once; it needs a series of refinement steps referring to both data and procedural objects;
- the objects and decisions relevant for the refinement process need to have formally assured properties that should be recorded to support overall product consistency and evolution.

To meet the above requirements DAIDA relies heavily on current work of J.-R. Abrial [3], [2]:

- TDL designs are translated into his notion of Abstract Machines with states represented by mathematical objects like functions and sets, and state transitions defined by Generalized Substitutions;
- the formal properties of Machines and Substitutions are assured and organized by Abrial's interactive proof assistant, the B-Tool [4].

6 TDL designs as a basis for high-quality software production

In the remainder of the paper we report on our first experience with a rule-based Mapping Assistant that helps refining TDL-designs into DBPL-implementations via Abstract Machines. First we outline the formal framework, and then we discuss in some detail the refinement process itself.

For an example we use part of the TDL-design introduced earlier. ResearchCompanies consists of three entity class definitions (Companies, Employees, Projects) and one transaction definition (HireEmployee). One of the invariants asserts that the projects a company is engaged in are exactly those which have this company as a consortium member. Another invariant enforces the projects on which an employee works to be a subset of the projects his company is engaged in. For some of the classes identifying attributes (attribute category UNIQUE) are specified. The class Agencies is defined as an enumerated class of identifiers (see Appendix, Fig.1).

6.1 On abstract machines and Generalized Substitution

Abrial's framework for specifying and developing software systems is based on the notion of Abstract Machines, a concept for organizing large specifications into independent parts with well defined interfaces [3]. An Abstract Machine is considered to be a general model for software systems and is defined by a state (the statics of the system) and a set of operations modifying the state (the dynamics).

The *statics of a system* includes the definition of three main Abstract Machine components: Basic Sets, Variables, and system Invariants.

Basic sets define time-invariant repositories of objects (types) that may occur in our application. In our first version, MACHINE researchCompanies.1 (see Appendix, Fig.2), they are completely abstract, defined only by their name.

BASIC SETS Agencies, Companies, Employees, . . . , EmpNames, . . .

The Context allows us to freeze the definition of basic sets as soon as we made a decision on their properties. In our TDL design, for example, we decided already on the representation of certain constants and names, and we would like to stick to that decision:

CONTEXT $Agencies = \{ESPRIT, DFG, NSF, \dots\}, \dots, EmpNames = Strings, \dots$

Variables are either time-dependent subsets of the Basic Sets and model the state of an Abstract Machine, or they are binary relations that associate elements of Basic Sets (these are normally viewed as functions from the domain to the range) :

VARIABLES $companies, employees, \dots, empName, belongsTo, \dots$

Finally, the Invariants express the static relationships within the system. They are defined in terms of predicates and sets (with a rich choice of operators for set, relation, and function (re-) definition).

INVARIANTS $employees \in \wp (Employees); \dots; worksOn \in (employees \rightarrow \wp (projects));$
 $\forall x. x \in employees \Rightarrow worksOn(x) \subseteq engagedIn(belongsTo(x)); \dots$

The *dynamics of a system* are defined by the Operations of an Abstract Machine. The definition of an operation, e.g. hireEmployee, contains all the constraints relevant for a state transition. Operations are specified by the constructs of the Generalized Substitution Calculus. The semantics of these constructs are defined by the weakest precondition under which the construct fulfils a given postcondition.

OPERATIONS $hireEmployee (name, belongs, works) =$
 $PRE name \in EmpNames \wedge belongs \in companies \wedge works \in \wp (projects)$
 $THEN ANY e IN (Employees - employees)$
 $THEN$
 $(empName(e), belongsTo(e), worksOn(e)) \leftarrow (name, belongs, works) ||$
 $employees \leftarrow employees \cup \{e\} || hireEmployee \leftarrow e$
 END
 $END hireEmployee;$

In our example, the ANY-substitution expresses an unbounded non-deterministic substitution. It chooses an arbitrary fresh member from $(Employees - employees)$, i.e., from the set of non-instantiated elements considered for employee representation. Next, the (attribute-) functions for that new employee element are re-defined in parallel⁵. The "type conditions" on the input parameters are expressed by the preconditions of the Generalized Substitution. In the subsequent sections we discuss some details of the proofs we are obliged to carry out to show that operations and invariants are consistent, and we outline the proof support we get from the B-Tool.

6.2 Proof organisation and formal proof support

The B-Tool is a general proof assistant and uses a goal-oriented proof technique. The built-in logic performs simple and multiple substitution and is designed to suit best Abrial's Calculus of Generalized Substitutions.

For the practical work with the B-Tool it is essential that proofs are organized such that any unreducible goal (e.g., in lack of appropriate theories or in case of not provable predicates) are generated as lemmata. The proof of a lemma can be postponed, and the initial proof can go on. Typical examples of such delayed proof obligations refer to lemmata that express type assertions or arithmetic properties. Therefore, as a result of a proof, some lemmata may be left over which, in a further step, may be proven by the Tool using additional theories (or by a less formal but convincing argument).

⁵The operator, \leftarrow , means textual substitution, and $func(x) \leftarrow y$ means $func \leftarrow func \oplus \{(x,y)\}$

6.3 Consistency of specifications

On our initial Abstract Machine we are in a position to prove the consistency of our specification. In general, we have to prove for each operation that it preserves the invariants of the Machine. In our example the consistency proof ends up in a number of lemmata out of which one is of particular interest since it is not provable. This lemma refers to the project an employees's company is engaged in and the ones he works on. A simplified version looks as follows:

$$(name \in EmpNames \wedge belongs \in companies \wedge works \in \wp(projects) \wedge e \in (Employees - employees)) \\ \Rightarrow (worksOn \oplus \{e \rightarrow works\})(e) \subseteq engagedIn((belongsTo \oplus \{\epsilon \rightarrow belongs\})(e));$$

which can be simplified further to

$$belongs \in companies \wedge works \in \wp(projects) \Rightarrow works \subseteq (engagedIn(belongs)).$$

By realizing that this lemma is not provable we have found a deficiency in our TDL-design and we have to reformulate our specification. The initial precondition of the transaction HireEmployee (see section 4.3) turns out to be too weak because we simply overlooked the fact that the company and the projects of a new employee are related.

This leads to a change in our TDL-design: transaction, HireEmployee, requires the precondition *GIVEN works subsetOf belongs.engagedIn*. Consequently, the precondition of our initial Abstract Machine, has to be strengthened from $works \in \wp(projects)$ to $works \in \wp(engagedIn(belongs))$.

7 From TDL designs to DBPL implementations

In our setting, refinement is defined between Abstract Machines having the same signature, i.e., the same number of operations and parameters. Some Machine, A (the more abstract one), is said to be refined by a Machine C (the more concrete one) iff the refinement predicate, P_{AC} , that relates the state variables of both machines is proven to be an invariant under all operations, S_A and S_C , of both machines, A and C:

$$I_A \wedge P_{AC} \wedge terminate(S_A) \Rightarrow [S_C]\{S_A\}P_{AC}.$$

The proof obligation also considers the case of a non-deterministic operation in machine A and a less non-deterministic one in machine C. The substitution using curly brackets, $\{S\}R$, is an abbreviation for the substitution, $\neg[S]\neg R$, which means that there is at least one state being reachable after S_A and one after S_C such that both fulfil the refinement predicate P_{AC} .

7.1 On data and procedure refinement

A mapping process usually consists of a series of procedure- and data-oriented refinement steps from one Abstract Machine into another. The refinement process can be directed interactively by the user and controlled formally by the proof assistant.

Data refinement is defined in two steps:

- a data representation has to be chosen that is "nearer" to the concrete data structures being offered by the target programming language that is used to implement the specification;

- the relationship between the more concrete state space and the more abstract one has to be expressed by a refinement predicate that involves variables of both machines.

Usually, changes in data representations imply refinements of operations. An operation S_A is refined by an operation S_C "if S_C can be used instead of S_A without the 'user' noticing it" [3]. Which means more formally: all postconditions R that are established by S_A are also established by S_C .

Two cases are possible:

- S_C can be less non-deterministic than S_A , or
- for each postcondition R the weakest precondition of S_C to establish R is weaker than the one for S_A (so S_C might terminate in states where S_A doesn't).

Within the framework of Generalized Substitutions algorithmic refinement is formally defined as a partial order relation between substitutions [3].

The *first refinement* step in our example, MACHINE researchCompany.2, (see Appendix Fig.3) addresses the basic issues of *data identification*. It is a rather specific decision to utilize the uniqueness constraints on properties for companies and projects for data identification. For employees, where the application doesn't provide such a constraint, we introduce an additional property, empId, for which the implementation assures uniqueness. The specification of the source for empId is left open, however, its type predicate guarantees already a fresh value each time the operation is invoked.

For our refined Abstract Machine, the above decisions imply an additional basic set, EmpIds, and an extended

CONTEXT Companies = CompNames; Employees = EmpIds; Projects = ProjNames \otimes Agencies; ...

The following definitions contribute to the refinement predicate and relate the refined representations to the previous ones:

DEFINITIONS compName = $\lambda x. (x \in \text{companies} \mid x)$; empId = $\lambda x. (x \in \text{employees} \mid x)$; ...

These definitions determine the function, compName, to be the identity function over the new representation of companies and the newly defined function, empId, to be represented by the elements in employees.

Due to the change in the data representation the operation, hireEmployee, becomes less non-deterministic: the arbitrary ANY-substitution is replaced by some operation, newEmpId, that provides a fresh EmpId-value, which is then associated with the required properties through function extension. Our refined data and procedure representation can be proven to meet all the invariants layed down in the initial Abstract Machine.

In the subsequent section we discuss further refinements that lead step by step towards a relational implementation.

7.2 Refinement process and alternatives

Having decided upon identification we are now ready for the *second refinement* step that designs the central *data structures* of our implementation. While our previous Abstract Machines have a purely functional view on data, MACHINE researchCompanies.3 (see Appendix Fig.4), refines version 2 into a representation that is based on Cartesian products. The newly introduced variable, e.g., empClass, becomes a total function that associates employees with structured data defined over EmpName, companies, and sets of projects.

INVARIANTS $empClass \in (employees \rightarrow EmpName \otimes companies \otimes \wp (projects)); \dots$

The following definitions determine the three functions from above to be represented by the three corresponding components returned by the single function, `empClass`.

DEFINITIONS $(empName, belongsTo, worksOn) = \lambda x. (x \in employees \mid empClass(x)); \dots$

Another change of representation is the introduction of a variable, `tEmpID`, that allows us to replace the parallel substitutions in the operation, `hireEmployee`, by serial ones.

Notice that in `researchCompanies.3` functions are set-valued. An alternative refinement could introduce “flat variables”:

VARIABLES $flatCompClass, flatEmpClass, flatProjClass, empProjClass, \dots;$

with constraints like

INVARIANTS $flatEmpClass \in (employees \rightarrow EmpName \otimes companies); \dots$
 $empProjClass \in (employees \leftrightarrow projects); \dots;$

The flat representation has, of course, consequences for the operation, `hireEmployee`, which has to contain substitutions on both variables, `flatEmpClass` and `empProjClass`:

OPERATIONS $hireEmployee(name,belongs,works) = \dots;$
 $flatEmpClass \leftarrow flatEmpClass \oplus \{(tEmpId, name, belongs)\} \parallel$
 $empProjClass \leftarrow empProjClass \oplus inverse(\lambda x.(x \in works \mid (tEmpId))) \dots$

On our way down to a relational implementation there is a *third refinement* step left that deals with *data typing*. Since DBPL is a strongly and statically typed database programming language we want to refine the variables to become partial functions over the Basic Sets instead of total functions over other variables of time-varying cardinality. This refinement step leads to MACHINE `researchCompanies.3` (see Appendix, Fig.5) with

VARIABLES $compRel, empRel, projRel$

constrained by

INVARIANTS $compRel \in CompNames \mapsto \wp (ProjNames \otimes Agencies); compClass = compRel;$
 $empRel \in EmpIds \mapsto EmpNames \otimes CompNames \otimes \wp (ProjNames \otimes Agencies); empClass = empRel;$
 $projRel \in (ProjNames \otimes Agencies) \mapsto \wp (CompNames); projClass = projRel; \dots$

In the subsequent section we will see how these invariants can be transformed into the type definitions (or schema) of a DBPL database module.

Similarly, the precondition of the hire operation is weakened into a static constraint that will finally become the parameter types of the `hireEmployee` transaction. To imply, however, the inherited specification we have to strengthen our constraints by a *conditional* substitution:

IF $belongs \in companies \wedge works \in \wp (engagedIn(belongs))$ THEN ... ELSE ...

Due to the semantics of the Generalized Substitution the conditional will finally result in a first-order predicate on the variables that represent database states and transaction parameters.

Our example also demonstrates that a refined version may have a weaker precondition than the initial one: hireEmployee is now defined in all cases in which the static type predicate holds (a condition which can already be verified at compile time). It either performs the required state transition or it returns as an exception a specific value, nilEmpId.

7.3 The final transformation: from abstract machines into DBPL implementations

By our three refinement steps we have decided upon three major tasks in data modelling: identification of data in extents of varying cardinalities, alternative ways of data structuring, and introduction of type constraints for variables, parameters, etc. that can be checked efficiently and at appropriate times. What remains to be done is a transformation of a final Abstract Machine into an equivalent DBPL program, MODULE ResearchCompanies (see Appendix, Fig.6).

The decisions for value-based identification and for sets and Cartesian products as basic data structures were motivated, of course, by a relational implementation language.

A language like DBPL with a rich typing system and with first-order calculus expressions for set evaluation and control turns out to be an appropriate target for the implementation of sufficiently refined Abstract Machines. Informally speaking, we can translate those machines into equivalent DBPL programs which are based on variables constrained by static type predicates and on operations defined by deterministic and sequential Generalized Substitutions.

Invariants and definitions, e.g., of our final abstract machine variable empRel, translate one-to-one into the type definitions of the corresponding DBPL variable, employeeRel.

In the final refinement step of hireEmployee the state-dependent precondition is transformed into a static one plus a conditional substitution. The static condition is transformed into DBPL parameter types, and the conditional substitution results in a database update statement controlled by a first-order database query expression:

```
IF SOME c IN companyRel (c.name = belongs) AND
  ALL w IN works (SOME p IN companyRel[belongs].engagedIn (w = p))
THEN tEmpId := Identifier.New( ... );
  employeeRel := employeeRelType{{tEmpId,name,belongs,works}};
  RETURN tEmpId;
ELSE RETURN Identifier.Nil; END
```

Here we can see how the DBPL expression based on first-order predicates correspond nicely with Abrial's Generalized Substitutions.

So far, we concentrated on the *algorithmic and structural part* of database application semantics. We ignored completely the *operational* aspects, such as the demand for concurrency of database operations, for persistence of database states, for failure recovery, access efficiency etc. While an extensive treatment of these issues goes beyond the scope of this paper we would like to close this discussion with a short reference to concurrency and persistence in DBPL.

In most realistic data-intensive applications operations are not issued one after another, and *serialization* is not an acceptable scheduling strategy. Instead, *serializability* of parallel database operations is the commonly agreed execution model. In our example, the THEN-branch of hireEmployee requires the IF-condition to be maintained as true. Otherwise, the string values of the parameters, belongs and works, may

not represent companies and projects, and the database update operation does not meet its specification. Parallel execution of hireEmployee and say, cancelProject, would lead to inconsistent result. On the other hand, if a parallel execution, e.g., startProject, changes the IF-condition from false to true, the result of the ELSE-branch is acceptable since it is the same as the result produced by the sequence, hireEmployee first and then startProject.

DBPL has transactions as first-class language construct which provide, besides functional abstraction, serializability and failure recovery. Another central requirement of data-intensive applications, i.e., the persistence of states across program executions, is achieved by declaring the corresponding variables inside the scope of persistent database modules. A first impression of the overall software engineering quality of DBPL implementations is given by the complete DBPL example program presented in the Appendix.

8 Summary

We have presented, rather sketchily, some of the highlights of two languages, TDL and DBPL for the conceptual design and the implementation of an information systems.

TDL is based on the language Taxis, but it has been refurbished to permit the predicative description of procedures (transactions, functions, script transitions), integrity constraints and coordinated transaction groups (scripts). TDL's expression language, unlike Taxis, is based on set theoretic notions. TDL maintains an object-oriented approach, thus allowing the designer who maps Telos requirements into TDL designs to concern herself mostly with global issues such as view integration and the elimination of omni-present temporal indices.

On the other hand, TDL adopts the standard (destructive) state based model of computation of most procedural languages, and thus the mapping to DBPL programs can concentrate on the appropriate choices for data identification, structuring and typing and corresponding refinement steps on transactions and functions.

As argued in the previous section, Abrial's methodology of refining abstract machines appears to be a promising direction in which to study the issue of this implementation in a provably correct manner. In this setting, a language like DBPL with a set- and predicate-oriented approach to data modelling, a rich typing system, and abstract support for concurrency, recovery and persistence, turns out to be an appropriate target for the refinement of TDL designs into database application software.

Acknowledgement: We would like to thank our DAIDA colleagues, in particular F. Fühler, F. Matthes, and M. Mertikas, for their contributions to this research.

Appendix: From TDL designs through Abstract Machines to DBPL implementations

```
TDLDESIGN ResearchCompanies IS
ENUMERATED CLASS Agencies = {'ESPRIT', 'DFG', 'NSF', ...};
ENTITY CLASS Companies WITH
    UNIQUE, UNCHANGING name : Strings;
    CHANGING engagedIn : SetOf Projects;
```

```

END Companies;
CLASS Employees WITH
  UNIQUE, 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 HireEmployee WITH
  IN name : Strings; belongs : Companies; works : SetOf Project;
  OUT, PRODUCES e : Employee;
  GIVEN works subsetOf belongs.engagedIn;
  GOALS (e.name = name) and (e.worksOn = works) and (e.belongsTo = belongs)
END HireEmployee
END ResearchCompanies;

```

Fig. 1: TDL Design of ResearchCompanies example

```

MACHINE researchCompanies.1
BASIC SETS Agencies, Companies, Employees, Projects, CompNames, EmpNames, ProjNames
CONTEXT Agencies = { ESPRIT, DFG, NSF, ... }, CompNames, EmpNames, ProjNames = Strings
VARIABLES companies, compName, engagedIn,
  employees, empName, belongsTo, worksOn,
  projects, projName, getsGrantFrom, consortium
INVARIANTS companies ∈ φ(Companies);
  compName ∈ (companies → CompNames);
  engagedIn ∈ (companies → φ(projects));
  employees ∈ φ (Employees);
  empName ∈ (employees → EmpNames);
  belongsTo ∈ (employees → companies);
  worksOn ∈ (employees → φ (projects));
  projects ∈ φ(Projects);
  projName ∈ (projects → ProjNames);
  getsGrantFrom ∈ (projects → Agencies);
  consortium ∈ (projects → φ(companies));
  ∀x,y. x,y ∈ companies ⇒ (compName(x) = compName(y) ⇒ x = y);
  ∀x. x ∈ employees ⇒ (worksOn(x) ⊆ engagedIn(belongsTo(x)));
  ∀x,y. x,y ∈ projects ⇒ (projName(x) = projName(y) ∧ getsGrantsFrom(x)=getsGrantsFrom(y) ⇒ x = y);
  ∀x. x ∈ projects ⇒ (consortium(x) = {y | y ∈ companies ∧ x ∈ worksOn(y)});
OPERATIONS hireEmployee (name, belongs, works) =
  PRE name ∈ EmpNames ∧ belongs ∈ companies ∧ works ∈ φ (engagedIn(belongs))
  THEN ANY e IN (Employees - employees)
    THEN (empName(e), worksOn(e), belongsTo(e)) |← (name, works, belongs) ||
      employees |← employees ∪ {e} || hireEmployee |← e
    END
  END hireEmployee;
End researchCompanies.1;

```

Fig. 2: Initial Abstract Machine

```

MACHINE researchCompanies.2
IMPLY researchCompanies.1
BASIC SETS EmpIds
CONTEXT Companies = CompNames; Employees = EmpIds; Projects = ProjNames ⊗ Agencies
VARIABLES empId
INVARIANTS empId ∈ (employees → EmpIds);
DEFINITIONS compName = λx.(x ∈ companies | x);
  empId = λx.(x ∈ employees | x);
  (projName, getsGrantFrom) = λx.(x ∈ projects | x);
OPERATIONS hireEmployee (name, belongs, works) =
  PRE name ∈ EmpNames ∧ belongs ∈ companies ∧ works ∈ φ (engagedIn(belongs))
  THEN
    (empName(newEmpId), worksOn(newEmpId), belongsTo(newEmpId)) |← (name, works, belongs) ||
    employees |← employees ∪ {newEmpId} || hireEmployees |← newEmpId
  END hireEmployee;
OTHER newEmpId ∈ (→ EmpIds - employees);
END researchCompany.2;

```

Fig. 3: Abstract Machine with identification refinement

```

MACHINE researchCompanies.3
  IMPLY researchCompanies.2
  VARIABLES compClass, empClass, projClass, tEmpId
  INVARIANTS compClass ∈ companies → ∅ (projects);
    empClass ∈ employees → empName ⊗ companies ⊗ ∅ (projects);
    projClass ∈ projects → ∅ (companies);
    tEmpId ∈ EmpIds; ...
  DEFINITIONS engagedIn = λx. (x ∈ companies | compClass(x));
    (empName, belongsTo, worksOn) = λx. (x ∈ employees | empClass(x));
    consortium = λx. (x ∈ projects | projClass(x));
  OPERATIONS hireEmployee (name, belongs, works) =
    PRE name ∈ EmpNames ∧ belongs ∈ companies ∧ works ∈ ∅ (engagedIn(belongs))
    THEN tEmpId ← newEmpId;
      empClass ← empClass ∪ {(tEmpId, name, belongs, works)};
      hireEmployee ← tEmpId
    END
  END hireEmployee;
  OTHER newEmpId ∈ ( → (EmpIds - employees));
END researchCompany.3;

```

Fig. 4: Abstract Machine with data structure refinement

```

MACHINE researchCompanies.4
  IMPLY researchCompanies.3
  VARIABLES compRel, empRel, projRel
  INVARIANTS compRel ∈ CompNames +→ ∅ (ProjNames ⊗ Agencies); compRel = compClass ;
    empRel ∈ EmpIds +→
      EmpNames ⊗ CompNames ⊗ ∅ (ProjNames ⊗ Agencies); empRel = empClass ;
    projRel ∈ (ProjNames ⊗ Agencies) +→ ∅ (CompNames); projRel = projClass ; ...
  OPERATIONS HireEmployee (name, belongs, works) =
    PRE name ∈ EmpNames ∧ belongs ∈ CompNames ∧ works ∈ ∅ (ProjNames ⊗ Agencies)
    THEN IF belongs ∈ companies ∧ works ∈ ∅ (engagedIn(belongs))
      THEN tEmpId ← newEmpId;
        empRel ← empRel ∪ {(tEmpId, name, belongs, works)};
        hireEmployee ← tEmpId
      ELSE hireEmployee ← nilEmpId
    END
  END HireEmployee;
  OTHER newEmpID ∈ ( → (EmpIds - employees));
END researchCompany.4;

```

Fig. 5: Final Abstract Machine with type predicates

```

DEFINITION MODULE ResearchCompaniesTypes;
  IMPORT Identifier,String;
  TYPE
    Agencies = (ESPRIT, DFG, NSF, ..);
    CompNames, EmpNames,ProjNames = String.Type;
    EmpIds = Identifier.Type;
    CompIdRelType = RELATION OF CompNames;
    ProjectIdRecType = RECORD name : ProjNames; getsGrantFrom : Agencies END;
    ProjectIdRelType = RELATION OF ProjectIdRecType;
    CompanyRecType = RECORD name : CompNames; engagedIn : ProjectIdRelType END;
    CompanyRelType = RELATION name OF CompanyRecType;
    EmployeeRecType = RECORD employee : EmpIds; name : EmpNames;
                        belongsTo : CompNames; worksOn : ProjectIdRelType END;
    EmployeeRelType = RELATION employee OF EmployeeRecType;
    ProjectRecType = RECORD name : ProjNames; getsGrantFrom : Agencies;
                        consortium : CompIdRelType END;
    ProjectRelType = RELATION name, getsGrantsFrom OF ProjectRecType;
END ResearchCompaniesTypes;

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

IMPLEMENTATION MODULE ResearchCompaniesOps;
  FROM ResearchCompaniesTypes
  IMPORT EmployeeRelType; ProjectRelType; CompanyRelType;
  IMPORT Identifier;
  VAR employeeRel : EmployeeRelType;
      projectRel : ProjectRelType;
      companyRel : CompanyRelType;
  TRANSACTION hireEmployee (name:EmpNames;belongs:CompNames;
                          works:ProjIdRelType) : EmpIds;

  VAR tEmpId : EmpIds;
  BEGIN
    IF SOME c IN companyRel (c.name = belongs) AND
      ALL w IN works (SOME p IN companyRel[belongs].engagedIn(w = p))
    THEN tEmpId := Identifier.New( ... );
      employeeRel :+ employeeRelType{{ tEmpId,name,belongs,works}};
      RETURN tEmpId
    ELSE Identifier.Nil
    END
  END hireEmployee;
END ResearchCompaniesOps

```

Fig. 6: DBPL implementation of the ResearchCompanies example

References

- [1] Abrial, J. R. , “Introduction to Set Notations”, Parts 1,2, 26 Rue des Plantes, Paris 75014, June 1988.
- [2] Abrial, J. R., “Formal Construction of an Order System”, 26 Rue des Plantes, Paris 75014, May 1988.
- [3] Abrial, J. R., P. Gardiner, C. Morgan, and M. Spivey, “Abstract Machines”, Part1 - Part4, 26 Rue des Plantes, Paris 75014 June 1988.
- [4] Abrial, J. R., C. Morgan, M. Spivey, and T.N. Vickers, “The Logic of ‘B’ ”, 26 Rue des Plantes, Paris 75014, September 1988.
- [5] J. Barron, “Dialogue and process design for Interactive Information Systems”, ACM SIGOA Conf. Proceedings, Philadelphia, June 1982.
- [6] Borgida, A., “Language Features for Flexible Handling of Exceptions”, ACM Trans. on Database Systems 10(4), pp.565-603.
- [7] Borgida, A., Jarke, M., Mylopoulos, J., Schmidt, J., and Vassiliou, Y., “The software development environment as a knowledge base management system”. In Schmidt, J.W., Thanos, C. (eds) Foundations of Knowledge Base Management, Springer Verlag, (to appear 1989)
- [8] Borgida, A., Greenspan, S., and Mylopoulos, J., “Knowledge representation as a basis for requirements”, IEEE Computer Vol.18, No.4., April 1985, pp.82-103.
- [9] Borgida, A., J.Mylopoulos, and H.K.T.Wong, “Generalization/Specialization as a basis for software specification”, in On Conceptual Modeling, M.Brodie et al. (eds.), Springer Verlag, 1984, pp. 87-114.
- [10] Borgida, A., M. Koubarakis, J. Mylopoulos, and M. Stanley, “Telos: A Knowledge Representation Language for Requirements Modeling”. Technical Report KRR-TR-89-4, Dept. of Computer Science, University of Toronto, February 1989.
- [11] K.L.Chung, “An Extended Taxis Compiler”, M.Sc thesis, Dept. of Computer Science, University of Toronto, January 1984.
- [12] DAIDA, “Final Version on TDL Design”, ESPRIT Project 892, DAIDA, Deliverable DES1.2, 1987.
- [13] Guttag, J., J.J. Horning, and J.W. Wing, “Larch in five easy pieces”, TR 5, DEC Systems Research Center, 1985.
- [14] Hayes, I. (editor), “Specification Case Studies”, Prentice Hall International, Englewood Cliffs NJ, 1987.
- [15] Matthes, F., A. Rudloff, and J.W. Schmidt. “Data- and Rule-Based Database Programming in DBPL”. Esprit Project 892, DAIDA, Deliverable IMP 3.b, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, March 1989.
- [16] Meirilaen, E. and J.-M. Trignon, “An object based prototyping workbench for Prolog”, Proc. ESPRIT Conference '88: Putting the Technology to Use, Brussels, Belgium, pp. 423-437.
- [17] Mylopoulos, J., P. Bernstein, and H. Wong, “A language facility for designing data-intensive applications”, ACM TODS 5(2), June 1980.
- [18] Schmidt, J.W., “Some High Level Language Constructs for Data of Type Relation”. ACM Transactions on Database Systems, 2(3), September 1977.
- [19] Schmidt, J.W., H. Eckhardt, and F. Matthes, “DBPL Report”. DBPL-Memo 111-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1988.
- [20] Schmidt, J.W. and M. Mall, “Abstraction mechanisms for database programming”. Proc. ACM SIGPLAN, Symposium on Programming Languages Issues in Software Systems, ACM SIGPLAN Notices, 18:6, 1983, pp. 83-93.

- [21] Spivey, J.M., *“The Z Notation Reference Manual”*, Prog. Res. Group, Oxford University, 1987.
- [22] Stemple, D. and T. Sheard, *“Specification and verification of abstract database types”*, ACM PODS Conference, Waterloo, Ontario, April 1984, pp.248-257.
- [23] Wirth, N., *“Programming in Modula-2”*. Springer-Verlag, 1983.
- [24] Wordsworth, J.B., *“A Z Development Method”*, IBM UK Lab Ltd., Winchester, 1987.