Specification and Refinment in an Integrated Database Application Environment*

Ingrid Wetzel, Klaus-Dieter Schewe, Joachim W. Schmidt
University of Hamburg, FRG

Draft Version, February 12, 1993

Abstract

The DAIDA** project developed a layered design architecture for data-intensive applications, in which the transformation of a conceptual design to an efficient implementation is realized by stepwise refinement using Abrial's framework of Abstract Machines. One of the key contributions of DAIDA is the development of a mapping technology from an extended object-oriented, constraint-based specification language to an integrated, efficient database programming language based on sets and first-order predicates. This paper reviews the experiences made with this approach and derives requirements for formal specification languages tailored to data-intensive applications.

1 Introduction

The specification of data-intensive information systems is dominated by the attempt to model application domains that can be viewed in terms of objects and their relationships developing over time and restricted by numerous constraints. The complexity of the programs (transactions) may be small but the data upon which they act are very large, long lived, sharable by many programs, and subject to numerous rules of consistency. These requirements result directly from the fact that databases serve as (partial) representations of some organizational unit or physical structure that exist in their own constraining context and on their own time-scale independent of any computer system. Hence, in contrast to applications like numerics or graphics, the emphasis is on consistency preservation.

The DAIDA project developed a three-layer architecture that relates knowledge-based requirements modeling, conceptual software design and efficient database application programming. The highest layer handles requirements modeling followed by a layer for conceptual design an an implementation layer.

For each of these layers we used a specific language:

*This work has been supported in part by research grants from the E.E.C. Basic Research Action 3070 FIDE: "Formally Integrated Data Environments".

**DAIDA ("Development of Advanced Interactive Data-intensive Applications") is an ESPRIT project funded by the E.E.C under research contract 892.

- the knowledge representation language **TELOS** [BKMS89] for requirements modeling,
- the semantic data and transaction language **TDL** [BMS87] for conceptual design and
- the database programming language **DBPL** [SEM88] for implementation.

After the elaboration of a satisfiable requirements model, the task is to map those parts that shall be implemented down to a conceptual design. In DAIDA this is done via a goal-driven, dependency-based mapping assistent [CKM+89]. This conceptual design must be the object of further refinements that lead to an efficient database implementation. In DAIDA we used a second mapping assistant based on formal specifications and stepwise refinement in Abstract Machines [BMSW89]. It was the explicit specification of states space and transitions that led us to the decision to use Abstract Machines. The main focus of this paper is to review this mapping from conceptual design to implementation.

In section 2 we give a short summary of the languages used by DAIDA. In section 3 we describe in detail our formal specification approach based on Abrial's Abstract Machines. Section 4 illustrates standard refinement steps that were used in DAIDA and that are based on a computational model tailored to data-intensive applications. We conclude with an outline of the experiences we made in the project with formal method application.

2 The DAIDA Languages

We present an object oriented way of describing the design of a database application by using the language TDL [BMS87] and the main aspects of the implementation language DBPL [SEM88].

2.1 The Conceptual Design Language TDL

TDL is a language that describes data as classes of objects related by attributes. Database states correspond to the extent of these data classes. A variety of different data classes is supported: Basic Classes, Enumerated Classes, Aggregate Classes and Entity Classes. Basic Classes and Enumerated Classes are used to model simple value sets. In general however, the structure of values in TDL can be complex. Values denoting tuples of other values are modelled by *AggregateClasses*. Entities in the application domain are modelled by *EntityClasses*. Specific integrity constraints may be assigned to the attributes of Entity Classes such as attribute categories, range constraints, initial/final conditions and general invariants.

Attribute categories can be used to characterize the values of an object to be (un)changing during the object's lifetime or unique within the class. Range restrictions require the values of the attribute to belong to some other class. Initial/final conditions specify conditions on new objects or on objects to be deleted.

Atomic state transitions are modelled as instances of *TransactionClasses*, where the input/output of a transaction and its actions is described by appropriate attributes and

logical formulas. The body of a transaction is specified using the familiar precondition/postcondition and used also in the formal specification language 'Z' [Spi88] [Spi89] [ScPi87]. Preconditions are given via the keyword *GIVEN*, whereas postconditions require the keyword *GOALS*. In opposite to Z only those changes are expressed that are caused by the transaction, it is implicitly assured that everything else remains unchanged – the "frame assuption".

Inheritance is supported for data classes as well as for transaction classes, allowing the organization of objects and the reuse of transaction specifications.

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

```
TDLDESIGN ResearchCcompanies 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 onEmpProj: True IS
        ( THIS.worksOn  ⊆  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 ∈ Companies :  THIS ∈ x.engagedIn});
  END Projects;
  TRANSACTION CLASS HireEmployee WITH
    IN name : Strings; belongs : Companies; works :  SetOf Project;
    OUT, PRODUCES e: Employees;
    GOALS (e'.name = name)  AND (e'.worksOn = works)  AND
        (e'.belongsTo = belongs);
  END HireEmployee;
END ResearchCcompanies;
```

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.

2.2 The Implementation Language DBPL

The database programming language DBPL [SEM88] integrates a set- and predicate-oriented view of database modelling into the system programming language Modula-2. 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. 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.

The main extensions of DBPL with respect to Modula-2 are the following:

- DBPL provides a new data type relation which allows to introduce relational database modelling to be coupled with the expressiveness of the programming language.
- The new datatype is orthogonal to the existing types of Modula-2, hence sets of arrays, arrays of relations, records of relations, etc. can be modelled.
- Complex access expressions as usual in relational databases allow to abstract from iteration.
- All modules can be qualified to be database modules, which tunes the variables in them to be persistent and shared.
- Specific procedures are characterized to be transactions denoting atomic state change on persistent data. Therefore, DBPL provides mechanisms for controlled concurrent access to such data and for redovery.

Let us now illustrate the simple TDL-example above in DBPL notation:

```
DEFINITION MODULE ResearchCompaniesModule;
  IMPORT Identifier,String;
  TYPE
    Agencies = (ESPRIT, DFG, NSF, ..);
    CompNames, EmpNames,ProjNames = String.Type;
    EmpIds = Identifier.Type;
    ProjIdRecType = RECORD projName : ProjNames; getsGrantFrom : Agencies END;
    ProjIdRelType = RELATION OF ProjIdRecType;
    CompRelType = RELATION compName OF
                  RECORD compName : CompNames; engagedIn : ProjIdRelType END;
    EmpRelType = RELATION employee OF
                  RECORD employee : EmpIds; empName : EmpNames;
                         belongsTo : CompNames; worksOn : ProjIdRelType END;
    ProjRelType = RELATION projId OF
                  RECORD projId : ProjIdRecType; consortium : RELATION OF CompNames END;
END ResearchCompaniesModule.
  TRANSACTION hireEmployee(empName:EmpNames;belongs:CompNames; works:ProjIdRelType) : Empl
END ResearchCompaniesModule.
```

```
DATABASE IMPLEMENTATION MODULE ResearchCompaniesModule;
  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.compName = 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.
```

Fig. 1: DBPL implementation of the ResearchCompanies example

3 The Formal Basis of the Mapping Assistant

Within the DAIDA project the major effort concentrated on the production of high-quality database application systems. This comprises to guide users in mapping TDL-designs to DBPL-implementations:

- 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 requires formal methods. In DAIDA we have chosen Abrial's Abstract Machines [Ab89] as a formal basis for the mapping and the B-Tool for the verification of proof obligations.

3.1 The Abstract Machine Formalism

An Abstract Machine specification consists of two components, namely its state space specification, and its state transition specification.

3.1.1 The Static Component of an Abstract Machine

The specification of a state space is given by a list of variable names, called the **state variables** and by a list of well-founded formulas of a many-sorted first-order language $\mathcal{L}$ called the **invariant** and denoted by $\mathcal{I}$. Free variables occurring in $\mathcal{I}$ must be state variables. Each state variable belongs to a unique basic set which has to be declared in some context. Hence, in order to complete the state space specification we must give a list of **Contexts** that can be seen by the machine.

Such a context is defined by

- a list of **basic sets**,
- a list of **constant names**,
- and a list of closed formulas over the language $\mathcal{L}$ called **properties**.

A basic set may be either the set of natural numbers, an abstract set given only by its name, a set given by the enumeration of its elements or a constructed set, where cartesian product, powerset and partial function space are the only allowed constructors. We may then assume to have a fixed preinterpretation of these sorts by sets.

The Language of States The language $\mathcal{L}$ associated with an Abstract Machine can then easily be formalized.

The basic sorts of $\mathcal{L}$ are NAT and the other non-constructed basic sorts. The set of sorts is recursively defined using the basic sorts and the sort constructors pow, $\times$, $\mapsto$ denoting powerset construction, cartesian products and partial functions.

Since we use a fixed preinterpretation of the sorts as sets, one may regard the elements of these sets as constant symbols in $\mathcal{L}$ of the corresponding sort. Other function symbols are given by the usual functions $+$, $*$ on NAT, $\cup$, $\cap$, $\setminus$ on powersets or by the constant declarations in some context. The terms and formulas in $\mathcal{L}$ are defined in the usual way. The semantics of $\mathcal{L}$ is given by an interpretation $(\mathcal{A}, \sigma)$, where $\mathcal{A}$ is a structure extending the preinterpretation on sorts and σ is a variable binding.

We assume $\mathcal{A}$ to be fixed and write $\models_\sigma \mathcal{R}$, iff $\mathcal{R}$ is true under the interpretation $(\mathcal{A}, \sigma)$.

Then the state space of an Abstract Machine with state variables $x_1, ..., x_n$ is semantically denoted by the set

$$\Sigma \quad = \quad \{\sigma : \{x_1, .., x_n\} \rightarrow \mathcal{D} \mid \sigma(x_i) \in D_{s_i} \; for \;\; all \;\; i\},$$

where each s_i is the sort of the variable x_i and $\mathcal{D}$ denotes the union of the sets $\mathcal{D}_s$.

A wff $\mathcal{R}$ of $\mathcal{L}$ such that the free variables of $\mathcal{R}$ are state variables denotes a subset of Σ, namely

$$\Sigma_\mathcal{R} \quad = \quad \{\sigma \mid\mid\!\!= _\sigma \mathcal{R}\}.$$

Hence the invariant serves as a means to distinguish legal states in Σ_I from others.

Preinterpretation $\mathcal{A}$ of Sorts Since natural numbers and arbitrary denumerable sets are unique (up to isomorphism) there is no loss of generality to assume this preinterpretation $\mathcal{A}$ of sorts being fixed.

Now let $\mathcal{A}$ be a preinterpretation of the sorts. $\mathcal{A}$ assigns to each sort s a set $\mathcal{A}(s) = \mathcal{D}_s$ in the following way:

- $\mathcal{A}(NAT)$ is the usual set of natural numbers,
- for each basic sort s $\mathcal{A}(s)$ is some denumerable set such that the sets assigned to basic sorts are mutually disjoint, and
- constructed sorts are carried over canonically to the corresponding set construction.

The semantics of $\mathcal{L}$ is given by an **interpretation** $(\mathcal{A}, \sigma)$ consisting of a **structure** $\mathcal{A}$ and a **variable binding** σ. The structure $\mathcal{A}$ gives a denotation of the sorts of the language and of the other language parameters, i.e. the predicate-, function- and constant-symbols. A structure consists of the following:

(i) $\mathcal{A}$ assigns to each sort s the set D_s. Moreover, it assigns each constant symbol c^s to itself.

(ii) The sort-constructors pow, $\times$ and $\mapsto$ are assigned their usual set-theoretic meaning of powersets, cartesian products and partial functions.

(iii) The same holds for the functions $\cup$, $\cap$ and $\setminus$ on $pow(s)$. The function $+$, $*$, $\setminus$ are assigned their usual meaning on NAT.

(iv) Each function symbol f of sort $(s_0, \ldots, s_n)$ is assigned to a function

$$f^{\mathcal{A}} : D_{s_0} \times \ldots \times D_{s_{n-1}} \rightarrow D_{s_n} .$$

(v) $\mathcal{A}$ assigns to the predicate symbols $\in, \subseteq$ their usual set-theoretic meaning, and $=$ denotes equality.

In order to define the truth of a well-formed formula we have to introduce the concept of variable-assignment. A typed variable-assignment σ in a structure $\mathcal{A}$ is defined as follows: For each sort s a function σ_s on the variable set V_s is defined assigning for each $v^s \in V_s$ a domain element in D_s: $\sigma_s : V_s \rightarrow D_s$. Then σ is the collection of all σ_s. A structure $\mathcal{A}$ together with a variable-assignment σ is called an **interpretation** $(\mathcal{A},\sigma)$ of the language $\mathcal{L}$. Given an interpretation $(\mathcal{A}, \sigma)$, we may extend σ in the usual way to terms and well-formed formulas. We write $\models_{(\mathcal{A},\sigma)} \mathcal{R}$, iff the formula $\mathcal{R}$ is true in the interpretation $(\mathcal{A}, \sigma)$.

3.1.2 The Dynamic Component of an Abstract Machine

The specification of state transitions is given through substitutions over the language $\mathcal{L}$. We distinguish two kinds of transitions:

- The **initialization** assigns initial values to each of the state variables.

- **Transactions** update the state space.

Both kinds of state transitions are specified using the substitutions introduced by Dijkstra [Dij76] [DiSc89] with the additional possiblity of unbounded choice:

> $@z\bullet(P \rightarrow S)$ used as an abbreviation for an IF with an infinite list of guarded commands of the form $P(z) \rightarrow S(z)$, i.e. select any value for z such that P is true and execute the operation S' resulting from S by replacement of all free occurrences of z in S by the selected value. Clearly, if such a value for z does not exist, the @-substitution fails to terminate.

The semantics of substitutions S is given by means of two specific predicate transformers[1] $wlp(S)$ and $wp(S)$[2], which satisfy the *pairing condition*, i.e. for all predicates $\mathcal{R}$

$$wp(S)(\mathcal{R}) \;\equiv\; wlp(S)(\mathcal{R}) \wedge wp(S)(true)$$

and the *conjunctivity condition*, which states for any family $(R_i)_{i\in I}$ of predicates

$$wlp(S)(\forall i \in I.\, R_i) \;\equiv\; \forall i \in I.\, wlp(S)(R_i)\ .$$

These conditions imply the conjunctivity of $wp(S)$ over non-empty families of predicates. As usual $wlp(S)$ will be called the **weakest liberal precondition** of S, and $wp(S)$ will be called the **weakest precondition** of S. The notation f^*, which we shall use later, denotes the **conjugate predicate transformer** of f. It is defined by

$$f^*(\mathcal{R}) \;\equiv\; \neg\, f(\neg\mathcal{R})\ .$$

The definitions of $wlp(S)$ and $wp(S)$ for all the substitutions that we allow are given in [DiSc89] or can be easily derived from there. For more details see [?].

In addition to the properties stated above we have for any substitution S

$$wp(S)(false) \;\equiv\; false\ .$$

This is the hard discussed **Law of Excluded Miracles**. The related work by the PRG group [MRG88] and by J.-R. Abrial dispenses with this law. The main reason not to follow them is that we do not see any need for miracles. A more detailed discussion on the consequences implied by the introduction of miracles can be found in [?].

Semantics of Substitutions Now we may associate a set-theoretic meaning to the substitutions of an Abstract Machine in terms of the state space.

For this purpose we introduce the **characteristic predicate** P_σ of a state $\sigma \in \Sigma$ as

$$P_\sigma \quad\equiv\quad x_1 = \sigma(x_a) \wedge \cdots \wedge x_n = \sigma(x_n).$$

Characteristic predicates have only one model, i.e.

$$\Sigma_{P_\sigma} \quad=\quad \{\sigma\}.$$

[1] By abuse of notation, wffs in $\mathcal{L}$ are simply called predicates.

[2] Note that we use different interpretations than J.-R. Abrial, P. Gardiner and C. Morgan [Ab89] [MRG88] for the pure guarded command and for the @-substitution, since we do not want to dispense with Dijkstra's Law of excluded miracles [DiSc89].

Then we may associate with a substitution S the following set of state pairs

$$\Delta(S) \quad = \quad \{(\sigma_1, \sigma_2) \in \Sigma \times \Sigma \mid\mid\!\models_{\sigma_1} wlp(S) * (P_{\sigma_2})\}.$$

Since $wlp(S) * (P\sigma_2)$ characterizes the states σ_1 such that there exists a computation of S in σ_1 leading to σ_2, $\Delta(S)$ describes the set of possible state transitions on Σ under control of S.

In addition, we need to know the subset of Σ that allows some computation of S not to terminate, this is given by

$$\Sigma(S) \quad = \quad \{\sigma \in \Sigma \mid\mid\!\models_{\sigma} wp(S) * (false)\}.$$

Summarizing, the set-theoretic semantics for substitutions is captured by a pair of assignement functions (Δ, Σ_{trm}), with signature

$$\begin{array}{rcl} \Delta & : & S \rightarrow pow(\Sigma \times \Sigma) \ \ and \\ \Sigma_0 & : & S \rightarrow pow(\Sigma). \end{array}$$

Then the global semantics of an Abstract Machine can be given in terms of traces on Σ, i.e. a set of finite of infinite sequences $\sigma = \sigma_o, \sigma_1, \sigma_2, \cdots$ of states $\sigma_i \in \Sigma$ such that the following conditions are satisfied:

(i) for all $\sigma_{-1} \in \Sigma$ we have $(\sigma_{-1}, \sigma_0) \in \Delta(S_0)$, where S_0 is the initialization,

(ii) for each $i > 0$ there is a transaction in S with $(\sigma_{i-1}, \sigma_i) \in \Delta(S)$,

(iii) if $t = \sigma_0, \sigma_1, \cdots, \sigma_n$ is finite, then there is a transaction S with $\sigma_n \in \Sigma_0(S)$.

3.2 Verification

In the following we describe which properties must be verified to assure thransaction consistency and correct refinement. Then we indicate how to use a mechanical theorem proving assistant to guide the proofs.

3.2.1 Transaction Consistency Proof Obligation

The invariant component of an Abstract Machine states properties defining the set of legal states. Therefore, it is necessary to check that the transactions of the Abstract Machine always preserve the invariant.

In fact, if S is a transaction, it should be sufficient to require each terminating execution starting from a state satisfying the invariant $\mathcal{I}$ to result also in a state satisfying $\mathcal{I}$. As to the initialization S_0, we should require that any computation results in a state satisfying $\mathcal{I}$ no matter which was the initial state. Hence the following formal definition.

Definition 3.1 (Consistency) *Let M be an Abstract Machine with invariant $\mathcal{I}$.*

(i) A substitution S in M is **consistent** *$\mathcal{I}$ iff $\mathcal{I} \Rightarrow wlp(S)(\mathcal{I})$.*

(ii) M is **consistent** *iff all transactions S in M are consistent with respect to $\mathcal{I}$ and the initialization S_0 satisfies $wp(S_0)(\mathcal{I})$.*

3.2.2 Refinement

Refinement is used as a means to map specifications down to implementations. This includes refining the operations as well as transforming the state space into a more suitable form. We shall first address operational refinement.

The intention behind refinement of an operation S is to eliminate step by step all non-terminating computations under control of S and to cancel some of the non-deterministic computations. Moreover, the remaining computations of S should establish at least the same final states.

These considerations lead to the following formal definition:

Definition 3.2 (Operational Refinement) *Let S and T be two substitutions on the same state variables. Then T* **refines** *S iff the following three conditions hold for all predicates $\mathcal{R}$:*

(i) $wlp(S)^*(true) \Longrightarrow wlp(T)^*(true)$

(ii) $wlp(S)^*(true) \Longrightarrow (wlp(S)(\mathcal{R}) \Rightarrow wlp(T)(\mathcal{R}))$

(iii) $wp(S)(\mathcal{R}) \Longrightarrow wp(T)(\mathcal{R})$

We shall now give another form of these proof obligations that do not require a universal quantification over predicates.

Proposition 3.1 (Normal Form Proof Obligation) *Let S and T be substitutions with state variables $x_1, \ldots, x_n$. Let $z_1, \ldots, z_n$ be another bunch of variable names such that the x_i and the z_j are mutually different. Let x and z be the usual abbreviations for these bunches of variables. Then T refines S iff:*

(i) $(z = x) \wedge wlp(S)^*(true) \Longrightarrow wlp(T)^*(true) \wedge wlp(\{x/z\}.T)(wlp(S)^*(z = x))$ *and*

(ii) $(z = x) \wedge wp(S)(true) \Longrightarrow wp(T)(true)$

As to data refinement, the usual approach taken by [Ab89] [MRG88] is to use an abstract predicate $\mathcal{A}$ involving the state variables of both the substitutions of S and T. Assume that these state variables are mutually different. According to proposition 3.1 we define in this spirit:

Definition 3.3 *Let M, N be Abstract Machines with mutually different state variables x and z. Let $\mathcal{A}$ be some predicate on x and z. Then N* **refines** *M iff there is a bijection σ from the transactions of M to those of N such that for all pairs S, $T = \sigma(S)$ of transactions we have:*

(i) $\mathcal{A} \wedge wlp(S)^*(true) \implies wlp(T)^*(true) \wedge wlp(T)(wlp(S)^*(\mathcal{A}))$ *and*

(ii) $\mathcal{A} \wedge wp(S)(true) \implies wp(T)(true)$.

Syntactically, the abstraction predicate $\mathcal{A}$ is used in data refinement is added to the specification of the refined machine N using the additional keyword $CHANGE$.

3.2.3 The B Proof Assistent

The B-Tool is a general proof assistant and uses a goal-oriented proof technique based on Gentzen's calculi. The built-in logic performs simple and multiple substitution and is designed to suit best the calculus of substitutions.

For the practical work with the B-Tool it is essential that proofs can be organized such that any unreducable goals (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).

In DAIDA the B-tool has been used for two purposes:

- We used the built-in rewriting capability for a purely syntactical transformation of the TDL design into a first Abstract Machine. By this we exploited the fact that pre-post-specifications as they are used in TDL are equivalent to substitutions.
- We used B's capability as a theorem prover in order to prove the consistency of a machine and the refinement relation between two machines.

4 Refinement in DAIDA

The purpose of designs is the exact representation of application semantics without taking care of efficiency criteria. For this purpose the DAIDA project used the design language TDL.

The purpose of implementations is to represent applications such that they can be executed efficiently. DBPL is a database programming language satisfying this requirement. The goal of the mapping process is to close the conceptual gap between these two layers by the exploitation of standard refinement steps.

In order to use the Abstract Machines for this purpose we have to address three problems:

- represent designs in Abstract Machines, i.e. transform a TDL-description into an Abstract Machine,
- identify final specifications in Abstract Machines that can be transformed automatically into DBPL-programs, and
- identify standard refinement steps within Abstract Machines that are directed towards such final specifications.

4.1 Standard Refinement Steps

Let us now describe a bunch of different standard refinement steps that turned out to be sufficient for the mapping task from TDL to DBPL.

4.1.1 Data Identification

The *first refinement* step in our example addresses the basic issues of *data identification*. In our example 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 does not 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 new 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
$Companies = CompNames; Employees = EmpIds; Projects = ProjNames \times Agencies; \ldots$

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

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

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.

In general, data identification implies the introduction of identifiers, the replacement of id-based references with value-based references, and the introduction of referential integrity constraints as preconditions of the operations.

4.1.2 Operational Refinement

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 new *EmpId*-value, which is then associated with the required properties by means of function extension:

OPERATIONS hireEmployee (name, belongs, works) =
PRE $name \in EmpNames \wedge belongs \in companies \wedge works \in pow\ (engagedIn(belongs))$
THEN
$(empName(newEmpId), worksOn(newEmpId), belongsTo(newEmpId)) \mid\!\leftarrow (name, works, belongs) \;||$
$employees \mid\!\leftarrow employees \cup \{newEmpId\} \;||\; hireEmployees \mid\!\leftarrow newEmpId$
END hireEmployee;

OTHER $newEmpId \in (\rightarrow EmpIds - employees);$

Our refined data and procedure representation can be proven to meet all the invariants layed down within the initial Abstract Machine.

4.1.3 Data Reification

Having decided upon the identification, we are now ready for another step that designs the central *data structures* of our implementation. While our previous Abstract Machines have a purely functional view of the data. We now refine to 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 *EmpNames*, *companies*, and sets of *projects*.

INVARIANTS $empClass \in (employees \rightarrow EmpNames \times companies \times pow\ (projects)); \ldots$

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

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

Another change of the representation comes from the introduction of a variable, *tEmpId*, that allows us to replace the parallel substitutions in the operation *hireEmployee* by serial ones.

An alternative refinement could introduce "flat variables":

VARIABLES *flatCompClass, flatEmpClass, flatProjClass, empProjClass, ...;*

with constraints like

INVARIANTS $flatEmpClass \in (employees \rightarrow EmpNames \times companies); \ldots$
$empProjClass \in (employees \leftrightarrow projects); \ldots;$

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) = ...;*
$flatEmpClass \mid\leftarrow flatEmpClass \oplus \{(\ tEmpId, name, belongs)\} \mid\mid$
$empProjClass \mid\leftarrow empProjClass \oplus inverse(\lambda x.(x \in works \mid (tEmpId))) \ldots$

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

4.1.4 Data Typing

On our way down to a relational implementation there is another 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 a machine with the following context, variables and invariants:

CONTEXT
$ProjIdRecType = ProjNames \times Agencies;$
$ProjIdRelType = pow\ (ProjIdRecType);$
$CompRelType = (CompNames \mathrel{+\!\!\rightarrow} ProjIdRelType);$
$EmpRelType = (EmpIds \mathrel{+\!\!\rightarrow} EmpNames \times CompNames \times ProjIdRelType);$
$ProjRelType = (ProjIdRecType \mathrel{+\!\!\rightarrow} pow\ (CompNames)); \ldots$
VARIABLES
compRel, empRel, projRel
INVARIANTS
$compRel \in CompRelType;\ compRel = compClass\ ;$
$empRel \in EmpRelType;\ empRel = empClass\ ;$
$projRel \in ProjRelType;\ projRel = projClass\ ; \ldots$

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

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

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

Due to the semantics of the generalized substitution, the condition 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 a specific value, *nilEmpId*, as an exception.

Invariants and definitions of the Abstract Machine variable *empRel* translate one-to-one into the type definitions of the corresponding DBPL variable *empRel*.

4.1.5 Adding Run-Time Semantics

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 the following first-order database query expression:

```
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
```

We can see how the DBPL query expressions based on first-order predicates nicely corresponds with substitutions.

5 Conclusion

Within the layered DAIDA architecture formal specification and verification come into play as a means for mapping step-by-step from conceptual designs down to implementations. We use the formalism of Abstract Machines, since they allow to capture at the same time the semantics of TDL-designs and to introduce via refinement the procedurality required by efficiency-oriented database programs.

Database applications require the conceptual specification of highly structured and interrelated data. It is the complexity of data that makes database application development a hard task, whereas most operations on the data are rather easy such as simple insert, delete and update operations. Integrity constraints on the data are normally

expressed by first-order predicates. Thus, the possibility to explicitly represent such constraints via invariants makes Abstract Machines a good choice for database application specification – much more natural than e.g. algebraic specifications.

Moreover, most commercial database systems do not offer any sophisticated mechanism in order to prove consistency. Integrity checking is done at run-time before transaction commit. In Abstract Machines, however, we were able to prove the consistency already at specification time. Transactions are modeled via operations in Abstract Machines, and consistency is semi-automatically verified using the B proof assistent. In addition, it can be shown that most lemmata resulting during a B-tool proof can be classified into only very few classes such that in most proof situations generic proofs may be reused.

Another positive experience we made with Abstract Machines is the possibility to refine a specification within one and the same language. This poses the strong requirement on formal specification languages to capture at the same time the semantics of a high-level conceptual design language and of an efficient implementation language. However, one big problem arises. Refinement in Abstract Machines as it is usually defined does only preserve the semantics of the operations, but not of the data, for which it is much too liberal. This is not better in other specification styles, since database application have not yet been the focal point of the formal specification community. Most of the work that has been done follows the line of abstract data types ignoring the fact that in database application systems the data live longer than the operations on them.

Our current work is built upon the DAIDA experience preserving the style of transaction specifications.

Acknowledgment

We should like to thank J. R. Abrial and our colleagues in DAIDA, especially Alex Borgida, Michael Mertikas and the researchers at BP research centre, for many fruitful discussions and comments. We also want to thank Paul Gardiner from the Programming Research Group at Oxford University for interesting discussions and his contribution to the set-theoretic semantics of substitutions.

References

[Ab89] J.R. Abrial: *A Formal Approach to Large Software Construction*, in J.L.A. van de Snepscheut (ed.): *Mathematics of Program Construction*, International Conference Groningen, The Netherlands, June 89, Proceedings, LNCS 375, Springer-Verlag, 1989.

[BMS87] A. Borgida, J. Mylopoulos, J. W. Schmidt: *Final Version on TDL Design*, DAIDA deliverable

[BKMS89] A. Borgida, M. Koubarakis, J. Mylopoulos, M. Stanley: *TELOS: A Knowledge Representation Language for Requirements Modeling*, Technical Report KRR-TR-89-4, Dept. of Comp. Sci., University of Toronto, 1989

[BMSW89] A. Borgida, J. Mylopoulos, J. W. Schmidt, I. Wetzel: Support for Data-Intensive Applications: Conceptual Design and Software Development, in *Proceedings of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, june 1989

[CKM+89] L. Chung, P. Katalagarianos, M. Marakakis, M. Mertikas, J. Mylopoulos, Y. Vassiliou: *From Requirements to Design: A Mapping Framework for Information Systems*, to appear in Information Systems

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DiSc89] E.W. Dijkstra, C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1989.

[EhMa85] H. Ehrig und B. Mahr: *Fundamentals of Algebraic Specification*, vol.1, Springer 1985

[Hay87] I. Hayes. *Specification Case Studies.* Prentice Hall, 1987.

[Heh84] E.C.R. Hehner. *The Logic of Programming.* Prentice Hall, 1984.

[HGM86] E.C.R. Hehner, L.E. Gupta und A.H. Malton, *Predicative Methodology.* Lecture Outline, Marktoberdorf Summer School, August 1986.

[Jon86] C.B. Jones: *Systematic Software Development using VDM*, Prentice-Hall International, London 1986.

[MRG88] C. Morgan, K. Robinson, P. Gardiner. *On the Refinement Calculus*, Technical Monograph PRG-70, Oxford University Computing Labaratory, Oktober 1988.

[SEM88] J. W. Schmidt, H. Eckhardt, F. Matthes: *DBPL Report*, DBPL-Memo 111-88, University of Frankfurt, 1988

[ScPi87] S.A. Schuman und D.H. Pitt: *Object–Oriented Subsystem Specification*, in L. Meertens, (ed.): *Program Specifiation and Transformation, The IFIP TC2/WG2.1 Working Conference, Bad Tölz, FRG, April 15-17, 1986*, North Holland Publishing Co, Amsterdam, 1987.

[Spi88] J.M. Spivey. *Understanding Z, A Specification language and its Formal Semantics.* Cambridge University Press, 1988.

[Spi89] J.M. Spivey. *The Z Notation, A Reference Manual*, Prentice Hall, 1989.

[Wir85] M. Wirsing: *Structured Algebraic Specifications – A Kernel Language*, Passau University Reports, MIP 8511, 1985

A From TDL Designs through Abstract Machines to DBPL Implementations

```
TDLDESIGN ResearchCcompanies IS
  BASIC CLASS CompNames, EmpNames, ProjNames = Strings;
  ENUMERATED CLASS Agencies = {'ESPRIT, 'DFG, 'NSF, ...};
  ENTITY CLASS Companies WITH
    UNIQUE, UNCHANGING compName : CompNames;
    CHANGING engagedIn : SetOf Projects;
  END Companies;
  ENTITY CLASS Employees WITH
    UNCHANGING empName : EmpNames;
    CHANGING belongsTo : Companies; worksOn : SetOf Projects;
    INVARIANTS onEmpProj: True IS (this.worksOn subsetOf this.belongsTo.engagedIn);
  END Employees;
  ENTITY CLASS Projects WITH
    UNIQUE, UNCHANGING projName : ProjNames; 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 : EmpNames; belongs : Companies; works : SetOf Projects;
    OUT, PRODUCES e: Employee;
    GOALS (e'.empName' = name) and (e'.worksOn' = works) and (e'.belongsTo' = belongs);
  END HireEmployee;
END ResearchCcompanies;
```

Fig. 1: TDL Design of ResearchCcompanies Example

MACHINE researchCcompanies.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 $companies \in \wp(Companies)$; $compName \in (companies \rightarrow CompNames)$;
$engagedIn \in (companies \rightarrow \wp(projects))$;

$employees \in \wp\ (Employees)$; $empName \in (employees \rightarrow EmpNames)$;
$belongsTo \in (employees \rightarrow companies)$; $worksOn \in (employees \rightarrow \wp\ (projects))$;

$projects \in \wp(Projects)$; $projName \in (projects \rightarrow ProjNames)$;
$getsGrantFrom \in (projects \rightarrow Agencies)$; $consortium \in (projects \rightarrow \wp(companies))$;

$\forall x,y.\ x,y \in companies \Rightarrow (compName(x) = compName(y) \Rightarrow x = y)$;
$\forall x.\ x \in employees \Rightarrow (worksOn(x) \subseteq engagedIn(belongsTo(x))$;
$\forall x,y.\ x,y \in projects \Rightarrow$
$(projName(x) = projName(y) \wedge getsGrantFrom(x) = getsGrantFrom(y) \Rightarrow x = y)$;
$\forall x.\ x \in projects \Rightarrow (consortium(x) = \{y \mid y \in companies \wedge x \in engagedIn(y)\})$;

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

End researchCompanies.1;

Fig. 2: Initial Abstract Machine

MACHINE researchCompanies.2

IMPLY researchCompanies.1

BASIC SETS EmpIds

CONTEXT $Companies = CompNames$; $Employees = EmpIds$; $Projects = ProjNames \times Agencies$

VARIABLES empId, projId, tEmpId, ...

INVARIANTS $empId \in (employees \rightarrow EmpIds)$; $tEmpId \in EmpIds$; ...

DEFINITIONS $compName = \lambda x.(x \in companies \mid x)$;
$empId = \lambda x.(x \in employees \mid x)$;
$(projName, getsGrantFrom) = projId = \lambda x.(x \in projects \mid x)$; ...

OPERATIONS hireEmployee (name, belongs, works) =
PRE $name \in EmpNames \wedge belongs \in companies \wedge works \in \wp\ (engagedIn(belongs))$
THEN

tEmpId $\rightarrowtail$ newEmpId;
(empName(tEmpId),worksOn(tEmpId),belongsTo(tEmpId)) $\rightarrowtail$ (name,works,belongs) ||
employees $\rightarrowtail$ employees $\cup$ {tEmpId} || hireEmployees $\rightarrowtail$ tEmpId
END hireEmployee;

OTHER newEmpId $\in$ *(*$\rightarrow$ *EmpIds - employees);*

END researchCompany.2;

Fig. 3: Abstract Machine with Identification Refinement

MACHINE researchCompanies.3

IMPLY researchCompanies.2

VARIABLES compClass, empClass, projClass, tEmpId

INVARIANTS $compClass \in (companies \rightarrow \wp\ (projects))$;
$empClass \in (employees \rightarrow EmpNames \times companies \times \wp\ (projects))$;
$projClass \in (projects \rightarrow \wp\ (companies))$;
$tEmpId \in EmpIds$; ...

DEFINITIONS $engagedIn = \lambda x.\ (x \in companies \mid compClass(x))$;
$(empName, belongsTo, worksOn) = \lambda x.\ (x \in employees \mid empClass(x))$;
$consortium = \lambda x.\ (x \in projects \mid projClass(x))$;

OPERATIONS hireEmployee (name, belongs, works) =
PRE $name \in EmpNames \wedge belongs \in companies \wedge works \in \wp\ (engagedIn(belongs))$
THEN $tEmpId \vDash$ newEmpId;
$empClass \vDash$ empClass $\cup$ {(tEmpId, name, belongs, works)};
$hireEmployee \vDash$ tEmpId
END
END hireEmployee;

OTHER $newEmpId \in (\ \rightarrow (EmpIds - employees))$;

END researchCompany.3;

Fig. 4: Abstract Machine with Data Structure Refinement

MACHINE researchCompanies.4

IMPLY researchCompanies.3

CONTEXT $ProjIdRecType = ProjNames \times Agencies$; $ProjIdRelType = \wp\ (ProjIdRecType)$;
$CompRelType = (CompNames \nrightarrow ProjIdRelType)$;
$EmpRelType = (EmpIds \nrightarrow EmpNames \times CompNames \times ProjIdRelType)$;
$ProjRelType = (ProjIdRecType \nrightarrow \wp\ (CompNames))$;...

VARIABLES compRel, empRel, projRel

INVARIANTS $compRel \in CompRelType$; $compRel = compClass$;
$empRel \in EmpRelType$; $empRel = empClass$;
$projRel \in ProjRelType$; $projRel = projClass$; ...

OPERATIONS hireEmployee (name, belongs, works) =
PRE $name \in EmpNames \wedge belongs \in CompNames \wedge works \in ProjIdRelType$
THEN IF $belongs \in companies \wedge works \in \wp\ (engagedIn(belongs))$
THEN $tEmpId \mid\!\leftarrow newEmpId$;
$empRel \mid\!\leftarrow empRel \cup \{(tEmpId, name, belongs, works)\}$;
$hireEmployee \mid\!\leftarrow tEmpId$
ELSE $hireEmployee \mid\!\leftarrow nilEmpId$
END
END hireEmployee;

OTHER $newEmpId \in (\ \rightarrow (EmpIds - employees))$;

END researchCompany.4;

Fig. 5: Final Abstract Machine with Type Refinement

```
DEFINITION MODULE ResearchCompaniesTypes;
  IMPORT Identifier,String;
  TYPE
    Agencies = (ESPRIT, DFG, NSF, ..);
    CompNames, EmpNames,ProjNames = String.Type;
    EmpIds = Identifier.Type;
    ProjIdRecType = RECORD projName : ProjNames; getsGrantFrom : Agencies END;
    ProjIdRelType = RELATION OF ProjIdRecType;
    CompRelType = RELATION compName OF
                  RECORD compName : CompNames; engagedIn : ProjIdRelType END;
    EmpRelType = RELATION employee OF
                  RECORD employee : EmpIds; empName : EmpNames;
                         belongsTo : CompNames; worksOn : ProjIdRelType END;
    ProjRelType = RELATION projId OF
                  RECORD projId : ProjIdRecType; consortium : RELATION OF CompNames END;
END ResearchCompaniesTypes.

DATABASE DEFINITION MODULE ResearchCompaniesOps;
  FROM ResearchCompaniesTypes IMPORT EmpNames, CompNames, ProjIdRelType, EmpIds;
  TRANSACTION hireEmployee(empName:EmpNames;belongs:CompNames; works:ProjIdRelType) : Empl
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.compName = 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.
```

Fig. 6: DBPL implementation of the ResearchCompanies example