THE SOFTWARE DEVELOPMENT ENVIRONMENT AS A KNOWLEDGE BASE MANAGEMENT SYSTEM

Alex Borgida, Matthias Jarke, John Mylopoulos
Joachim W. Schmidt, and Yannis Vassiliou

RCC/CSI/TR/1987/003

Computer Science Institute
Research Center of Crete
THE SOFTWARE DEVELOPMENT ENVIRONMENT AS A KNOWLEDGE BASE MANAGEMENT SYSTEM

Alex Borgida, Matthias Jarke, John Mylopoulos  
Joachim W. Schmidt, and Yannis Vassiliou

September 1987

RCC/CSI/TR/1987/003

Technical Report Series
Computer Science Institute, Research Center of Crete
P.O.Box 1385, Heraklio, Crete, 711-10 Greece
Tel.: +30.81.229302, Telex: 262389 CCI GR, Fax: +30.81.229342
E-mail – UUCP: mcvax!ariadne!yannis

THE SOFTWARE DEVELOPMENT ENVIRONMENT AS A KNOWLEDGE BASE MANAGEMENT SYSTEM

Alex Borgida
Matthias Jarke
John Mylopoulos
Joachim W. Schmidt
Yannis Vassiliou

Abstract. This paper sketches a software development environment for data-intensive information systems. It is based on the premise that software can be viewed as a multilayered description, which includes a requirements specification, a design and an implementation, and that this description should be managed as a knowledge base. The proposed environment uses the requirements modelling language CML, the design language Taxis, and the implementation language DBPL. The paper outlines the features of these languages and illustrates the proposed multilayered structure of software with an example. It also discusses the features that will have to be supported by an environment suitable for constructing, maintaining, and testing software knowledge bases.

1. INTRODUCTION

The development of useful software demands the integration of a great deal of multifaceted knowledge that is extracted from many different sources and reshaped into an operational computer program. For example, in order to develop a a student registration system, one needs to "know" that students have an associated department, and courses have instructors; also that students enrol in courses if certain rules are satisfied. The knowledge relevant to a particular software development task may be expressed in terms of a specialized language. Indeed, a large class of tasks involving scientific computing have been handled relatively successfully in terms of conventional programming languages starting with FORTRAN and APL. But for many other tasks, conventional and not-so-conventional languages offer an inadequate basis for explicitly

---

1 This work is supported in part by the European Community under Eirit contract 592 (DADA).
2 Rutgers University, USA
3 University of Passau, FRG
4 University of Toronto, Canada
5 University of Frankfurt, FRG
6 University of Crete, Greece
capturing and representing the relevant knowledge; this exacerbates the so-called software crisis.

Over the past fifteen years AI has made some progress towards the development of concepts, languages and techniques for the representation of knowledge. The research outlined in this paper is based on the premise that software development tools, methodologies and environments can benefit from this research. The paper presents a rationale for this thesis and sketches the main ingredients of a software development environment based on it.

In studying the application of knowledge representation techniques to software development we focus here on the construction of a software development environment for Information Systems. According to accepted practice in Software Engineering, such an environment needs to include facilities for requirements modelling, design, and implementation.

The application of knowledge representation techniques to requirements modelling has already been discussed in the literature [BGM85]. A requirements model must include an account of the environment within which the intended software will function (sometimes referred to as domain knowledge) along with the functional characteristics of the software to be built. The requirements modelling language RML [GREE84, BGM86], and its successor CML [STAN86] exemplify the kinds of languages one can obtain by applying knowledge representation techniques to this task. Both languages treat a requirements specification as a history of the world being modelled. CML, however, goes further in treating a requirements specification as a history of our knowledge of the world. This feature makes CML powerful enough to talk about the completeness, accuracy and precision of a model with respect to the world being modelled.

One might argue that a system design should simply provide an account of the system's behaviour, and, unlike a requirements specification, it should be independent of the environment within which the proposed system will eventually function. Consider, however, an example information system which maintains information about students at a University. A formal specification of such a system which merely describes its behaviour seems incomplete in an important way: it tells us how symbols will eventually be pushed around inside a machine, once the intended system is in operation; it does not give us any guidelines, however, on how to interpret those symbols because it does not provide an account of the meaning of the information handled by the system, i.e., how it relates to reality. This observation suggests that, at least for information systems, a design specification should include an account of how the behaviour mechanism corresponds to the world being modelled. For functional specifications of this sort we need a language which on one hand allows the description of system components, their states and I/O behaviour, on the other hand comes with a semantic theory that allows one to relate system state and function to the world being modelled. So called semantic data models attempt to do just that [BMS84].

An example of a semantic data model heavily influenced by knowledge representation ideas is Taxis [MBW80, MW80, NIXO84, MYLO86]. It allows one to design an information system by defining and structuring a large collection of classes. Some classes represent static information, e.g., the students known to the system at a particular moment; others dynamic information, such as the enrolment procedure or the
(long-term) event of studying at a university. One of the aims of the Taxis project is to develop a compiler that translates Taxis programs to Pascal/R [SCHM77], a dialect of Pascal offering relational database management facilities. Results of this research on the implementation of a semantic data model are reported in [NIXO87].

The implementation of an information system demands database management facilities as well as applications programming. For this purpose we have chosen DBPL, a database programming language [MRS84, ECKH85] for describing and implementing database structures and transactions so that they run efficiently on a computer system. The language and its runtime system focus on the efficient storage of data, and on the correct and optimized execution of transactions, concurrently submitted by multiple users. At the same time, DBPL provides full programming flexibility for nonstandard features of a particular application, and supports software engineering through features such as modularity, type-checking, data and control abstractions.

Viewing software as a multilayered description, where the higher layers constitute knowledge bases, influences greatly the structure and features of software development environments. To start with, each language used in the software development process is provided an environment. Those for the more procedural levels can offer typical facilities such as special-purpose editors, interpreters, tracing and debugging packages, version control, and the like [BABI86, KAC86, O'BR82, WATE85]. The environments for the less procedural levels also need reasoning and animation facilities so that a user can probe the knowledge base to see if it is consistent with his expectations.

A second desirable facility involves the maintenance and management of multilayered specifications; it may provide bookkeeping operations that allow software developers to keep track of inter-dependencies among the components of the different layers. With such a setup, it would be possible to determine how changes of the specification at one level affect the specifications at the other levels.

Third, a multi-lingual software development environment can provide help in generating lower level (more procedural) specifications from higher level (more declarative) ones. Depending on the nature of the two levels, it may be possible to have a compiler that handles this job. Alternatively, the environment may assist the interactive generation of the lower level specification from the higher level one(s). Through the use of expert system facilities, the environment can play the role of an active assistant rather than a passive bookkeeper.

The rest of this paper focuses on a particular software development environment intended to provide all three facilities mentioned above, namely local environments, mapping assistants, and global management tools [JARK86]. It is fair to add that there are scores of research issues to be addressed in realizing an environment of this type and that a research program addressing such issues can only be described as long-term. [BARS87] presents a fine survey of the general topic of knowledge-based programming environments.

The architecture of the environment we are proposing is grounded on the following observations concerning the nature of the software engineering process:

- Software engineering environments have been most successful when limited to a particular domain of tasks (e.g., telephone switching), where domain-specific
knowledge can be exploited. It is for this reason that it was decided to focus on the
important sub-domain of data-intensive information system applications, e.g.,
reservation systems, point-of-sale inventory control, and credit card verification.
Such systems are typically centered around one or more databases, along with
some packages of application programs; less aid is needed here for complex and
purely algorithmic tasks.

* Software development involves several steps: an abstraction process, in which
the real world is modeled, a design process, during which an abstract system
model is developed consistently with the world model, and an implementation
effort that gradually transforms a design into working programs. These processes
are related by mappings between the different models and call for different
language features. One approach is to choose so-called "wide spectrum languages"
[BAUE76], which provide all necessary features within a single framework. In
contrast, we have chosen a different and appropriate language for each level:7
CML for requirements modelling, a dialect of Taxis for design and DBPL for
implementation.

* The conceptual distance between the three linguistic levels supported by the
environment is too great to permit the automatic generation of software. More-
over, each level of the architecture (cf. Figure 1) requires new knowledge to be
added, and decisions to be made on the basis of information outside the environ-
ment. We therefore propose a workbench environment rather than an automated
software engineer.

* The prototyping approach [NJ82] has shown the usefulness of providing a close
connection between specifications at any level of abstraction and (a mock-up of)
the concrete system resulting from such specifications. In consequence, it
becomes easier to relate and find correspondences between the specification at
one level to those of other levels, and to validate models with respect to the real
world.

The subsystems interact dynamically as shown in Figure 1. Each level of analysis
or design is seen as a knowledge base which constrains and guides lower-level deci-
sions. The implementation of this interaction requires maintaining interdependencies
between any two neighbouring levels in the hierarchy as well as between each level
and a global knowledge base of global requirements and bookkeeping information. In
addition to these external constraints, each level of description is supported by its own
level-specific knowledge base which serves as an assistant to the developer for that
level.

In the remainder of this paper, we first sketch the three major levels supported by
our proposed environment and then return to the features of the environment and the
global knowledge base.

7 This decision provides, we feel, strong guidance on what aspects are relevant during any one phase of the software
development process.
2. REPRESENTING REQUIREMENTS IN CML

The specific focus of CML [BGMV85] is requirements modelling and analysis (also called "enterprise analysis" for information systems). This forms the initial stage of software development, where the designers of an information system describe the environment in which the system will eventually operate. As indicated earlier, CML adopts the premise that a requirements specification is itself a knowledge base capturing knowledge about the enterprise, the conceptual units that compose it, its rules of operation, and the way the enterprise handles information about the real world.

For example, suppose we want to computerize the information handling aspects of project accounting. As a first step, the software developers attempt to understand what projects are, how people are involved, what money is spent and on what, etc. The result of this familiarization with the domain is a CML specification where concepts are formalized and disambiguated, and rules and procedures governing the behaviour of the environment are laid down. It is important to emphasize that the concepts, rules and procedures described in the requirements specification need not be represented in the information system to be built.

For modelling such a situation, CML adopts an object-oriented representational framework based on semantic networks that emphasizes the organization of the requirements model. More specifically, a model consists of interrelated objects and assertions which impose constraints on the possible relationships, and relate its possible states to those of the domain. Individual objects in the knowledge base are organized into classes which describe commonalities of their instances in the form of constraints -- e.g., attributes applicable to them, the valid ranges of values of such attributes, additional integrity constraints, etc. In CML, classes themselves are objects, and
hence can be members of meta-classes. Therefore, a class can have its own attributes, which might represent, for example, aggregate information about its instances. Furthermore, classes are organized into a hierarchy with general classes located above their specializations. If one class (e.g., Employee) is defined to be a specialization, or subclass of another (e.g., Person), then at all times every instance of the former is considered an instance of the latter. An important consequence of this organization is that attributes can be inherited from superclass to subclass, e.g., the subclass Employee inherits attributes such as name, address, and so on, from the superclass of Person.

Apart from this representational framework, shared to a large extent with Taxis and many other semantic data models, CML views a requirements model as an account of a history of events and activities, thus emphasizing the use of time in the description of a corporation or office within which the intended information system will function. The model of time adopted is based on time intervals [ALLE81]. Every CML expression includes a temporal component which serves as a filter on its possible values. In addition, assertions, expressible in a formal logic, can be part of a CML specification in order to impose constraints on requirements. Assertions are themselves viewed as another kind of object and receive to a large extent the same treatment as other objects.

A crucial issue in building and using an information system is the interpretation of the information handled by the system, i.e., the means by which it is determined what the data says about its subject matter. For instance, if a project management system is to answer in a useful way the question:

"How much is left in the travel budget of project X?"

it is essential to know, in addition to a number retrieved from a database, the accuracy and completeness of the data stored, e.g., how the expenses recorded in the database relate to the actual expenses, and when the database was last updated. To express the above, CML offers facilities for talking about the referents of objects in a CML specification. This allows, among other things, for the representation of different forms of the closed world assumption [REIT78], including forms that take time into account.

The object-oriented framework adopted by CML also has several novel aspects. In particular, attributes (links) are treated as full-fledged objects, introducing attribute classes, attribute metaclasses, etc. The resulting mechanism of "attribute categories" allows customizing the language for particular applications, much as higher-order predicates would. CML also provides a way of dealing with over-abstraction in parts of the specification, through a mechanism for reconciling conflicting constraints such as exceptional subclasses.

As an illustration of CML facilities we present below portions of a requirements specification for the project management example. For this example, it is assumed that research projects begin after the acceptance of a proposal, hold meetings, produce progress reports approximately every 6 months, and a final report. Part of a progress report is an accounting of the moneys spent (projects have budgets which must be spent between their start and end date.)
IndividualClass Project IN ActivityClass WITH
  single input
    budg : Budget
  single output
    finalReport : ResearchReport
  single control
    proposal : ProjectProposal
    nm : Name
    num : ProjectNumber
    startDate, endDate : Date
  part
    meets: HaveMeeting
    genProgressReport : GenerateReport
  activationCondition
    NOW DURING startDate
  terminationCondition
    NOW DURING endDate
  initialCondition
    Accepted(proposal)
  postCondition
    budg.amountLeft = 0
  necessary
    {* one progress report is produced every 6 months 
       during the project *}
    (FOR EACH x/Time) 
    [ x DURING THIS => 
      (EXISTS 1..1 y/genProgressReport) 
      [(y STARTS AFTER x) and (y ENDS BEFORE x+6month)] ]

This class defines projects as activities which take a budget as input and produce a final report as output. Projects also have an associated name, number, project proposal, and start and end dates. Input and control\(^8\) attributes have to be available at the beginning of an activity and inputs are consumed by the activity. Outputs, on the other hand, are produced by the activity. The parts of an activity define component activities, i.e., a project includes zero or more "HaveMeeting" and "GenerateReport" activities. The rest of the defined attributes describe conditions that must be satisfied by every instance of the class "Project". Thus, projects are activated by the arrival of startDate and terminated on endDate.

The special variable NOW has as value an atomic interval which marks current time. Thus,

\(^8\) Inputs, outputs and controls of activities are similar in many respects to analogous features of SADT (trademark of Softech) [RS77].
NOW DURING startDate

becomes true when NOW falls completely within the designated starting date for the project. A precondition for a project to begin is that its proposal must have been accepted. Finally, a necessary condition for every project is that there is a progress report every six months. The expression of this condition in CML sheds some light on the model of time adopted. The special variable THIS in the assertion refers to an instance of the class being defined. Thus the antecedent of the implication constrains $x$ to fall within the time period of a project (an instance of the class "Project") while the consequent asserts that there is a (unique) progress report within a six month interval from $x$. The notion of activity used here is very similar to that used in RML [GREE84]. However, in CML one can actually define the meaning of meta-attributes such as "input", "postcondition" and "single", whereas these are built-in (and fixed) in RML.

Continuing with the example, the next class defines persons as entities that have several (single-valued) attributes and work on zero or more projects.

IndividualClass Person WITH
necessary single
    nm : Name
    org : Organization
    adr : Address
part
    worksOn : Project

A meeting for a project is called with a notification letter and generates minutes. Participants, who must be associated with that project, may travel to the meeting:
IndividualClass HaveMeeting IN ActivityClass WITH
    single input
    notification : Letter
    single control
    location : City
    proj : Project
    single output
    min : MeetingMinutes
    control
    participants : Person
    part
    travelTo, travelFrom : LongDistanceTravel
    [agent in participants over this]
    localActs : LocalActivity [agent in participants over this]
    necessary
    (* only one travel per participant to the site of the meeting *)
    (FORALL x/participants)
    (EXISTS 0..1 y/travelTo, z/travelFrom)
    y.agent = x AND y.dest = location AND
    z.agent = x AND z.orig = location
    (* all participants work on this project *)
    x IN participants => proj IN x.worksOn

Traveling is done by an agent, with a ticket from an origin to a destination:

IndividualClass PersonActivity IN ActivityClass WITH
    single control
    agent : Person

IndividualClass Travel IN ActivityClass ISA PersonActivity WITH
    single control
    orig, dest : City
    ticket : Ticket

Local activities take place in one city and on one day only:

IndividualClass LocalActivity ISA PersonActivity WITH
    single control
    where : City
    necessary
    (EXISTS 1..1 d/Day) (THIS DURING d)

with subclasses Lodging, Eating, LocalTransport, etc.
3. CONCEPTUAL SYSTEM SPECIFICATION IN TAXIS

The Taxis language is intended for the design specification of information systems [MBW80, WONG83]. The language provides a means for describing the information content of the proposed information system through an object-oriented framework similar to that adopted by CML. A major advantage of this framework over traditional record-based approaches is the direct and natural correspondence between the model and the world (e.g., no reliance on keys), which facilitates the design, access and interpretation of the information handled by the system. Data classes are intended to model the entities that are relevant to the subject matter and at the same time will eventually be stored in the database(s) of the information system. In addition, Taxis can be used to specify the dynamic aspects of an information system through transactions and scripts, which define respectively atomic and long duration operations on the database(s).

Taxis, as described in [MBW80], has been modified for the purposes of this project to allow for a declarative specification of these operations, using the familiar notions of pre/post conditions. The language also provides a procedural exception handling mechanism in order to deal with unexpected situations that might arise in the operation of the system.

The following design description presents some of the information which must be managed by the proposed system supporting financial project management.

First we define some classes to keep track of problems caused by different currencies and exchange rates.

```
ENUMERATED CLASS Currencies == ["FF","DM,..."];
ENUMERATED CLASS Months == ["Jan,..."];

INTEGER CLASS Money ISA Integer
WHERE (0 < SELF) AND (SELF < 10,000,000)

INTEGER CLASS ECU ISA Integer
WHERE (0 < SELF) AND (SELF < 10,000,000)
(* like money, but with fixed currency *)
```

```
ENTITY CLASS ExchRates WITH
UNCHANGING
   from: Currencies
   to: Currencies
CHANGING
   factor: REAL
UNIQUE
   (from,to)
END ExchRates;
```

Exchange rates are described by pairs of currency identifiers, in fields `from` and `to`, and the conversion factor. Attributes are grouped under several predefined categories in order to provide additional semantics in the form of constraints on updates. Thus UNCHANGING attributes cannot be modified, and no duplicate tuples
of UNIQUE attributes are allowed for objects in a class.

We next define the class of employee entities by first abstracting the generalized concept Person, which would presumably be useful elsewhere in the design.

ENTITY CLASS Person WITH
   name: String;
   address: Address;
...
END Person;

ENTITY CLASS Employee ISA Person WITH
   worksOn: SET OF Project;
END Employee;

Every employee therefore has attributes such as name, inherited from the Person superclass, and has the additional attribute worksOn, whose value is the set (actually, the bag) of all projects on which they are supposedly working.

Consider next information about expenses, which are incurred in various currencies.

ENTITY CLASS Expense WITH
   emp : Employee
   when : Date
   amount : Money
   currency : Currencies
   ECUamount : ECUs
   == SELF.amount * (ExchRates:
      (from=SELF.currency) AND (to="ECU")).factor
END Expense;

ENTITY METACLASS ExpenseKinds;

ENTITY CLASS FoodExpense IN ExpenseKinds ISA Expense
ENTITY CLASS HotelExpense IN ExpenseKinds ISA Expense
ENTITY CLASS TransportExpense IN ExpenseKinds ISA Expense

The ECUamount attribute of Expense is an example of a derived attribute. As indicated in the example, such attributes have an associated expression that is to be evaluated whenever a value is desired. Of course, such attributes cannot be updated.

Various subclasses of Expense are defined in order to distinguish their nature. These subclasses are made instances of the meta class ExpenseKinds in order to assign them a type, and hence allow them to be passed as arguments to transactions.
Finally, the following definition of Project illustrates the use of integrity constraints: in this case, the budget is checked to be 0 whenever a project instance is about to be removed, and if this is not the case, an exception is signaled. This results in the invocation of an exception handling procedure.

**ENTITY CLASS** Project WITH

- title: String
- workers: SET OF Employee
- budget: ECUs

**FINAL**

- noneLeft: True
  - (SELF.budget = 0) SIGNALS MoneyLeftoverException

**END** Project;

In order to effect changes in the information system state, or to manage, retrieve and display complex information, Taxis allows for the specification of operations.

*Transactions* represent atomic database updates or queries, and are the units of integrity maintenance and recovery. As usual, they also enforce consistency in concurrent execution, though this issue is generally ignored at the Taxis level. Transactions are specified using logical assertions which relate the values of the parameters and global variables (class extents and object properties) at the beginning and at the end of the transaction.

In order to detect dynamic integrity constraint violations, or to avoid roll-backs, transactions have Initial and Final constraints whose failure leads to the invocation of exception handlers. In fact, Taxis attempts to maintain uniformity by casting transactions in the same mold as entities. A transaction is viewed as an object, with its parameters, conditions, and actions defined by its attributes. As in Simula and its derivatives, procedure definitions become class descriptions; hence, procedure invocations are represented with instances of these classes. Most innovatively, transactions are also organized into specialization hierarchies; for example, organizing a project meeting is a specialization of organizing scientific meetings in general.

The following example shows the definition of a simple transaction to add a new expense:

---

See [BORG85] for details of an exception handling mechanism for information systems.
TRANSACTION AddNewExpense(meet:Meeting, emp:Employee,
d:Date, amo: Money, cur: Currency) WITH
PRODUCTION
ex : Expense
INITIAL
belongs : True
== (emp IsIn meet,whichProj,workers)
SIGNALS NotThereException
GOALS
: (ex.comp’ = cmp) AND (ex.when’ = date) AND
(ex.amount’ = amo) AND (ex.currency’ = cur)
END AddNewExpense;

The semantics of the property category PRODUCES is that in the final state of
the transaction the class Expense will have an extra new instance, which will be the
value of ex; the GOAL assertion prescribes the values of the attributes of this object in
this final state (as indicated by the apostrophe symbol ' ). The INITIAL condition
labelled "belongs" verifies that the person claiming the expense actually works on the
project which was holding the meeting.

To model activities with prolonged duration (e.g., running a project) as well as to
describe the systems' interaction with its users, Taxis supports the notion of scripts
[BARR82, PILO83, CHUN84, NIX084]. A script is built around a Petri-net skeleton
of states connected by transition arcs which are augmented by condition-action rule
pairs. The rules are described in Taxis but also allow reference to the passage of time,
and permit the exchange of messages following Hoare's CSP mechanism.

Scripts are integrated completely into the Taxis framework: script classes are
organized into a subclass hierarchy according to their generality/specificity, have states
and transitions defined in terms of properties, and their instances can be accessed
through the same facilities used to access instances of entity classes. Among other
things, this allows queries concerning the currently executing set of scripts.

In our expense note examples, a script would be used to describe the orchestration
of various accounting activities. Among others there would be transitions such as:

"IF there is a message with data about a new expense
THEN create a new instance of class EXPENSE;
deal with possible errors as follows ...;"

"IF 6 months have passed
THEN invoke the appropriate MonthlyProjSummary procedures;
format and print the results in a report;"

An extension to Taxis allows designers to describe user interfaces to information
systems, e.g., the query or interaction language along with the dialogues supported by
an interface, in the same uniform framework of objects, attributes and classes. Such an
extension establishes a direct link between the referring expressions used in the query
language and their referents in the database. The extension described in [PILO83,
NIXO84] also provides for modelling tools to be used for the specification of a grammar and a lexicon which are integral components of any user interface.

4. SCHEMA DESIGN AND TRANSACTION IMPLEMENTATION IN DBPL

The database programming language DBPL [ECKH85] is intended to be a productive tool for the efficient implementation of data-intensive applications. It is based on a data model supporting the prediscursive access and control of large shared data sets and has the system programming language MODULA-2 [WIRT85] as an algorithmic kernel. In this sense, DBPL is a successor of PASCAL/R [SCHM77, SM80], which was based on the relational model and the programming language Pascal.

DBPL data sets (relations) are made up of elements constructed by the data structures of MODULA-2, and can be restricted and tested by first-order logic expressions. Such predicates are used for data selection and querying, for integrity and access control, and for the partitioning of data spaces required for the scheduling of concurrent access to shared data sets [MRS84].

MODULA-2, a general systems programming language, supports on one hand the cooperative and high-level design of efficient implementations; on the other hand, it offers low-level programming features that allow for a close interaction with a given hardware environment. In summary, DBPL provides [ECKH85]:

- Set data structures intended to hold large and varying amounts of structured data objects that can be identified and altered individually.
- First-order query expressions in form of predicates to be evaluated over such data sets, and of set expressions for predicative definition of subsets.
- Predicative set selectors for the flexible denotation of subset variables allowing the access to exactly those elements fulfilling the selection predicate and disallowing any assignment of elements not fulfilling the selection predicate.
- Predicative set constructors for the definition of data sets by (nested or recursive) first-order query expressions. With constructors, DBPL provides the capacity of rule-based systems, similar to deductive databases and Prolog-like languages [JLS85]. Constructor types and variables provide a basis for integrated fact and rule management in DBPL (cf. the chapter by Schmidt/Linnemann/Jarke in this volume).
- Transactions provide safe and shared access to database variables. Concurrent transactions are guaranteed not to interfere with each other, and aborted transactions have no effect on the accessed variables.
- Modules support data abstraction, program partitioning, and software evolution, particularly by separating the definition of a module from its implementation part. A specific database module defines those data objects, e.g., data sets, that are required to be
persistent and shared by a user community. The implementation part of a database module can be altered to optimize physical data allocation and access without the need for changing the definition and implementation of user modules.

Below we present portions of the expense note example in DBPL. Note that all objects defined in a database definition module are persistent, i.e., they outlive all other modules that have access to its objects. There are three different categories of such objects: type, variable, and procedure/transaction.

Database objects of category type can be defined by any type inherited from the MODULA-2 type system. In our example database, ProjExpDb, we introduce, amongst other types, the enumeration types Currencies and Months, the subranges Money and ECUs, and various record and array types. The relation type PersonRel, for example, defines a set type over elements of type PersonRec, with the additional constraint that no two elements have the same value in the name attribute.

Database objects of category variable are exemplified by the relations Projects and Expenses; the array variable, ExchRates, demonstrates, however, that DBPL databases are not "just relational".

In the example below, a database object of the third category, procedure/transaction, is given by the function ConvToECUs, that converts money given in one of various currencies into ECUs. Our database definition module does not define transactions; however, a user transaction will be given in the application module, ExpenseManager.

Database objects can be put on the EXPORT list of a database definition module and thus be made accessible to the external world. A corresponding database implementation module holds all the internal information required for object implementation but is not to be provided at the external interface. Examples of such information are secondary indexes that support access to relations and the code that implements a transaction or a procedure.
DATABASE DEFINITION MODULE ProjExpDB;

EXPORT
  (* all defined types and procedures, all declared variables *)

TYPE
  (* Domain Types *)
  Currencies = (BF, CANS, DM, DRS, ECU, EP, FF, US$);
  ExpenseCategories = (Food, Hotel, Travel);
  Months = (Jan, Feb, ... Dec);
  Money = 0 .. 10 000 000;
  ECUs = 0 .. 10 000 000;

  (* Aggregate Types *)
  Dates = RECORD day : (1 .. 31);
    month: Months;
    year: 85 .. 95;
  END;

  Names = . . . ;
  Addresses = . . . ;

  (* Entity Types *)
  PersonRec = RECORD name: Names;
    address: Addresses;
  END;

  ProjectRec = RECORD title: LongString;
    budget, totalExpenses: ECUs;
  END;

  EmpInProjRec = RECORD eName: Names;
    pTitleSet: RELATION of LongString;
  END;

  ExpRec = RECORD name: Names;
    when: Dates;
    amount: Money;
    currency: Currencies;
    expenseKind: ExpenseCategories;
    expSlipNr: CARDINAL;
  END;

  (* Extension Types *)
  PersonRel = RELATION name OF PersonRec;
  ProjectRel = RELATION title OF ProjectRec;
  EmpInProjRel = RELATION OF EmpInProjRec;
  ExpRel = RELATION name, when, expSlipNr OF ExpRec;
  ExchRateTable = ARRAY [Currencies, Currencies] OF REAL;
PROCEDURE ConvToECUs (from: Currencies; amount: Money); ECU;

(* Extension Variables *)
VAR
  Persons      : PersonRel;
  Projects     : ProjectRel;
  EmplOnProject: EmplOnProjRel;
  Expenses     : ExpRel;
  ExchRates    : ExchRateTable;
END ProjExpDB;

Note the structural differences between the Taxis and the DBPL schema resulting from the differences in the underlying data models. The Taxis isa relationship between the classes Person and Employee, for example, is represented by the two DBPL relation types PersonRel and EmplOnProjRel, the latter containing the key of the former as the relating attribute. The extensions of those two classes are modelled explicitly by the two relation variables, Persons and EmplOnProject. In Taxis, the relationship between employees and the projects they are on is modelled by sets of project instances, while in DBPL that information is held by a set of project titles, referring to keys of project instances.

The database implementation module for the above database definition is quite short since only a single procedure was defined above and needs to be implemented.

DATABASE IMPLEMENTATION MODULE ProjExpDB;

PROCEDURE ConvToECUs (from: Currencies; amount: Money); ECU;
BEGIN
  IF from = ECU
    THEN RETURN amount
  ELSE RETURN amount * ExchRateTable[from, ECU];
END ConvToECUs;
END ProjExpDB;

The application module, ExpenseManager, makes use of objects offered by others for export. It imports, besides some standard procedures for input and output, just those types and variables from the database, ProjExpDB, that are needed for expense summary calculation. Furthermore, this module defines a selector through which access to the relation Expenses is restricted to exactly those entries that refer to the employees on a given project and to the expenses of a given month. This subrelation is imported as a read-only data object into the transaction, MonthlyProjSummary, that computes the final expense note. The record, Summary, after being initialized with a project name and a month, accepts in its substructure, expSlots, the discriminated expense sums computed by the transaction, and hands it over to the output routine.
MODULE ExpenseManager;
FROM InOut IMPORT Read, Print;
FROM ProjExpDB IMPORT ECUs, Month, ExpRel, ConvToECUs,
Expenses, EmplOnProjects;

TYPE
  ExpSlotsRec = RECORD
    foodExpense, hotelExpense, travelExpense: ECUs;
  END;

  ExpSumRec = RECORD project: LongString;
          month: Months;
          expSlots: ExpSlotsRec;
  END;

SELECTOR OfProjectAndMonth FOR Exp: ExpRel
  WITH (ProjectTitle: LongString; ReportMonth: Months);
BEGIN EACH ex IN Exp: ReportMonth = ex.when.month AND
  SOME ep IN EmplOnProjects
    ( ProjectTitle IN ep.pTitleSet AND ep.name = ex.name )
  END OfProjectAndMonth;

TRANSACTION MonthlyProjSummary (Pro: LongString; Mo: Months): ExpSlotsRec;
IMPORT CONST ExpOfInterest IS Expenses(OfProjectAndMonth(Pro, Mo));
VAR    ExpSlots: ExpSlotsRec;
BEGIN
  ExpSlots := ExpSlotsRec( 0, 0, 0 );
  FOR EACH Exp IN ExpOfInterest: TRUE DO
    DO WITH Exp, ExpSlots
      DO CASE expenseKind OF
        Food:  foodExpense :=
               foodExpense + ConvToECUs(currency, amount);
        Hotel: hotelExpense :=
               hotelExpense + ConvToECUs(currency, amount);
        Travel: travelExpense :=
               travelExpense + ConvToECUs(currency, amount);
      END;
    RETURN ExpSlots;
  END MonthlyProjSummary;
VAR Summary: ExpSumRec;

BEGIN
    (* Read project title into Summary.project *)
    (* Read current month into Summary.month *)

    WITH Summary
    DO expSlots:= MonthlyReportSummary( project, month);
    END;

    (* Print Summary *)

END ExpenseManager.

5. RELATING THE DIFFERENT LEVELS

Clearly, there are intended relationships between the world model described in CML, the design of the information system described in Taxis, and the corresponding DBPL program. Such relationships can provide useful information on the system under construction by telling software developers and users alike what the different components of the system stand for or accomplish. Of course, the relationships need to be specified explicitly.

To study these relationships, we first note that the information system can be considered to act as a model of some part of the (users' knowledge of the) real world described in CML. Therefore, the information in the system is intended to correspond in some manner to the "facts of the world". We will use for this purpose a mapping REP between CML and Taxis constructs. For example, we may wish to state that

- the Taxis class "Project" represents the CML class "Project" in the CML model in the sense that there is an isomorphism between their instances at all times: this means that the information system has complete and accurate knowledge of all the projects which are known, as well as of their attributes.

- Taxis Expenses correspond to Travel and LocalActivities undertaken by workers, but there is a delay of up to two weeks between the time when the activity occurred and the time when the corresponding expense object is created in the database.

The REP relationship can therefore be used to circumscribe the portion of the world about which the intended information system keeps data, as well as present accuracy and completeness constraints required to interpret the data. Note that some of these constraints can be enforced through the design of the system (e.g., generating reports on time), while others must rely on the proper behaviour of the external environment (e.g., people submitting expense claims on time).

On the other hand, the implementation mapping IMP between components of a Taxis specification and the DBPL program is much more strict: every data class and attribute must be correctly implemented by some expression in the DBPL program, as
must every operation. (This mapping is more like the familiar representation mapping of data abstraction languages such as CLU and APLHARD.) For example, one would specify that:

- instances of FoodExpense are implemented by instances of the relation Expenses with Category attribute equal to Food;
- the (set valued) workers attribute of Project is implemented by the EmplOnProject relation;
- the DBPL transaction MonthlyProjSummary is implemented based on an analogous Taxis transaction (not shown in section 3).

An overview of some of the derivation relationships between the CML, Taxis, and DBPL design objects is given in Figure 2. This section first examines in more detail the differences in the nature of the software layers supported here, and sketches some of the issues that arise in trying to generate a design from a requirements specification and an implementation from a design. Then, it discusses a "global KBMS" concept intended to preserve consistency of these relationships over time.
5.1. Relating CML and Taxis descriptions

A critical consideration in selecting languages for the requirements and design level is that they be compatible to the greatest degree possible. The representational framework used here (objects, attributes and predicative assertions) does provide a considerable degree of similarity between CML and Taxis. At the same time, there are significant differences arising from the distinct objectives of the two languages. For instance, in contrasting data classes in Taxis with their apparent counterparts in CML, classes of individuals, one can easily arrive at several conclusions.

Firstly, there generally won’t be a one-to-one correspondence between the two kinds of classes for a requirements and a corresponding design specification. This is the case because both CML individual and activity classes may end up being represented by Taxis data classes. Thus the dichotomy between static and dynamic aspects of the environment, as represented in a requirements specification, do not
necessarily map directly to static and dynamic components of the information system. Moreover, some individual classes will have no data class counterpart.

Secondly, as in traditional databases, attributes and class extensions are supposed to reflect a snapshot of the current state of the world; hence any historical information represented directly in CML must be recorded through explicitly associated timing information in Taxis. For example, in CML we may talk about the history of exchange rates, while in Taxis we must decide between alternatives such as storing only the current rate, storing rates for the past five years or the ones since 1978. In the process of transforming one view into the other, as advocated in [BUBE77], a number of important concepts have to be recognized such as the period of relevance, referring to the interval of time throughout which histories (i.e., past states) have to be maintained, and the period of validity, referring to the interval of time during which an attribute value applies. The recognition of these concepts, then, leads to a number of design alternatives as to the organization of and reasoning with the time-restricted entities [SNOD86].

Similarly, an important issue for CML activities is whether they should be transformed into (atomic) transactions or long-term scripts. Several criteria are relevant in deciding this point: System activities which require process communications or persistence would generally be represented in terms of Taxis scripts. Activities dealing with a small number of entities or performing atomic operations would require Taxis transactions. Other criteria that are important include the determination of starting and ending times for an activity and the presence of periodicity in its execution. The starting times and ending times, for instance, may act like sliding windows whose behaviour depends not only on the mutual dependencies among activities but also on global temporal constraints [AH85].

Concurrency control is another important issue. Detecting mutually exclusive activities would be based on sorting out temporally disjoint activities from temporally overlapping ones.

The attribute categories through which data, transaction, script and exception classes are defined constitute another source of differences. In Taxis, these are fixed and their main purpose is to detect errors in the updates being proposed. In CML they are user-defined and provide a powerful constraint mechanism. Similarly, attributes are either stored or derived (computed) in Taxis, while CML ignores this distinction, as it should, by offering assertions as a constraint mechanism for relating attributes.

5.2. Mapping Taxis Designs to DBPL Programs

The Taxis description of an application serves as a specification for the final software product by describing all possible data objects, states and state transitions contemplated in the application area. We discuss some aspects in which Taxis designs differ.

---

10 This should not be surprising, actually. Only a subset of the activities described in the requirements model are supposed to be carried out by the intended system, and only those have a transaction or script counterpart.

11 The fact that an attribute is derived implies only that it will not be directly updated by users; an implementor might choose to compute and store redundant information for the sake of efficiency.
from DBPL programs, differences that define the levels of concern of the respective languages, and that determine the mapping effort between them.

A Taxis design is largely declarative and specifies a DBPL computation exclusively in terms of an application's objects, states and related assertions. The assertions associated with data objects describe conditions that must hold at the time the object is created, deleted or are invariant over its lifetime. Individual assertions on data objects can be mapped directly onto DBPL declarations such as types or selectors. DBPL assignment statements controlled by first-order conditionals should provide high-level and fairly efficient procedural representations for the full class of assertions on data objects.

Assertions associated with Taxis transactions come in two different flavours: one category (initial and final conditions) leads to run-time test obligations, and, if violations are detected, to calls of an exception handler. The other category (assumptions and goals) are expected to hold before transaction invocation and after transaction commitment. They result in proof obligations for the transaction caller and transaction programmer respectively; these can be attacked by verification techniques such as those advocated by Hoare [HOAR69], Dijkstra [DIJK75], Gries [GRIE81], and more recently, by Hehner [HEHN84, HGM86].

Transaction programming should be a good candidate for predicative program specification and verification for two reasons. First, in many applications Taxis assertions specify only simple computations (leading, for example, to conditional assignments or set iteration); second, many of the first-order Taxis assertions can be represented directly by DBPL predicates. For example, the proof obligations in our expense note example include verifying that the Taxis derived attribute, ECUamount, declared in class Expense, is appropriately implemented by the DBPL procedure ConvToECUs, and by the way this procedure is used in DBPL transactions.

Another mapping issue arises from differences between the fundamental data types of the two languages and their type constructors. In Taxis, these include the notions of entity, set of attribute values, membership in the extent of a class; in DBPL, these are sets of arbitrary record structures, arrays, etc. In our example, the Taxis class ExchRates is known to have a small extension and therefore represented by the DBPL array structure, ExchRateTable.

Taxis designs tend to be global in the sense that any information given somewhere in one data class can be used anywhere else. Additional non-local effects originate from Taxis's incremental type and extension semantics: creating an instance of some class automatically affects the extensions of all classes higher up the generalization hierarchy. Deletion affects all extensions up and down the generalization/specialization paths.

In contrast, DBPL implementations emphasize modularity and, therefore, restrict the visibility of program information to local scopes. In cases where DBPL objects (types, variables, procedures/transactions) are to be used outside their local module, they are communicated explicitly via export and import lists. On the extension level, DBPL also tries to minimize access rights to increase access granularity and transaction independence. Information on persons and on their existence as company employees, for example, may be kept in two separate data sets and thus be accessed by
different transactions simultaneously. However, the existential integrity constraints need to be maintained. Furthermore, DBPL selectors can restrict access rights within a single data set and thus allow transaction concurrency on the level of individual extensions.

There is a wide range of alternatives to represent Taxis data hierarchies by DBPL data structures. One extreme would be a representation based on a single (universal) relation holding one relation element per Taxis data entity with null values for attributes that do not apply. The other extreme dissolves Taxis entities essentially into sets of triples <entity_id, attribute_id, value_id> and pairs <identifier, class_or_attribute_or_value_definition>. Brian Nixon used this alternative in his Taxis implementation [NIXO87]. Both extremes have apparent disadvantages in storage utilization. Adaplex [SFL83] takes a middle position and associates each node of the semantic data hierarchy with a single relation and allows for various alternatives to distribute the inherited and specific attributes over those node relations [DAYA85]. The choice between different mappings based on expected access patterns has been studied in [WEDD87].

Taxis encourages generic specifications. In part, this results from the semantics of type inheritance that qualifies transactions defined on one data class to apply to objects further down the class hierarchy. Taxis gains additional generic power from the fact that types in the DBPL sense are modelled by Taxis classes and that all classes and meta-classes are accessible through Taxis transactions. DBPL as a systems programming language supports the production of generic software to some extent. In cases in which the number of argument types for which a procedure is supposed to work is small, one may prefer, however, to generate a family of specific DBPL procedures, one for each type, preferably with the aid of an editor. Assuming that Taxis class definitions are not redefined dynamically, meta-knowledge used in Taxis designs may be compiled into DBPL programs and exploited by IF and CASE statements. Another alternative is to represent explicitly meta-knowledge in the DBPL database and then interpret this knowledge at runtime. Such decisions will be influenced by the Taxis data class definitions (their depth, degree of overlap, etc.) and the use made of them in Taxis transactions and scripts.

Taxis and DBPL also differ in the way they identify and associate data objects. Taxis uses the notion of a (system provided) surrogate for object identification, while DBPL relies on (user provided) keys for identifying set elements. Different kinds of identification mechanisms come along with other DBPL types; examples are index lists for array selection, or names for entire variables. To meet Taxis’s reference semantics we might feel the need to develop for DBPL an identification mechanism with a generalized reference semantics (DBPL selectors may serve as a starting point). The two languages differ even more in the way they build composite data objects by referring to component objects. Taxis does it through association_by_object, i.e., the component objects becomes part of the composite object. DBPL, on the other hand, provides association_by_name, i.e., it associates data objects indirectly by creating a new data object that has the associates’ keys as attribute values.

Consequently, object evaluation and object alteration inherit substantially different semantics in these languages. For an example, the deletion of an object related to others through association_by_object results in the immediate deletion of all its associates.
In the case of association, however, the names held by an object to be deleted have to be considered, one by one, in order to identify and delete the named associates individually (and this may be a recursive process). Thus, a single operation specified on a single Taxis object may result in a DBPL implementation with several operations on more than one DBPL object.

In summary, when mapping from Taxis designs to DBPL programs we see at least two major subtasks to be supported. One mapping task that, according to [DK86], may be called mapping_in_the_small, exploits that part of a Taxis design that can be mapped into DBPL declarations and statements. This task is concerned with data structure selection, type and variable introduction, module interface definition (export and import lists) including the definition of procedure and transaction signatures. Since the DBPL objects introduced by that mapping may differ from the corresponding Taxis objects, there will be also a mapping between the assertions at the two levels.

It is also part of this mapping task to collect the resulting assertions that are supposed to be met by the DBPL objects and transactions, and to develop their implementation, i.e., the actual DBPL statements.

Secondly, there is the task of implementing Taxis scripts. Since they represent long-term activities, they cannot be represented by ordinary DBPL programs. Instead, scripts have to be mapped into libraries of conditions and modules which are to be evaluated and activated by some supervising scheduler (mapping_in_the_large). For the design of a script implementation, see [CHUN84].

5.3. Global Knowledge Base Management

Each of the languages outlined in the previous three sections can be viewed as a knowledge representation framework with specific language constructs and design guidelines. The development of an information system corresponds to the construction of a heterogeneous multi-level knowledge base using these three representations. Figure 1 can also be understood as an architecture for this knowledge base.

While each of the three languages requires its own knowledge base management environment, it is the purpose of the "global" knowledge base manager (GKBMS) to coordinate the activities at the three levels of systems development, in the presence of changing requirements and conditions at any of the levels. Essentially, we can view the individual software level descriptions and their associated environments as parts of a cooperative problem-solving environment, and the GKB as a documentation knowledge base (Figure 3), intended to provide communications and maintenance services to the problem-solving environment. In this sense, the GKBMS plays a similar role in our architecture as the global Adaplex-based data manager in the MULTIBASE project [DAYA85].
The systems development environment supported by the GKBMS allows for several modes of systems development between which the designers can switch back and forth. First, there is a structured development mode which proceeds methodically from level to level; in this case, each level of design imposes a set of constraints on the design choices at the level below, similar to integrity checking in DBMS.

Second, a prototyping mode allows the designers to test tentative designs informally by translating them into executable Prolog specifications. In this case, no hierarchy can be established; rather, systems development must be seen as a process of cooperative learning between the systems designers and the KBMS in which decisions at multiple levels are made without much pre-specified order. Moreover, if decisions are made concurrently at multiple levels of design, problems may result from the fact that multiple designers with possibly conflicting goals are working on the system.

Finally, beyond the initial development process, maintenance of the information system may require subsequent changes initiated from any of the three levels. On one hand, these changes must be checked for feasibility with respect to higher-level specifications; on the other, their consequences to lower levels must be traced in a way that avoids repeating large parts of the initial development process with every change.

The problems mentioned above are far from being solved at this point; however, current plans call for studying the following components:

* Support for designing individual descriptions, and direct mapping between any two neighbors in the hierarchy that allow partial automation of the translation process. Mapping problems were already sketched in the two previous subsections, whereas specific ideas for standardized requirements analysis and design tools can be found in [BG84, JS84]. Such standard solution proposals can be incorporated into expert system-like tools called language respectively mapping assistants.

* Translation mechanisms between each language and Prolog for the purpose of rapid prototyping. These mechanisms will be attached to knowledge base debugging tools, in order to enable feedback from the prototypes to the software descriptions.
Project documentation tools that document design decisions and help designers locate their current position in a tree of design decisions [REIN84]. Design decisions will be modelled by associating justifications with each newly created or altered design object. A justification consists of dependencies which relate the new objects to those it was derived from, and to representations of the rules or tools that were used for the derivation. For example, Figure 2 shows, at a very high level of abstraction, the documented relationships between the CML, TDL, and DBPL objects mentioned in sections 2 through 4.

Belief maintenance mechanisms that encourage consistency among the design decisions made by each designer [DOYL79, DEKL86]. Such mechanisms exploit particular kinds of justification information to detect inconsistencies caused by poorly coordinated changes. In special cases, they can also propose automatic corrections. However, these systems have been able to deal only with relatively small knowledge bases (up to about 1000 objects). We are therefore investigating concepts to combine belief maintenance with abstraction and version control mechanisms, in order to reduce the number of objects to be considered simultaneously [JR87].

Negotiation support mechanisms for arriving at globally valid design decisions in case of conflict. Different designers may follow different lines of reasoning which may or may not be compatible with each other. Ideas from software configuration management [BABI86], game theory, and theories of debate [LOWE85] become relevant here.

Machine learning components are intended to acquire knowledge about design object classes and design rules semi-automatically. We are considering components that request and generalize rationales for design decisions, thus extending the knowledge bases of design rules during a systems development project [DJ85, DJ87], as well as components that learn from exceptions occurring during the use of an information systems (e.g., incomplete data structures or integrity rules [BW85]).

In summary, while the local knowledge base managers emphasize representation and execution of the outcomes of design processes, the global knowledge base manager focuses on the support of the processes themselves. Basically, this requires a meta-level description of the history of individuals and activities in the "software world" consisting of CML, Taxis, and DBPL descriptions. It appears, therefore, very natural to adopt a special dialect of CML as a knowledge representation language for the GKBMS [JR87]. An important side effect of this choice is that the combined CML model of the requirements and the resulting system could be utilized as the basis for a knowledge-based end-user interface (e.g., in natural language [BJ86]) to the DBPL-based information system. Alternatively, one can also view our approach as creating a flexible database backend for a CML-based KBMS, thus extending the concept of coupling expert systems with database management system [JV84] to the case where the database to be coupled must first be designed or adapted.
6. CONCLUDING REMARKS

Software development and maintenance is a long-term, multiple-level, and often multi-person activity. For a software environment to be useful, it must maintain not just executable code, but also the requirements and design considerations that led to it. We have sketched an environment that supports this multilayered view of software by offering three languages (CML, Taxis, and DBPL) with compatible design philosophy in their perspectives of the world (requirements analysis, system design, and efficient implementation), controlled by predicative assertions.

The environment described here constitutes the starting point for an Esprit project called DAIDA, initiated in 1986 and funded by the European Community [JARK86, JARK87]. We consider such a project as one of the first attempts to tackle programming-in-the-large through the use of knowledge base technology.

Two final comments on the bottlenecks encountered in applying knowledge representation ideas to software development. The first concerns the lack of research on performance issues for knowledge bases. Implementation of very large databases, concurrency control, error recovery, query optimization, compilation are just a few of the issues that have been thoroughly studied elsewhere and need to be re-examined for knowledge bases. This volume, along with [BM86] provide a glimpse of where we are and how far we need to go with respect to these issues.

The second, and more fundamental, bottleneck is our understanding of the nature of knowledge. As we improve our understanding of philosophical and psychological issues concerning the features of knowledge, how we memorize and use it, we will be in a position to offer better knowledge representation techniques, and by extension better software development tools. Ultimately though this enterprise is bounded by progress in Philosophy, Psychology and other disciplines of Cognitive Science. Information systems software development, and the computational environments within which it takes place, ultimately must be linked to how we process information and conceptualize the world around us.

REFERENCES


