ON THE DEVELOPMENT OF INFORMATION SYSTEMS
– FROM REQUIREMENTS MODELING TO SYSTEM DESIGN –

Maria Mamalaki, Manolis Marakakis, Michalis Mertikas,
Thodoros Topaloglou, and Yannis Vassiliou

November 1987

RCC/CSI/TR/1987/006
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

ON THE DEVELOPMENT OF INFORMATION SYSTEMS
Submitted for publication.
FROM REQUIREMENTS MODELING TO SYSTEM DESIGN

Maria Mamalaki, Manolis Marakakis, Michalis Mertikas
Thodoros Tsalpaglou, and Yannis Vassiliou

Institute of Computer Science
Research Center of Crete

Abstract

The power of today's hardware manufacturing offerings is not used effectively, considering that the development of easy to use, reliable, and maintainable software has not followed the same pace of technology advancements. A promising approach to the development of such complex software is based on the premise that knowledge representation research can be applied to the creation of methodologies, software development tools, and environments. The work reported here is a portion of a major European effort (DAIDA, Esprit no.892)\(^2\) for database intensive software development. Requirements modeling, logical system design, and the correspondence/mappings between them are outlined.

1. Introduction

Manufacturers keep widening the gap between hardware and software technology. While by 1992 we expect to have the equivalent of 100 VAX 11/780 on our desk, the easy to use, reliable, maintainable and with predictable construction software system may be missing.

In recent years computer scientists and developers have shifted attention from computation to managing information and the creation of complex systems. Even though progress is substantial, it has not reached the desired levels. The traditional life cycles have been replaced by more flexible iterative schemes, prototyping was introduced, new software tools were created and used, yet, the gap with hardware remains wide.

Possibly the most fundamental factor for successful construction of complex software is our understanding of the target application and the environment within which the final software will function. The ability to represent the resulting knowledge gives us a better handle of the data structures to be used and facilitates the evolvement and maintainability of the software. In addition, the executable specifications allow for rapid prototyping at early stages of software development.

This key observation has guided many researchers and developers to the use of knowledge representation as an integral part of the software development process. Several projects in Europe and elsewhere apply the appropriate Artificial Intelligence techniques with the expectation to succeed where traditional approaches have made little progress [Brodie et al., 1984, Brodie and Mylopoulos, 1986].

\(^1\) This work is partly supported by the European Community under Esprit contract 892 (DAIDA)
\(^2\) The DAIDA partners are: Belgian Institute of Management (IBM), Institute of Computer Science Research Center of Crete, University of Frankfurt, University of Passau, GFI, SCS Technische Automation und Systeme GmbH, and BP Research Center.
One such approach is investigated in the ESPRIT project DAIDA [Daida, 1986]. The objective is to create a software development environment for database intensive information systems [Jarke and Venken, 1987, Borgida et al, 1987]. Software is viewed as a multilayered description, which includes a requirements specification, a logical system design and an implementation. Furthermore, this description is managed as a knowledge base.

The paper presents a part of the project which retains to the work performed in Crete on requirements modeling and system design. Specifically, we focus on introducing the two basic models/languages used for these tasks, and to the issues of moving from one to the other.

The basic premise behind the work presented is that both a requirements model and a system design should depend on the environment (world being modelled) within which the resulting software will function. Therefore, the object-oriented and non-procedural formalisms used are heavily influenced by knowledge representation ideas.

For requirements modeling, SML (Systems Modeling Language) has been developed, evolving from other languages in the same family [Greenspan, 1984, Borgida et al, 1985]. SML is sketched in Section 2. A successor to Taxis [Mylopoulos et al, 1980, Chung 1984], named TDL (Taxis Design Language) is used for logical system design, and is described in Section 3.

Section 4 addresses the conceptual distance between SML and TDL. It also sketches the mapping issues for the transition between the two models/languages. Finally, Section 5 contains some concluding remarks.

2. SML for Requirements Modeling

The language SML (Systems Modeling Language) has evolved from a Requirements Modeling Language - RML [Greenspan, 1984] and a Conceptual Modeling Language - CML. [Borgida et al, 1985] over the past four years. All three languages were developed for the "highest level" description and representation of an information system that assumes a database component. Each one emphasizes specific and at times, differing aspects of the description. While RML focuses on "requirements modeling" and has a procedural nature, CML is geared to "conceptual modeling" or "world modeling" and is based on more formal foundation principles retaining a declarative specification. Lastly, SML is a superset of CML so that it can also address "system modeling" and "requirements specification."

In the sequel we give a brief illustration of SML, presenting an example, and we point out the distinguishing features from other "highest-level" languages and from the other languages in the same family (RML, CML).

2.1. Essentials of SML

In a conceptual modeling language it is important to provide rich knowledge organization principles [Borgida, 1985], which can help both the developers and the users of a knowledge base conceptualize its overall structure and understand its contents.

In SML, an object-oriented representation framework based on semantic networks is adopted and the organizational principles of classification, generalization and aggregation are supported.

As an example of how we can use SML to model an application, we present the well known example of employees and directories, where we assume that a director is an employee and the usual constraints hold (every department has a unique director, and employee salaries are lower than their directors).

We declare that an Employee is a specialization of Person, using the ISA relationship, as well as, the class Person is an instance of the metaclass EntityClass, using the IN relationship. In general, concepts of the real world are represented as objects (PROPOSITIONS), or classes of objects (INDIVIDU-ALCLASSES, ATTRIBUTECLASSES). Particular objects or classes are interrelated with other objects through properties (links on the semantic nets).

Assertions which impose constraints on the possible relationships are also encoded through properties. Property categories offer another way to impose constraints on the values of the properties.
Two types of property categories that appear in SML are the predefined ones (e.g. single, necessary) with well defined semantics and the user defined ones (e.g. socialRelation in class Person).

```plaintext
INDIVIDUALCLASS Person IN EntityClass, Humans WITH necessary, single
    name : Name
    address : Address
    socialRelation
        parent : Person
END Person

INDIVIDUALCLASS Employee ISA Person WITH necessary, single
    salary : DollarAmount
    hiringDate : Date
    department : Department
END Employee

INDIVIDUALCLASS Department WITH necessary, single
    building : Building
    staff : Employee
END Department

INDIVIDUALCLASS Researcher ISA Employee WITH necessary, single
    seniority : Seniority
END Researcher

INDIVIDUALCLASS Director ISA Employee WITH
    necessary
    uniqueDir? : (x/Employee) and x.department=THIS.department \rightarrow x=THIS
    salaryOK? : (x/Employee) and (x.department=THIS.department) and
                (x.salary > THIS.salary)
END Director

INDIVIDUALCLASS R&DDepartment ISA Department WITH necessary
    staffCategory? : (x/Employee) (x.department = THIS \rightarrow x IN Researcher)
END R&DDepartment
```
In SML we not only model static concepts of a world, but also dynamic ones, like the activity of producing a monthly expense report for an employee:

```
INDIVIDUALCLASS GenMonthlyEmpExpReport IN ActivityClass WITH
    input
    emp : Employee
    month : Month
    output
    expenseRep : MonthlyEmpExpReport
    control
    exp : Expense
    precondition
    isThereValidExpense? : (exist x/exp where x.expDate.month = month) and
        (x.emp = emp)
END GenMonthlyEmpExpReport
```

In a world model we often represent concepts that may not be represented in the systems model, but they only advance our knowledge about the world being modeled and make the representation more accurate. For instance, such a world model object may be an employee's office environment.

The distinguishing and unique aspects of SML are:

1. uniform framework – properties/links are treated as first class citizens and enjoy all the privileges of objects. Furthermore, assertions about the knowledge base or the domain are also treated as objects and constitute an integral part of the knowledge base.

2. integration of time references – all units of a knowledge base are assumed to have a time inferred to as part of their internal structure. Requirements models should provide a history of the domain [Stanley, 1986], thus emphasizing the importance of capturing the dynamics of the application in a declarative fashion by explicitly acknowledging the passing of time [Allen, 1983]. According to this, what is being modelled, is not the application domain, but the knowledge about the domain at a particular period of time.

3. executable specifications – having requirements captured in a formal and explicit manner offers the ability to reason about them and consequently detect inconsistencies.

4. dealing with exceptional cases – in many practical situations it is natural to overabstract by ignoring special cases during early stages of modeling, which lead to the need of either overriding previous generalizations or going back to modify the generalization assumptions. SML provides an exception handling mechanism which allows constraints on the structure of the knowledge base to be cancelled within a particular context.

SML, as already mentioned, is an extension of CML. As opposed to CML, SML can express the composition of compound objects as a set of smaller units. Furthermore, SML can express changes in a world (that is, the state of the system before and after a change).

2.2. SML Implementation and Status

Before the description of SML's implementation status, we refer to and motivate some strategy decisions that were made for the implementation.

a) Use of prolog as the implementation language.

Prolog offers capabilities for implementing semantic network representational schemes easily. Furthermore, it is straightforward to transform SML's assertion language into prolog clauses, while its reasoning procedure is replaced by prolog's execution mechanism. The need for rapid prototyping and a preexisting rudimentary prolog based implementation of CML are two additional reasons for choosing prolog as the implementation language.
b) Use of BOOP for efficiency.

BOOP is a set of prolog object oriented predicates, which offer the advantage of an easier and more efficient low-level representation and manipulation of a semantic network. BOOP consists of prolog predicates for defining, updating, deleting and retrieving objects, object classes, properties and isa and instance relationships.

c) View of the Knowledge-base (the model) as an ADT.

The advantage of viewing the resulting model as an Abstract Data Type is offered by the implementation of a knowledge level interface based on two operations: TELL and ASK, while other high level operations such as RETELL, UNTELL and RETRIEVE are being added [Levesque and Brachman, 1986]. The user/designer does not have to know the internal representation (structure) of the KB - the only interface to the Knowledge-base is through these high-level operations.

d) Modeling of time in SML based on time intervals [Allen, 1983].

SML extends Allen's theory by modeling elastic time intervals (open ended), and time constants for binding a sequence of actions with the real world timing system (conditional time and date). Time intervals and their relationships (e.g. before, after, during, meets, ...) are represented by a graph where the intervals are nodes and the relations edges. The representational framework is completed by a number of facts which denote the elastic intervals and a number of facts which relate intervals with time constants. The time reasoning procedure we offer in SML takes as input a goal of the form 't1 op t2', where t1, t2 time intervals and op temporal relation, or 'Tc op t1', where Tc a time constant and tries to find a path of the graph after computing its transitive closure, to validate the goal in the first case [Allen, 1983] or use point based time algebra [Villain and Kautz, 1986] to answer the second case’s goal.

e) Efficient, in the expense of generalized, consistency checking.

Consistency checking in SML consists of a syntactic check of the ISA and INSTANCE postulates [Stanley, 1986], and a semantic consistency check dealing with the integrity of the constraints. Constraints in SML are encoded using its assertion language, a function free, typed First Order Language (FOL). An efficient algorithm for consistency checking is being developed. This algorithm works locally rather than examining the whole semantic network.

The current status of SML's implementation is summarized as: a) BOOP predicates have been developed, b) a lexical analyzer and a parser, which takes SML input and produces BOOP predicates, exist. c) TELL and ASK are functionally specified but their implementation assumes the completion of the consistency checker and the temporal reasoner. d) the reasoning components (time reasoner, consistency checker) are under investigation and their formal specification will soon be finalized.

3. TDL for Logical Design

TDL (Taxis Design Language) has evolved from its ancestor, Taxis [Mylopoulos et al, 1980, Chung 1984], for the specific needs of the DAIDA project. It is an appropriate language for functional specifications of systems design [Borgida et al, 1987]. As such, it has been designed with the additional requirement of declarative nature, unlike Taxis which makes low-level implementation decisions.

Like Taxis, it is a language for semantic data modeling, influenced by knowledge representation ideas. TDL preserves Taxis' organizational concepts of data, but instead of procedural code, it only includes a predicative specification of actions to be finally implemented in the software system.

The purpose of TDL is to express the conceptual design of an information system. To realize that, it offers a powerful assertion language and the concepts of data classes, procedures (functions, transactions) and scripts. In the sequel, we present a short introduction to TDL demonstrating its use with examples.
3.1. Essentials of TDL

Data classes describe a conceptual database structure according to a "conceptual data model". In addition to the basic classes (integer, real, string, boolean) TDL supports other classes appearing in the examples below:

ENTITY CLASS Person WITH
    Unchanging
        name: String ;
    Changing
        addr: Address ;
    Unique
        id : name ;
END Person

Entity classes, are identified by a subset of attribute values. The attribute values can be either single values or sets of values. The attributes are partitioned, where each partition can be given special semantics through attribute categories. Unchanging declares the name of a person to be constant, unlike changing which declares that the address of a person can change. Unique indicates that the attribute value should determine a unique instance within the class Person. Additional attribute categories for entity classes are initial, final and invariant which associate an integrity constraint to a class. Address is an aggregate class, in which the attribute values uniquely identify each instance of the class.

ENTITY CLASS Employee ISA Person WITH
    Changing
        salary : DollarAmount ;
        department : Department ;
    Unchanging
        hireDate : Date ;
END

ENTITY CLASS Department WITH
    Unchanging
        building : Building ;
    Changing
        staff : SET OF Employee ;
END

ENTITY CLASS Researcher ISA Employee WITH
    Changing
        seniority : Seniority ;
END

Seniority is an enumerated class, e.g., ['junior, ...]

ENTITY CLASS Director ISA Employee WITH
    Invariant
        uniqueDir? : True ==
            ALL x IN Director : NOT (x.department EQ THIS.department) OR (x EQ THIS) ;

        salaryOK? : True ==
            ALL x IN Employee : x.department EQ THIS.department
            AND x.salary LT THIS.salary ;
END
ENTITY CLASS R&DDepartment ISA Department WITH
  invariant
    staffInCategory? : True ==
      ALL x IN THIS.staff : x IN Researcher ;
END

The assertion sublanguage, a First Order Language, allows either single-valued or set valued expressions (the sets can not have sets as elements). The set construct makes the assertion language powerful allowing assertions to be expressed succinctly. Assertions can be used to obtain more complex data class definitions. These classes are derived from other classes through a necessary and sufficient condition.

Two kinds of procedures are provided in TDL, transactions, which correspond to database state changes and effect database updates, and functions, which do not change the database state and return a value. Transaction updates are specified by stating constraints on the values of variables and parameters in the states before and after the execution of the transaction. The constraints are specified in the GIVENS and GOALS attributes categories respectively. In order to detect violation of integrity constraints stated on entity classes and to check dynamic integrity constraints, transactions have initial and final assertions which must be met at entry and commit time respectively. Their failure causes exceptions to be raised which are handled by invoking some specially designated procedure (exception handler). Additional allowable attribute categories for transactions are: in/out used to indicate whether the parameters communicate information into/out of the transaction; produces/consumes which add/remove that entity to/from the class that is the type of the attribute. Locals are attributes which are bound to an expression.

TRANSACTION GenMonthlyEmplExpReport WITH
  In
    emp: Employee ;
    mc: MonthClass ;
  Produces
    c1: MonthlyEmplExpenseReport ;
  Locals
    ex: Expense ;
  Initial
    isThereAcceptableExpense?: True
      == (SOME ex IN Expense)(ex.expDate.mo = mc) AND
        (ex = emp.proj.int.ex) ;
  Goals
    : (c1.mo' = mc) AND (c1.paid'amnt' =
      SUM(( ALL ex IN Expense : ex.PaidBy = emp AND
        ex.expDate.mo = mc).homeCurAmnt)) AND
      (c1.paid'.cur' = emp.addr.entry.homeCur) ;
END GenMonthlyEmplExpReport

TDL supports a temporal sublanguage with expressions involving time points, time intervals and temporal durations. The special variables $today and $now represent the current date and clock time respectively. Operations on time points and intervals are defined as functions while the infix operations Plus and Minus, as used for temporal durations, take as operands a temporal duration and a time point and give as result another time point.

Scripts in TDL describe long duration operations such as the sequencing of atomic transaction steps. Scripts are organized into classes which are assumed as generic scenarios while their instances are specific executions of that scenario.
3.2. TDL Implementation and Status

TDL's implementation is meant to provide a tool for prototyping. The implementation language is chosen to be prolog, for the same reasons that were outlined for the SML implementation, while BOOP is also used for the lowest-level predicates.

The implementation treats all TDL generic objects (Data Classes, Transactions, Scripts) uniformly. Every TDL object has a unique internal identifier which is generated automatically by the system during the creation of the object. Therefore, objects are distinct and "user independent". A side-effect of this is that renaming has no ill-effects to the model. For every generic object a prolog file is created containing information about the relations of the object with other objects - Instance and ISA relations - and all its properties (structural) and integrity constraints. This allows for easier management of information and facilitates consistency checking.

TDL imposes several implicit and explicit constraints in order to insure consistency of the model. Such constraints have to be satisfied at all times and they are checked automatically every time an update of the database is attempted. We briefly present some of these constraints:

Every object referenced anywhere must be predefined (Referential Constraint), while before the removal of an entity, it must be confirmed that it is not referenced anywhere in the database (Deletion Constraint). There are some additional constraints/rules imposed by the organizational mechanisms of TDL (aggregation, classification, generalization):

All instances of a class are also instances of its superclasses. (Extensional ISA Constraint).

A specialized class besides its own properties, inherits all properties of the more general one. The inherited properties may be specialized - their property type is a specialization/restriction of the general one (Structural ISA Constraint).

We have adopted the idea of strict property inheritance for our implementation. Therefore, all inherited properties - from all superclasses - are asserted automatically in the description of the most special entity.

Properties in TDL are grouped into categories, which impose some further constraints.

The designer, in order to insure integrity of the database with respect to the subject matter of the model, may impose constraints explicitly. These are the integrity assertions, expressed in the TDL Assertion Language and organized in a similar way as attributes.

Currently, (a) the syntactic parser has been implemented, while a first step of semantic analysis is attempted. This step includes the most basic semantic checks (Referential Constraint, checking of multiple inheritance, etc.), while in parallel, it produces facts following the rules of attribute inheritance, the generalization mechanism axioms and others. (b) An elementary environment gives the ability to the designer to create a TDL object, to update its description, to retrieve all possible information concerning it or to remove it. (c) Type checking will enhance the mechanism for consistency checking, while prototyping issues will be considered concurrently with the syntactic and environment improvements.

4. SML to TDL Mapping Issues

After the presentation of the SML and TDL formalisms, the reader may be questioning the need for having both in a software development environment. After all, they appear quite similar!

It is the basic thesis in our methodology that each language serves its own purpose - a purpose which is essential for the development of a system if it is to follow a natural and intuitive logical sequence of processes. SML describes "what is required" and TDL describes "the logical design of a system to do it." Furthermore, any similarity between SML and TDL common language design philosophy are not coincidental; they were insisted upon and influenced the development of the two languages to facilitate the transition from a requirements model (SML) to the logical design model (TDL).

At this stage, it may be argued that a mechanism, which takes as input a SML model and automatically translates it to a TDL model can be built. If that was the case, then our arguments for having both formalisms would have been weak; why then need to use both TDL and SML? Is it justified merely for the sake of style and elegance? On the contrary, we believe that there are more
substantial reasons and we immediately proceed analyzing them.

It would be incorrect to assume that all of the world model should be implemented. This is neither technically feasible nor desirable (too many environmental details would hinder the efficient operation of the final system). Of course, it is desirable to keep the "live" world model as a documentation tool and as a yardstick for the limitations of the system and its correspondence with the slice of reality it models. Furthermore, there are concepts, like transactions, that in a design model require details and design decisions specific to the system in mind and with no natural counterpart in the world or system model. Viewing this in another way, the process of creating the higher-level model which precedes that of the design model construction, can not possibly foresee such details.

With the danger of overabstraction and oversimplifying our argument, we claim that the two models (SML and TDL) viewed as sets of objects, have a non-empty intersection and also non-empty difference. However, the object-oriented environments adopted by SML and TDL, based on semantic networks and the close syntactic and semantic relationship of the two languages offer a valuable assistance to the system-designer in specifying the logical design of an information system based on the requirements specification.

TDL can be viewed as a subset of the SML language that doesn't support multilevel classification hierarchies, user defined property categories and time-handling capabilities. These features indicate some actions, that should be carried out to map a SML model to a TDL model. We namely have to eliminate attributes as objects, to suppress classification hierarchies and to express time dependent assertions into the temporal sublanguage of TDL or to transform them into transactions or scripts.

The different purposes, that each model serves, impose additional mapping actions. The system designer has to extract from the world model all the necessary information to describe update operations of the information system as well as its maintenance for a complete logical design. During this activity he should be careful with the semantics of the two models. For instance, the statement "an Esprit project may have third-party contractors" means that an Esprit project may or may not have contractors, while in system design it usually means that the specific piece of information may or may not be present.

In general, we do not expect a one-to-one correspondence of concepts between the two models, because of the conceptual distance. However, a general classification of the real world concepts in static and dynamic concepts offers a good starting point for studying mapping issues.

4.1. Mapping of static concepts

Static concepts of the world, represented in SML by Entity classes, are usually mapped in Data classes in TDL. The exact mapping depends on the semantics of each data class. For example the mapping of a SML entity class to a TDL aggregate data class is based on the following semantics of aggregation: Aggregation allows one to view an object as a composite of the objects to which it is related by properties. SML entity classes which are made by the composition of other classes should be mapped to aggregate data classes. Similar comments apply to the other TDL data classes. For example, the class Date is mapped from SML to TDL as follows:

In SML:

```
INDIVIDUALCLASS Date IN
    EntityClass, SystemClass WITH
    necessary, single
    day: DayClass
    mo: MonthClass
    year: YearClass
END Date
```

In TDL:

```
AGGREGATE CLASS Date WITH
day: DayClass ;
m: MonthClass ;
year: YearClass ;
END Date
```

We have already defined the classes Person, Employee, Director and Researcher both in SML and in TDL. Specialization hierarchies of entity classes can be mapped in a straightforward way.
Every object in SML is associated with temporal information indicating the time interval, that the object was active. Active objects are usually mapped to TDL objects, since we deal with the current state of an information system. If we want to represent in the information system entities that were active some time in the past, we have to associate explicitly an attribute with these entities indicating the time that the entities were active. For instance, we have to keep track of income-taxes of past years (or rate of interest), so that we are able to make various calculations.

Property categories are a convenient notational tool for expressing assertions in a short and well understood way. Predefined SML property categories have known semantics and they can be easily mapped to appropriate TDL predefined property categories, as we noticed in the class mapping examples. User defined property categories, viewed as constraints on the property values, should be mapped to explicit assertions defined by the system designer.

Mapping of assertions that do not contain temporal knowledge does not impose any mapping difficulty because the expressive power of TDL's assertion languages is equivalent to SML's.

4.2. Mapping of dynamic concepts

Dynamic concepts of the world, represented in SML by activity classes, are usually mapped in Transaction or Script classes in TDL, which model instantaneous changes to the data and long term processes respectively. The action of generating monthly expense reports for an employee has been modelled by the GenMonthlyEmplExpReport activity class in SML and has been mapped to the Gen-

Transaction GenMonthlyEmplExpReport creates a new instance of the class MonthlyEmplExp-

enseReport with property values determined by the assertions in the goals property category. Precondition and postcondition property categories of SML are mapped directly to initial and final property categories in TDL, which prevent the execution and commitment of the transaction unless the associated to these categories assertions are satisfied. Initial and final property categories, consequently, serve test obligations, to be met at transaction entry and commit time. Assertions, associated with given and goals property categories are to be established by the caller of the transaction and by the execution of the transaction, respectively.

Temporal knowledge represented with SML assertions, concerns either historical information or sequencing of events and activities. We already, mentioned how to retain historical information, if needed. Sequencing of activities and events in TDL is expressed through transactions and scripts.

Specifications for TDL update operations on defined entities should also be determined. These TDL objects do not have counter SML objects. Their specifications can be extracted from assertions expressing integrity constraints in the description of the entities in the world-model, or from assertions expressing degree of accuracy of represented entities.

The main difficulty in determining the specifications of a transaction is the location of the integrity constraints related to all entities affected by the transaction, since they appear in various places of the semantic network. For instance when a researcher becomes a technician then he should be removed from the staff of the R&D department, otherwise the integrity constraint indicating that the staff of the R&D department consists of researchers would be violated. If the R&DStaff is a derived class no violation has occurred, since the instances of the class are computed and not explicitly stored. Otherwise we must somehow find all the assertions referring to researchers or technicians, so that we prevent any constraint violation.

Specifications concerning degree of accuracy of represented knowledge, for instance the requirement that the represented amount of the budget left (including interest rates) of an ESPRIT project should not diverge more than 10% from the actual, determines how often the budget should be updated. Another example of accuracy requirements is the statement that the currencies should be always updated indicating daily execution of the transaction UpdateCurrencies.
4.3. Open questions and difficulties for a mapping assistant

The static and dynamic concepts in the requirements model do not necessarily map to respective static and dynamic concepts in the design model. For instance, the concept Project in the world model may be described as an activity and in the logical design may be described partially as an entity and partially as a long term activity modelled by a script. The SML class Project:

```
INDIVIDUAL CLASS Project IN ActivityClass WITH
  single input
    budget: Budget
  single output
    finalReport: ResearchReport
  single control
    proposal: ProjectProposal
    name: Name
    startDate, endDate: Date
  part
    meet: HaveMeeting
    genProgressReport: GenerateProgressReport
    reviewRep: GenerateReviewReport
  activationCondition
    : NOW DURING startDate
  terminationCondition
    : NOW DURING endDate
  preCondition
    contractAccepted?: proposal.approved="true"
  postCondition
    budgetConsumed?: budget.amountLeft=0
    (* all the budget must be consumed at the end of the project *)
  necessary
    projectReviewPositive?: (forall x/reviewRep)
    (x.proj = THIS) and
    (x.ProjContinues = "true")
    (* all reviews must approve further development of the project *)
  meetActivity?: (forall x/meet)
    (x StartsBefore startDate+3months) or
    (exists y/meet)(x StartsBefore y + 3months)
    (* there is a meeting every 3 months *)
  finalReportGeneration?: (genFinalReport after endTime)
    AND (genFinalReport before endTime + 3mo)
END
```

is naturally mapped to the TDL constructs:

```
ENTITYCLASS Project WITH
  Unchanging, Unique
  name: String;
  Changing
    budget: Budget;
    beginTime, endTime: Date;
    meet: SET OF Meeting;
    status: ProjectStatus;
END
```
The mapping issues that have been proposed should be treated as guidelines supporting the development of the logical design based on requirements specification. They can be grouped in a dialog oriented mapping assistant, that guides the designer to build the TDL model by asking him/her certain questions. An extension of the mapping assistant would be the support of automatic mapping in some simple cases and another extension would take into account the possibility of learning by the designer's initiatives. These extensions turn the mapping assistant into an advanced expert system.

5. Concluding Remarks

The suitability of knowledge representation techniques for the development of large and complex software is the basic thesis of our work. A requirements model must include an account of the environment within which the intended software will function along with the functional characteristics of the software to be build. Moreover, following an integrated knowledge representation approach in software development, a system design offers, in addition to an account of the system's behaviour, an account of how the behaviour mechanism corresponds to the world being modelled.

We presented two knowledge representation models, SML and TDL, for requirements modeling and logical system design, respectively. Finally, in addition to the richness and power of the models, a key to their successful application is the construction of an environment within which they both function and are managed. Major part of this environment is a mapping assistant, which was outlined in the paper.
Acknowledgment

John Mylopoulos and Alex Borgida have played the major role in the development of the TDL and SML languages and their precursors.

References


