A Framework for Representation, Validation and Implementation of Database Application Semantics

M. van Keulen, J. Skowronek, P. M. G. Apers, H. Balsters, H. M. Blanken, R. A. de By, J. Flokstra
Information Systems Workgroup
Informatics Faculty, University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands

Abstract

New application domains in data-processing environments pose new requirements on the methodologies, techniques and tools used to design them. The applications' semantics should be fully represented at an increasingly high level, and the representation should be subject to rigorous validation and verification. We present a semantic representation framework (including the language, methods and tools) for design of data-processing applications. The new features of the framework include a small number of precisely defined domain-independent concepts, high-level possibilities for describing behavioural semantics (methods and constraints) and the validation and verification tools included in the framework. We present examples of the use of the framework, including the use of its tools.

Keyword Codes: D 2.1, H 2.1, H 2.3, D 2.2
Keywords: Requirements/Specifications, Logical Design, Languages, Tools and Techniques

I. Introduction.

The traditional model of a data-processing application is undergoing profound changes. In the past decades, those applications were designed, implemented and used by specialized, highly trained and qualified database professionals. These specialists served as an interface between the users and their view of the problem, and the realization of the system. However, as computer networks permeate still more aspects of our professional and daily life, computing systems are designed by an ever-growing public of non-computer experts. Those experts do not possess the computer knowledge which was so important before. The group of *non-computer, domain experts* will want to design and implement high-quality, reliable database applications (encapsulating their complex semantics) themselves, and make them available to a broad user community. In a way, thus, there is and will be a shift of activities traditionally associated with computer professionals to other, non-computer specialist groups. The experts in these groups ordinarily possess a degree of knowledge in their specific domain, but they do not have specialized data modelling skills, and are not proficient in database concepts. Their detailed domain knowledge is often supported by a considerable body of techniques and methods. Those methods and techniques are based, not uncommonly, on underlying precise and formal models, required by the complex nature of users' domain-specific problems. In many cases, validation and verification of these user models is also supported. These models describe semantics as seen by the experts from their domain-specific perspective. We may thus describe them as representing user-world semantics (subjective user model of the objective, but indescribably rich real-world semantics).

The expert users perceive as uncomfortable and ineffective the difference in concepts and methods in their domains and in the software means they have to use to represent them. This difference often causes the users to abandon provided methodologies and accompanying tools. We claim, however, that this happens only when the methodologies are not sufficiently precise, and the tools are not adequately developed. These users do not see complexity as a problem, as they are used to it in their respective domain. Their problems lie in inconsequent and incomplete conceptual solutions and awkward tools. They will therefore, in our opinion, accept the complexity of the models, if they are complete and consequent, and are embedded within intuitive tools.

In summary, looking at the problem of data application design and use, we will thus be confronted with two important factors: the increasing use (in the broad sense of the word, including also design) of data-processing applications by groups of users, and the lack of specific database knowledge accompanied by detailed domain-specific expertise, often supported by strict models and methods. These two facts form a fundamental challenge for the new generation of database design tools: that of providing methods and tools serving these growing groups of users.

In the sequel, we present a design framework for representation, validation and implementation of data-processing application semantics. The framework is based on the precise modelling language, which can be subject to diverse verifications and validations, as

well as automatic translation into an implementation platform, contributing to the quality of the design *and* the implementation. In Section II, we present our view of the requirements for systems used to represent semantics of database applications. In Section III, we introduce the language used to represent semantics of applications, and in Section IV, we present the tools which are based on the language and which can be used for verifi cation and validation of the representation, as well as for its implementation. In Section V we investigate the shortcomings of our framework and discuss possible directions of future research.

II. Requirements

In answer to the challenges mentioned in the introduction, various *representation systems* (defi ned as the combination of the modelling language used to describe the semantics, the methodology and the tools supporting it) have been proposed and successfully implemented. They provided the users with varying degrees of automated support for construction and validation. These representation systems differ in two aspects: that of *scope* and *underlying semantics*. The aspect of scope of the system concerns the extent to (and the manner in) which a system can represent (varying) user-world semantics. In other words: how its (model-world) semantics maps to the user-world semantics of the domain problem. An example of that mapping can be the mapping of the user-world imprecise concept of a mechanical part, to the model-world concept of the class `mechanical_part`, with a strictly and precisely defi ned set of attributes and methods (using concepts known in object-oriented data models). Scope differences include the extent and relation of structural and behavioural representation capabilities of the models, as, for example, one system may choose to use subtyping relationships and the other not at all.

The representation systems vary also in the aspect of the underlying semantics: the manner in which a model itself is constructed. For example, in this respect systems may differ when one is based on a strict mathematical model (formal semantics), while in the other, behavioural aspects are represented informally, partly as 'signature-only,' or through a programming language notation. The aspects of scope and underlying semantics can be used to judge and classify past and present representation systems.

In the past, it was often the case that various aspects of the representation systems were unsuitable for the needs of users. They either did not possess the necessary modelling power, or they were providing insuffi cient basis for computer-supported actions using the model, such as construction of the models, their validation and implementation. Additionally, those systems often used metaphors specifi c to some domain (or even to some implementation platform), which restricted their suitability in other domains. Those differences made domain experts spend much time adapting to new terminology and customs. Some of the systems provided the appropriate scope and underlying semantics aspects, but they did not provide the tools for effi cient use, which also restricted their usage.

Deriving from this situation, we can pose a number of requirements for representation systems, which are essential in order to make them able to respond to the new challenges mentioned above. These requirements, in our opinion, are:

- preciseness,
- expressiveness,
- high-level concepts, and
- high-level tools.

The fi rst and foremost requirement is that of *preciseness*. We perceive this requirement as paramount to the ability to understand and create models by diverse user groups. In design environments of varying size, structure and lifetime, in the presence of cooperative strategies, and increased communication (also based on automatic tools for knowledge retrieval), it is very important that the semantics of the designed data application is understood unambiguously, and expressed in an exact way. As an added value, a precise model, founded on a theoretical basis, makes possible extensive automatic validations and implementations, which may shorten and ease the design process. These capabilities also contribute to the desired correctness of the representation.

The desired representation systems should also provide a high level of *expressiveness*, defi ned as the ability to fully represent the semantics of the problem. That ability makes design of the system more straightforward (and less costly). The expressive power of the model should be geared towards its perceived use; however, the model should not attempt to achieve or provide elements which are unnecessarily redundant. (Some redundancy, however, may be justifi ed by ease of use.)

The data modelling capabilities should be using *high-level concepts* familiar to users; object-oriented data models have been proposed as most closely approximating user-world semantic concepts (e.g., classes, objects, and their taxonomic relationships). The preferred representation system should, however, avoid too close an association with some specifi c domain, reflected by the introduction of domain-specifi c methodology and concepts; as this will seriously restrict its user community.

Last, but not least, the model should be embedded in an array of *high-level tools* hiding (but still using) the complexities of the model to provide extensive creation and validation possibilities. Such possibilities contribute to the quality of the resulting design.

A representation system fulfilling the above requirements will, in our opinion, accommodate the needs of diverse groups of users. In this paper, we attempt to convince the reader that we achieved this goal.

III. Modelling of the domain using the TM model and notation.

A. The TM model and notation

TM is a database specification language based on object-oriented principles (TM stands for Twente-Milano, the sites of the principal designers of the language). Its design was motivated by ideas expressed in theories like that of Cardelli [3], [11] and Reynolds [22], programming languages like Smalltalk [15], and data models like that of O_2 [7]. An important design decision for the language was that specifications resulting from its use should be subject to formal scrutiny by (semi-automatic) tools.

The language allows to define mixed models consisting of values and objects, the latter of which have unique object identities (oid's). To this end, a TM specification contains sort and class definitions, each of which functions as a common interface to a particular kind of structured value or object. The language can be further characterized by its support for various type constructors (record, variant, set and list), object identity, object sharing, structural and behavioural (multiple) inheritance, static type-checkability, and a weak notion of encapsulation. We have reported on several matters relating to the theory [5], [6] and the use of the language [4], [14] in the past.

A class (or sort) definition is constructed from three optional definition sections: a *structure section* in which, for instance, attributes can be defined, a *constraint section* in which constraints that restrict the possible structural values are defined, and a *method section* in which operations on the objects and their collections are defined. Since the core of the model is a *specification* language, we view the structure section not as an implementational definition, but rather as an integrated part of the interface of the class. Hence, an attribute declaration can be informally understood as the declaration of two methods: one to request the value of the attribute, and one to change it.

The language distinguishes itself from other proposals in the literature by the following characteristics. First and foremost, the sublanguage used for 'coding' the methods and constraints is a non-imperative, functional language designed to be used for elementary object manipulation *and* more intricate set manipulation. As such, it contains a full object query language, of which we present example expressions below. This 'coding' sublanguage was also designed bearing the required method inheritance in mind: one can define methods in a functional style that can be reused for subclass objects. All this leads to a specific typing scheme for methods that implements covariant method types based on the type variable notion of **selftype**. Therefore, although the language is functional, we do not use functions, but rather families of functions, to represent methods. The language does not include a relationship construct, which can be, however, modelled in TM through the use of attributes and constraints.

Benefits of a non-imperative language over such now widely spread languages like C++ is that we can build tools to analyse the specification code and prove their correctness (an important step in that direction was the construction of the TM proof checker [9]). Other advantages of the use of a high-level declarative language is its independence of the particular implementation platform as well as improved maintainability of the design.

B. Modelling the user domain

To illustrate the process of the creation of a precise model of an application from a user-world description, we briefly present an example of a user-world description of an electrical network, and subsequently provide its representation in TM.[1]

The energy management system (and any technical information system in general) poses specific requirements for safety and reliability, thus asking for a disciplined approach to software engineering. We will explain the way a user-world model of the application domain is transformed into a precise model expressed in TM. Furthermore, we will attempt to assess the difficulties in the transformation process and the benefits of the new descriptions employing precise semantics.

In electrical network management, domain specialists often distinguish between the structural and behavioural part of the model. The structural aspect of the network describes the topology of the network, as well as various properties (attributes) characterizing its nodes and edges. The values of attributes are often gathered from the electrical network by means of measurements. In the behavioural part, procedures and actions performed by dispatchers (network operators) are described.

---

1. The example originates in the demonstrator application of the IMPRESS project [14]. It concerns a database representing a subset of the electrical network, managed by one of the largest electricity distribution companies in Spain, IBERDROLA [19]. In the scope of the project, a database schema (together with methods and constraints) was designed and validated using the TM language and its tools.

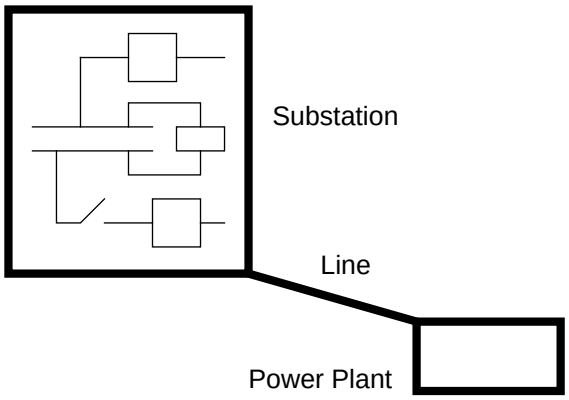


Figure 1: Electricity network.

The structure of the electrical network is modelled by the domain specialists in the form of installations connected by lines (Figure 1). The installations include various kinds of substations and power plants (hydraulic, thermal or nuclear). Substations are responsible for energy transport, transformation, and distribution; they 'drive' the electric energy from the generation points (power plants) to the clients. Power plants provide the electrical energy to the network and lines allow for physical connections between the installations.

Substations, power plants, and lines are complex structures: they are themselves composed of installation elements (bar systems, protections). Those elements in turn are composed of equipments: bars, switches, breakers. The domain experts therefore model the network at different levels of detail: *network element level*, *installation element level*, and *equipment level*. Elements at various levels are often grouped in aggregation relationships, e.g., a substation is composed of a set of installation elements, a bar system is composed of a number of bars. Based on these modelling principles, domain specialists model specifi c element types at various levels: e.g., they specialize double-bar substations among substations as those having a double bar installation element.

The structural part of the user model is complemented with the behavioural part, in which possible manoeuvres (actions) concerning elements are described, as well as the broader concept of operator procedures. The procedures are performed by network operators (dispatchers), and concern tasks such as switching, monitoring network safety, monitoring and optimization of network resources, fault actions, and preparation of reports. These procedures are composed of sequences of (elementary) retrieval and update actions. The actions include opening and closing of diverse equipments: bars, switches, etc. The actions typically change the values of attributes of network elements. The actions also include diverse retrieval queries, which are used by dispatchers to assess the network state.[2] These queries include connectivity queries as well as hypothetical update ("what-if") queries.

The user-world semantics described above has been registered in a quasi-formal way through extensive and redundant descriptions of element structures and the actions performed on them. In the presence of increasing trends to automate yet more aspects of the network, there is a recognized need within the domain for a more systematic approach to the modelling problem.

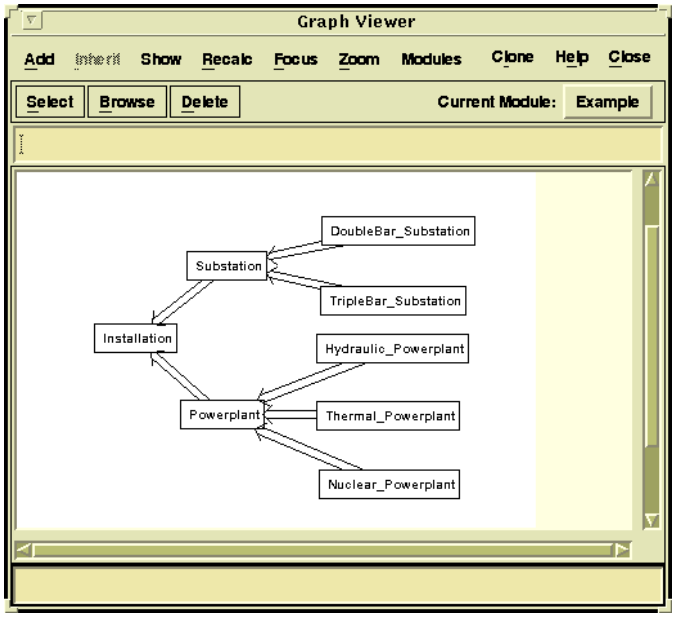
The construction of representation systems can, in our opinion, benefi t from the observations made on how domain experts create their models of the user world. In many cases, the user-world semantics is not precisely described; it is often summarized in manuals, procedures, and in the unwritten experiences of the experts themselves. Such a representation is extremely diffi cult to validate, implement, and maintain, and this fact is often recognized by the domain experts themselves. In many cases, the semantics is directly 'implemented' in languages, in which the modelling power is insuffi cient to express its complexity. There is thus a recognized need to represent the imprecise semantics of the user world in precise terms, and there have been different domain-specifi c models and tools designed for that purpose (e.g., representation systems in CAD: constructive solid geometry, boundary representation). The TM model attempts to provide suffi cient means for semantically-rich modelling through a number of domain-independent structural and behavioural primitives, providing a strict and unambiguous representation of the semantics.

In the following paragraphs, we present how precise semantics can be constructed using the TM model, and we formulate our observations of the conceptual diffi culties arising during such transformations, and the benefi ts arising from them.

A well-accepted general guideline for object-oriented modelling is to start with identifying classes and the subtype/supertype relationships between them. In the example, it is not diffi cult to see that concepts like 'Installation', 'Substation', etc. are prominent candidates for being specifi ed as a class. In this process, it is possible that new (abstract) classes are found (e.g through generalization), that do not have direct counterparts in the original user models. We observed that this fi rst step in object-oriented modelling is straight-

2. Example queries are: "provide references to all manuals that are relevant in this fault situation," "give all protections of the Aldeadavila substation that are connected to line Villarino-Aldeadavila and that are open," "give elements protected by element X."

forward for domain experts, as they already use similar semantic concepts in their user models (e.g., the use of words like 'is a kind of'). An illustration of how a part of the example can be modelled in TM, is given in Figure 2. The fi gure shows one of the TM tools (see following section) with which one can specify the structural part of the model using diagrams. The double arrows represent subtype/supertype relationships where the arrow points from the subclass to the superclass.



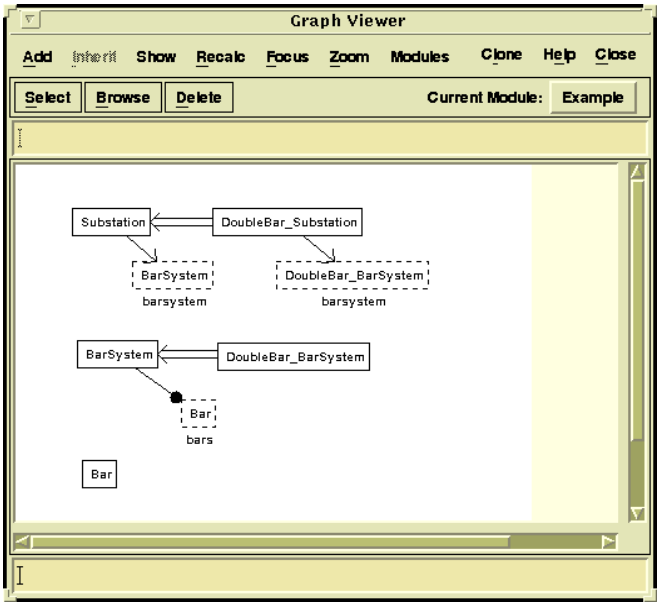

```
module Example1
    Class Installation
    end Installation
    Class Substation ISA Installation
    end Substation
    Class Powerplant ISA Installation
    end Powerplant
    Class DoubleBar_Substation ISA Substation

//  other classes are specified similarly

end Example1
```

Figure 2:Excerpt of the class hierarchy and the corresponding fragment of the TM specifi cation.

A logical next step in object-oriented modelling is the addition of attributes. Attributes can be used to represent various kinds of common properties among instances of classes, for example, connected_to or drives_power_to. How hard it is to place the attributes is very much dependent on how the class-hierarchy is structured. It is therefore likely that during this phase one would feel the need to rearrange and fi ne-tune the class hierarchy. We have observed that domain specialists benefi t from the graphic view of the schema, as it provides them with a means to organize their detailed knowledge of the domain. The representation also enhances their understanding of the model, as it often shows why certain properties appear in the model multiple times.




```
module Example2
    Class Substation
        attributes
            barsystem:BarSystem
    end Substation
    Class DoubleBar_Substation ISA Substation
        attributes
            barsystem:DoubleBar_BarSystem
    end DoubleBar_Substation
    Class BarSystem
        attributes
            bars:P Bar
    end BarSystem
    Class DoubleBar_BarSystem ISA BarSystem
    end DoubleBar_BarSystem
    Class Bar
    end Bar
end Example2
```

Figure 3:Attribute specialization.

The user model usually contains a signifi cant level of inaccuracy and redundancy. The inaccuracies are easily found (and removed) by exploiting the preciseness of the specifi cation. The amount of redundancy can be reduced by the proper structuring of the class hierarchy. Figure 3 gives an illustration of one of the more advanced attribute modelling facilities TM offers: it shows how one can represent the fact that a bar system in DoubleBar_Substation has to be a DoubleBar_BarSystem.[3] The DoubleBar_Substation class is a specialized version of Substation, and simul-

taneously, its attribute barsystem is a specialized version of the same attribute in its superclass.

TM offers extensive facilities for enhancement of the model with precise descriptions of behaviour (methods) and of the valid states of the objects (constraints). The language contains an expressive and high-level expression language that is used to specify both constraints and methods in a uniform way. Constraints are used to restrict the number of valid object states to ensure that the class's universe is in accordance with the user-world model. In the example of Figure 2, BarSystem contains a set of bars. Consequently, a DoubleBar_BarSystem contains a set of bars as well. However, DoubleBar_BarSystems, containing a number of bars other than two, 'make no sense' in the user-world semantics. Therefore, it is obvious that in the model, we would like to express that DoubleBar_BarSystems can only have two bars. In TM, this is specified as the object constraint[4] for this class "**count**(bars)=2", meaning that the number of elements in the set-valued attribute bars must be equal to two.

The same language is used to specify the methods, which express the retrieval and update operations performed on objects. TM provides a rich set of expressions, examples of which are:

- *explicit expressions*, like for instance
  < name = "Aldeadavila, voltage = 280>
  which denotes a two-field record, or
  true
  which denotes the boolean value for truth, or
  [1, 3, 2, 8]
  which denotes a list of four integer numbers.
- *selection expressions*, like for instance record field selection in
  < name = "Aldeadavila", voltage = 280>.name
  which evaluates to the string " Aldeadavila", or list selection, as in
  head([1, 3, 2, 8])
  which evaluates to integer constant 1.
- *predicative expressions*, which allow the description a set of elements by characterization, as in
  {x:<a:**int**, b:**int**> | x.a = x.b **and** (x.b=1 **or** x.b=2)}
- *iterative expressions*, which take a set or a list and a function as arguments and evaluate that function for each element of the set or list. The result obtained is again a set or a list. Hence, the expression

  collect <voltageInKv = x.voltage/1000>
  for x **in** [<name="Aldeadavila", voltage=280>,
  <name="Villarino", voltage=380>]
  renders a new list
  [<voltageInKv=0.28>, <voltageInKv=0.38>]

The expression language of TM is statically typed, making it possible to determine at compile time the types of all results of expressions and their subexpressions.

One of the queries in the electrical network management example was " Give all protections of the Aldeadavila substation that are connected to line Villarino-Aldeadavila". This concrete query can be generalized by parametrising it with the substation and the line in question, and can be specified in TM as the following method (assuming there is a protections attribute on the class Line and a IsElementOf method on the class Installation).

```
ObtainProtections(in s:Substation, ln:Line,
    out P Protection) =
{p in ln.protections | IsElementOf[s](p)}
```

The expression can be read as " collect those p from the set-valued attribute protections of ln, that are an element of s".

Domain experts who transform their user-world view of data application to a precise model may encounter difficulties arising from the inherent differences in which reality is described in both worlds. These difficulties manifest themselves when the concepts used in the domain cannot be directly represented in the model. We believe that TM incorporates a number of domain-independent concepts that can be successfully used by experts in many domains. The initial difficulties that the domain experts may experience stem from the precise character of the representation system. Those difficulties are outweighed by the benefits, which can be summarized as follows:

- the preciseness of the representation often causes the vague and redundant user-world models to become exact and compact
- misplaced concepts in the user model are positioned correctly
- the user view is transformed into an organized and structured model
- due to the precise and unambiguous character of the model, extensive validation and verification has become possible

IV. The Database Design Toolset.

As presented above, we believe that a precise representation of data application semantics provides the designers with important advantages, such as increased modelling capabilities and validation possibilities. However, the impact of those advantages on

3. In the diagram, an arrow represents an attribute. It points to its type and its name is written under the type. An arrow with a filled circle at the end represents a set valued attribute. The dotted line around an attribute's type indicates that the type is defined elsewhere in the diagram (i.e. it's a reference).

4. An object constraint is a condition that must be valid at all times for any object of the class.

users will be restricted if they are not embedded within easy-to-use tools suitable for use by non-computer experts, who do not necessarily have to be proficient in formal techniques. Therefore, for wide acceptance of the language, it is imperative that it is accompanied by an array of tools. The tools provide an easy-to-use interface to the various facilities of the representation system. The tools should be intuitive, but they should still provide all the advantages of the underlying language: its preciseness and expressiveness. In the past, various proposals for design tools were presented in the literature, with varying scope and underlying semantics properties (DBE [8], EcrinsDesign [1], TRAMIS [18], MOSAICO [20], to name just a few). Those tools often include only some of the design and validation techniques, such as graphical schema design, high-level modelling of methods and constraints, or prototyping. All of them are available together in our toolset. The contribution of our system amounts to the founding of the design and validation framework on a firm theoretical basis. This base enabled us to integrate different design, verification and validation techniques in one framework. The nature of the framework enabled us to extend the framework with tools uncommon in the field of data application design, such as the safeness checker.

In the following paragraphs, we describe the tools for creation of TM specifications, validation of their various aspects, as well as their translation to an implementation platform.

A. Creating specifications.

In Section III, we have outlined the process of transformation of user-world semantics into model-world semantics expressed in the TM language. During this process the domain experts map the domain concepts into the (semantically more precise) concepts of TM. We have observed that this mapping is not always straightforward, as during the process of precise specification, new semantic relationships are discovered. This is a natural consequence of the fact that the user-world semantics is often redundant and imprecise. That redundancy, however, is not a desired characteristic of models, as redundant designs are not necessarily the ones easiest to understand and communicate. Therefore, it is important that the experts possess the necessary means for careful examination of their models, so that their undesired characteristics are quickly discovered. To that end, in the Database Design Toolset the experts create the specifications in an intuitive, graphical way (see Figure 2 and Figure 3), which makes it easy to observe semantic relationships, detect redundancies and perform appropriate changes. The graphical notation used for modelling provides all structuring primitives available in the underlying TM language, representing them through orthodox graphic primitives. In this way, the designer is not forced to learn the syntax of (yet another) structuring language. In case of behavioural semantics (constraints and methods), the designer uses the expression sublanguage of TM to precisely define their semantics. The sublanguage is not defined with a specific execution paradigm in mind (being a specification language and not a programming language), and provides a rich set of expressions for manipulation of various type structures (e.g., collections). As the behavioural semantics is expressed at the same level of preciseness as the structure, it can be subject to the same level of validation.

In large design projects, it is often important that the errors in the model are detected as soon as possible, in order to limit and simplify the necessary corrections. To that end, the designer should be able to validate the model at all stages of its development. The possible validations of the model, which are all consequences of its precise foundation, are the verification of the model against the typing rules, verification of the behavioural semantics against the requirement of safety (absence of expressions with infinite results), validation of the semantics of constraints and methods through prototyping as well as verification of transactions in terms of constraint preserving properties.

B. Verification and validation of specifications: type checking, prototyping and safety detection.

The advantage of precise semantic representation becomes particularly evident when we consider the verification and validation possibilities which are supported by a representation system. Basing the system on a sound foundation greatly extends these possibilities.

The verification possibilities of TM are realised by tools that check two important aspects of TM specifications, namely:

- consistency of the created types, constraints, and methods with respect to the syntax and typing rules.
- safety of the specified constraints and methods, defined as the guerantee that a certain constraint and method will not generate infinite results. The notion of safety has therefore been defined in a similar way as in Elamsri [13], while in Ullman [23] safe expressions have been defined as a subclass of domain-independent expressions (those whose result is not dependent on the values that are not mentioned in the expression and associated relations).

TM is based on a strict typing discipline, which can be used to precisely determine the types of all structural

and behavioural elements of the specifi cation. This discipline is expressed in a collection of typing rules, which are used by the *type checker* to assess the type consistency of a specifi cation. Any type errors detected are reported to the designer, who can then perform appropriate corrections in the specifi cation. The type errors detected include erroneous syntax and typing of expressions, as well as improper use of subtyping and attribute specialization.

The rich expression sublanguage of TM provides constructs which can be used to form expressions (possibly) evaluating to infi nite collections. In line of the example presented above, let us suppose a designer wants to specify a constraint that ensures that all electrical connections are reflexive. One could specify that as in Figure 4 (fi rst expression). The **in** expression is a boolean expression testing the membership of an object (in this case the object itself: **self**) in a set. We observe that the right hand-side set expression of the **in** expression is infi nite, because it is constructed from the type ElecComponent representing all instances of that type that *can* exist. Such expressions are useful at initial stages of specifi cation, when we are not concerned with implementation possibilities, but they could, however, cause problems for a 'naive' code generator. However, the whole expression *can* be evaluated within fi nite bounds by rewriting the expression fi rst in a safe way (see Figure 4 - second expression). In TM, we are therefore confronted with a situation when an expression can be safe even if its subexpressions are unsafe. The Safeness Detector analyzes this class of potentially unsafe expressions, and attempts to decide whether a given expression of that class is safe.This is done through trying to rewrite of the given expression into another expression that it is safe.

```
MutuallyConnected:      // potentially unsafe
    self in
       {e:ElecComponent |
         forall c in e.connected_to |
         e in c.connected_to})

MutuallyConnected:    // safe
    forall c in self.connected_to |
       (self in c.connected_to)
```

Figure 4:Example of a potentially unsafe and safe constraint

Apart from verifi cation possibilities, the precise semantics of TM expressions makes various validations possible, such as early detection of semantic errors in the specifi cation. Such errors manifest themselves when the specifi cation is type-correct and safe, but its methods and constraints do not return the intended results. Detection of semantic errors is desired at the earliest possible moment in the design process and defi nitely before the implementation phase, when the necessary corrections can be very costly to perform. This detection is realized through prototyping, when constraints and methods can be evaluated on test data and their results can be examined. As TM expressions are complex, the *Prototyping Environment* enables creation of objects, evaluation of methods and constraints, and navigation through (complex) results of expressions and subexpressions (see Figure 5). In this way, the designer is able to detect semantic errors and perform appropriate changes in the specifi cation.

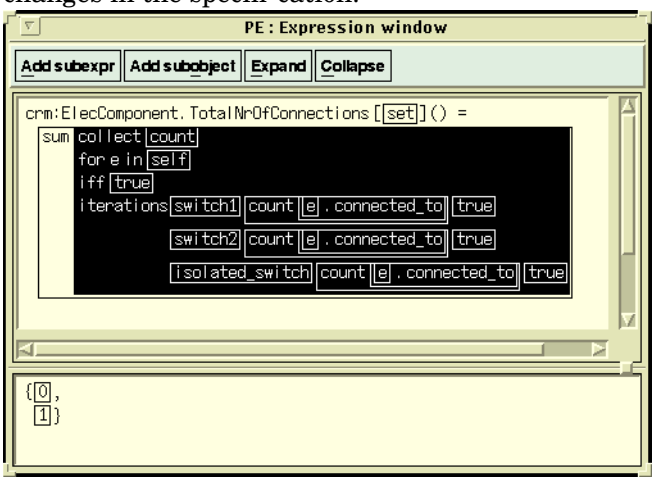


Figure 5:Example of the result of a method evaluation

Summarizing, the TM tools provide a basis for verifi cation and validation of designs, including type and safeness checking, as well as prototyping. All these tools serve to increase the level of (syntactic and semantic) correctness of specifi cation. Afterwards, the specifi cation can be implemented using automatic translation into a target language.

C. Implementing specifi cations.

In most design projects, the design is meant to be implemented on a certain implementation platform. In case of simple designs, the semantics of the application is often simple enough to be directly implemented in a programming language. Even in small projects, however, this can lead to diffi culties in its extension and evolution and in understanding the application by larger teams. As the scale of the projects grows, the need for high-level, independent and powerful representation becomes evident, and more and more design activities are performed at a higher level. If these high-level representations are not precisely defi ned, the task of their translation to an implementation becomes increasingly complex and more diffi cult to realize by teams of programmers. When a representation system fulfi lls the requirement of preciseness, the semantics is expressed unambiguously; in that case

the road is open towards an automatic translation to an implementation platform.

Among the tools based on the TM language, we have provided an automatic translation of TM specifi - cations into a persistent programming language SPOKE. However, the use of existing persistent programming languages and database systems as target languages for this translation causes problems in cases where there are semantic differences between TM concepts such as subtyping, and their representation in the target language. These issues have to be taken into account when target platforms are considered, and we are now studying possible target platforms, on which TM concepts can be represented directly. An example of such a platform is a dedicated layered TM expression evaluator, which we are currently implementing. The evaluator is constructed around an abstract machine, which enables precise representation of the semantics of expressions.

D. Example of toolset usage

The integration of the toolset and the iterative character of the design process can be conveniently illustrated by means of a possible design scenario:

- The designer graphically creates the classes, inheritance links, attributes, constraints, and methods using the Graphical TM interface.
- The designer type-checks the specifi cation, which may result in a number of syntax and typing errors.
- The designer returns to the Graphical TM Interface to correct those errors. After they are corrected, she/he type-checks the specifi cation again until success is reached.
- The designer decides to check whether a specifi c method is specifi ed correctly (in the semantic sense). She/He does this by opening the Prototyping Environment, creating some test data, and evaluating a method or a constraint using the test data. She/He then examines the fi nal and/or temporary results of the expression body to detect possible semantic errors. If there are errors, she/he corrects them using the Graphical TM Interface.
- The designer can check whether some of the specifi ed constraints and methods contain unsafe constructs, such as infi nite sets. She/He does this by invoking the Safeness Detector, which analyzes the constraints and methods to detect unsafe constructs.
- Once the designer is satisfi ed with the design, she/he uses one of the Code Generators to generate optimized programs realizing the specifi cation.

We feel that the above design process is not specifi c to some methodology, but can be encompassed within different methods. The above scenario is the most unstructured possible; specifi c design methodologies may pose restrictions on the way each of the tools are used. Thanks to the extensible architecture, an organization can structure the design process differently or provide other tools, such as versioning tools, authorization mechanisms, expert system shells, and cooperative design environments.

E. Components and status of the toolset.

Various tools mentioned in the above subsections have been implemented in the scope of the IMPRESS project [14]. The tools have been implemented on a Unix platform, with the use of Motif window libraries for graphical user interfaces. The tools which have been implemented are:

- the TM type checker, which takes a TM specifi cation as input and checks its syntax and type consistency,
- the Graphical TM Interface, which aids in creation of TM specifi cations, using diagrams presented above,
- the Prototyping Environment, which validates the methods and constraints, enabling graphical browsing of expressions and complex objects, as well as creation of test databases,
- the Safeness Detector, which detects unsafe expressions within specifi cations. The tool uses Life [2] to rewrite expressions,
- the Code Generator, which generates executable code from a specifi cation. In IMPRESS, the execution platform was a persistent programming language SPOKE; currently we are constructing a dedicated evaluator for TM expressions, in which we can more precisely represent the semantics of TM,

Currently, we are constructing public-domain versions of the tools with the intention of providing them to the interested research users.

V. Conclusions and future research.

In the introduction, we formulated a number of requirements which in our view are essential for the acceptance and use of semantic representation frameworks in new application domains by growing user communities. We stated that those frameworks should exhibit the properties of preciseness and expressiveness, should use high-level concepts and be embedded within high-level, easy-to-use tools.

Subsequently, we presented the TM modelling language, which, in our opinion, provides a *precise way of expressing the semantics* of complex applications in diverse domains. The expression sublanguage of TM offers a high-level, declarative way of expressing the behavioural semantics (constraints and methods). The TM language is domain-independent and offers a small number of high-level concepts. Its object-oriented flavour leans itself well for modelling complex

application domains. The preciseness of the language has been the basis of the *validation and verification tools*, such as the type checker, prototyping environment and the safeness detector presented in Section IV.

The extensible structure of the framework makes possible addition of new tools. An example of such a tool is the Proof Tool, which can check whether a method violates the constraints specified for the database. This will enable fast detection of specification errors without any need of prototyping. In the verification process, the Proof Tool will thus ensure robustness of methods and constraints. During the project, we tried to use available theorem provers (ISABELLE [21], HOL [16]) to assist in this task, and we plan to further pursue this effort in future.

We also conduct active research into further extensions of TM, and we scrutinize theoretical foundations of the language. During the project, new modelling primitives have been added to enable modular design, as well as a better formal foundation for method inheritance. We also examine the use of TM and the tools in new application areas, such as distributed databases, in combination with languages and tools in this area such as LOTOS [10]. These and other activities, resulting in high-level tools are meant to make the framework more suitable for use by non-computer experts.

Further research is also being conducted in the area of translating TM into executable languages, e.g., using the principles and tools of algebraic optimization. We are investigating the translation of TM into industry-wide standard data models and query languages, such as ODMG [12] and its query language OQL. Such translations often have to overcome significant differences between corresponding underlying semantics and type systems, therefore we also investigate the process of evaluation of TM expressions in a way more closely reflecting their semantics.

All the above mentioned activities can be placed in the broad perspective of providing a suitable representation of data-application semantics: a representation which can be created by the concerned users themselves, which offers them suitable modelling power, which equips them with validation tools, and which helps them in implementation. Through those activities, we hope to help the application developers to answer the challenges which are posed by the complex semantics of specialized technical domains.

VI. Acknowledgments

We would like to thank all members of the IMPRESS project for helpful cooperation throughout the project. In particular, Ricardo Capobianchi, Florence Ardorino and Marc Mautref of Alcatel AAR were actively involved in the development of the methodological guidelines of IMPRESS. Ghislaine Eble, Cyril Autant and Yves Garrier of Alcatel ISR cooperated with us in the design and implementation of TM to SPOKE translation. Enrique Burguera, Fernando Marquinez and Inaki Angulo of IBERDROLA introduced us to the intricacies of electrical network management. Reinier Boon and Karina Weening helped us with the implementation of the Database Design Tool. Alexander Bosschaart and Bart Termorshuizen developed the first functional specification of the Design Tools.

VII. References

[1] F.Adreit, M. Bonjour, " EcrinsDesign: A Graphical Editor for Semantic Structures", Proc. 3rd Int. Conf. CAiSE'91, Trondheim, Norway, 1991, pp. 264-284.

[2] H. Aït-Kaci, " An overview of life", in J.W. Schmidt and A.A. Stogny, eds., Proc. of the 1st Int. East-West Database Workshop, Springer-Verlag, 1991, LNCS 504, pp. 42-58.

[3] A. Albano, L. Cardelli, R. Orsini, " GALILEO: A strongly-typed, interactive conceptual language", ACM Trans. on Database Systems 10, June 1985, pp. 230-260.

[4] H. Balsters, R. A. de By & R. Zicari, " Typed sets as a basis for object-oriented database schemas", in Proc. Seventh European Conf. on Object-Oriented Programming (ECOOP), July 26-30, 1993, Kaiserslautern, Germany, 1993.

[5] H. Balsters & M.F. Fokkinga, " Subtyping can have simple semantics", *Theoretical Computer Science* 87 (September, 1991), pp. 81-96.

[6] H. Balsters & C.C. de Vreeze, " A semantics of object-oriented sets", in The Third Int. Workshop on Database Programming Languages: Bulk Types & Persistent Data (DBPL-3), August 27-30, 1991, Nafplion, Greece, P. Kanellakis & J.W. Schmidt, eds., Morgan Kaufman Publishers, San Mateo, CA, 1991, pp. 201-217.

[7] F. Bancilhon, C. Delobel, " Building an Object-Oriented Database System - The story of O_2", Morgan Kaufman, 1992.

[8] D. Bitton, J.C. Millman, S. Torgersen, " DBE: an Expert Tool for Database Design", Proc. 3rd Int. Conf. CAiSE'91, Trondheim, Norway, 1991, pp. 240-264.

[9] R. J. Blok, " A proof system for FM", M.Sc. Thesis, University of Twente, 1993.

[10] T. Bolognesi, E. Brinksma, " Introduction to the ISO Specification Language LOTOS",*Computer Networks and ISDN Systems*, vol. 14, 1987, pp. 25-59.

[11] L. Cardelli, " A Semantics of multiple inheritance", *Information and Computation* 76, pp. 138-164, 1988.

[12] R.G.G. Cattell, " ODMG-93: A Standard for Object-Oriented DBMSs", Proc. of the 1994 SIGMOD Int. Conf. on Management of Data, pp. 480-481.
[13] R. Elmasri, S.B. Navathe, " Fundamentals of Database Systems", Benjamin/Cummings, 1989.
[14] J. Flokstra, M. van Keulen, J. Skowronek, " The IMPRESS DDT: A Database Design Toolbox Based on a Formal Specification Language", Proc. of the 1994 SIGMOD Int. Conf. on Management of Data, pp. 506-507.
[15] A. Goldberg, D. Robson, " Smalltalk--80: The language and its implementation", Addison-Wesley, 1993.
[16] M. Gordon, " HOL - A Proof Generating System for Higher-Order Logic", in *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam, Kluwer, 1987.
[17] P.W.P.J. Grefen, R.A. de By, P.M.G. Apers, " Integrity Control in Advanced Database Systems", Bull. Techn. Committee on Data Engineering, vol. 17, no. 2, 1994, pp. 9-13.
[18] J. A. Gulla, G. Willumsen, " Using Explanations to Improve the Validation of Executable Models", Proc. 5th Int. Conf. CAiSE'92, Paris, France, pp. 118-143.
[19] IMPRESS-IBD-REPORT-W5, " Physical model of the electric network", IBERDROLA, 1993.
[20] M. Missikoff, M. Toiati, " MOSAICO - A System for Conceptual Modeling and Rapid Prototyping of Object-Oriented Database Application", Proc. of the 1994 SIGMOD Int. Conf. on Management of Data, pp. 508-509.
[21] L. C. Paulson. " Introduction to Isabelle", Technical Report 280, University of Cambridge, Computer Laboratory, 1993.
[22] John C. Reynolds, " Three Approaches to Type Structure", in *Mathematical Foundations of Software Development*, H. Ehrig et al., eds., Springer-Verlag, 1985, Lecture Notes in Computer Science, no. 185, pp. 97-138.
[23] J.D. Ullman, " Principles of Database and Knowledge-base Systems", Computer Science Press, 1989.