Validating Database Components of Software Systems

Perdita Löhr-Richter

Technical University of Braunschweig
Dept. of Computer Science
Gaußstr. 12, D-3300 Braunschweig
Germany
Tel.: ++49/531/391 7447
Fax.: ++49/531/391 3298
email: loehr@idb.cs.tu-bs.de

Andreas Zamperoni

Leiden University
Dept. of Computer Science
Niels Bohrweg 2, NL-2333 CA Leiden
The Netherlands
Tel.: ++31/71/27 7103
Fax.: ++31/71/27 6985
email: zamper@wi.leidenuniv.nl

Abstract

In this article, we present a framework and a concrete method to support validation of central database components in application software. We explain how validation can cope with both the need for a formalized evaluation of correctness as well as the need for prototyping high level database specifications in early development phases. We define different levels of correctness suitable for database specifications and show how these levels can be checked by our validation method. For this purpose, we describe both the method's formal basis and explain step-by-step the corresponding pragmatic algorithm to actually generate test data and validate the specification.

Keywords: consistency, correctness, database-centered application software, extended ER model, high level database specification, prototyping, test data, validation method

1 Introduction

In the area of software engineering, testing has a well-known tradition and corresponding theories exist [1, 2, 19]. These approaches focus on tests of programs or programming systems, i.e., dynamic structures. In this respect, most of them derive test data to test for functionality, coverage criteria, boundary cases, etc. . Nowadays, a lot of software systems rely on central, complex structured database components. This demands to test and validate the underlying database component before dynamic structures can be tackled. But testing data structures requires new notions of correctness, because the traditional ones suitable for dynamic structures don't hold for data structures. Therefore, testing and validating database components requires the definition of specific correctness properties and the development of matching validation strategies.

Few approaches exist which work on this topic. We classify them into four groups: *General approaches* which investigate general validation principles for classes of specifications [5]. *Logic based approaches* which consider a database specification as a collection of predicate logic formulas which has to be satisfied [3, 16]. *Proof approaches* which evolve a theory to prove specific correctness properties for a certain class of specifications leaving out methodological aspects [12, 22]. *Low level testing approaches* which deal with database implementations instead of database specifications [14, 17, 18]. They all agree on the underlying idea to populate the database specifications with (reasonable) test data. By this, they try to show the database specification to be consistent which in general is only a semi-decidable property [15].

In contrast to the above approaches, we propose a **validation approach** which

- adapts validation techniques to **high level, graphically oriented specifications** of databases. For this purpose, we choose one representative from the group of semantic data models [10] to specify the database component, in our case: an extended ER model [7]. Such data models are well-known as high level specification languages for database components during the early development phases. By using a semantic data model, we support early validation, testing, and prototyping activities.

- bases on a **proof** and **formally described correctness properties** to determine the achievable levels of correctness for a database specification. One benefit of this was that we could prove that consistency is a decidable property for EER specifications [24].

- offers a **concrete, pragmatic validation algorithm** to actually generate sound test data and to support prototyping of database specifications. The algorithm basically uses the EER specification as input to construct test data. To ease prototyping, some additional information is required for the data to support the generation of comprehensible test data.

Nowadays, prototyping in general is realized by applying standard transformations of semantic data models to high level database specifications (e.g., relational) [6, 8, 9]. Thereby, database prototypes can be automatically generated, executing the (high level) database specification [8]. This eases prototyping of software specifications, too, because necessary test data can easily be stored in a database prototype and re-used later in the context of the whole system.

To present our approach, the paper is organized as follows: section 2 describes how the validation objects, i.e., high level database specifications, are expressed in terms of an EER model. Section 3 formally defines the correctness properties which are relevant for such database specifications. Section 4 gives the structural frame for the validation method, i.e., for how to investigate the defined correctness properties. It demonstrates the power and the theoretical basis of our approach. In section 5, the concrete validation algorithm is introduced. That section shows how the theory is implemented by an applicable and pragmatic validation method ensuring the consistency of database specifications and delivering concrete test data. The paper concludes by an outlook of how to apply this approach to other specification types.

2 Validation Objects: High Level Database Specifications

As mentioned in section 1, a database schema represents the specification for a central component of the entire application software. To describe such schemata, semantic data models are widely used [10]. Such models usually offer a graphical language which provides a comprehensible but still formal enough description of the problem domain during early development phases. To concretely develop our approach, we choose one representative of such models: the **extended ER model** [7]. We will give a brief overview over this specification technique in the following.

A conceptual database schema: The EER schema

The EER model is an extension of Chen's classical ER model [4]. Due to the additional concepts and the intuitive graphical specification language, EER specifications especially

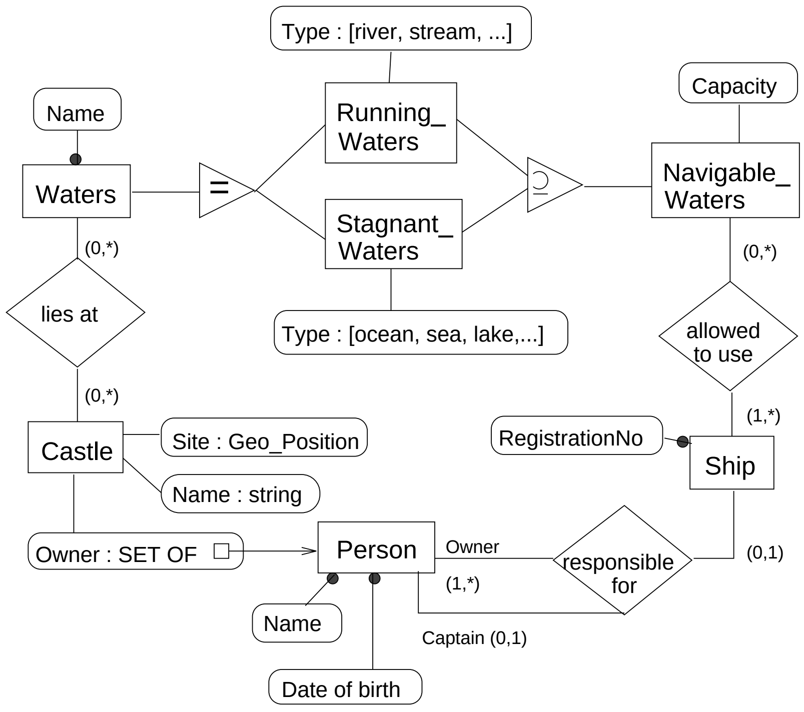


Figure 1: Example EER diagram

support the development of non-standard database applications [8] (e.g., geoscientific databases). Figure 1 depicts a small example of an EER diagram which we use to introduce the basic concepts. Compared to Chen's original ER model the main extensions include:

multi-, complex-, and entity-valued attributes: List, set, bag and record constructors are used to specify ranges of attributes. In this way, complex data types are built on top of already defined or existing data types like `int, real, string`. Entity-valued attributes use an entity type as range and may as well be multi- or complex-valued. With this extension, the classical concepts of "association" and "aggregation" of entities are modeled [21].

In figure 1, the attributes of the entity type `Castle` illustrate different kinds of attribute ranges[*]. `Owner` is an entity and set-valued attribute ranging over `Person` entities which models that a group of persons may together own a castle. `Site` uses the complex data type `Geo_Position` which is defined as abstract data type and could be a record of longitude, latitude, and height over sea level. `Name` is an ordinary attribute ranging over the (pre)defined data type `string`.

generalization and specialization of entities: For this purpose, the EER model provides a "type constructor" (noted as triangle in figure 1). With a type constructor, entities of "input types" can be distributed over new entity types, the "output types". For *generalizations*, we have many input types, i.e., the specific types, and one output type, i.e., the general type. For *specializations*, we have one input type, i.e., the general type, and many output types, i.e., the specific types.

A type constructor can be *partial* or *total* ("$\supseteq$" or "$=$"). In case of a partial type constructor, not every entity of an input type needs to appear in one of the output types. For total type constructors hold that every entity of an input type has to appear in (exactly) one of the output types.

Additional attributes can be defined for the output types, which in principle inherit the attributes of the input types.

Figure 1 gives an example for the use of type constructors. Input type `Waters` is (totally) specialized to `Running_Waters` and `Stagnant_Waters`, adding a new attribute `Type` to both output types. These specialized entity types become input types for a (partial) generalization themselves to create the new output type `Navigable_Waters` to which the attribute `Capacity` is added.

role concept for relationship types: Through differently labeled roles, entity types may participate multiple times at the same relationship (type) (cf. fig. 1 and the example below). By using roles, we distinguish different entities of the same entity type participating at one relationship.

cardinality intervals for relationship types: Cardinality intervals specify how many times an entity of an entity type may participate at relationships of a certain relationship type.

In figure 1, an example for the use of cardinalities and roles can be found for the relationship type `responsible for`. An entity of type `Person` (i.e., a person) can be responsible for a ship through two different roles: as `owner` and/or as `captain`. The cardinality intervals determine that a person owns at least one ship and may

[*]Due to space limitations, we only show only a few data types in the diagram.

own as much ships as he/she can afford. But a person is only allowed to work on not more than one ship as a captain.

A detailed discussion of the concepts and semantics of the EER schema is provided in [7].

3 Validation Goals: Correctness Criteria

The quality that can be achieved for a specification is one of the most important questions posed during the development of an application. Often, designers use an intuitive, experience-based notion of a specification's correctness to decide to what level their work is correct. Nevertheless, a more **formal classification of "correctness"** is necessary to capture the levels of quality in the development process. For programs, the classical notions of syntactical, partial, and total correctness hold. But for database specifications, the latter two are useless because they measure correctness of dynamic structures. Consequently, we need new, more suitable correctness criteria for database specifications. For this purpose, we introduce three levels of correctness for database specifications:

Definition 1: Syntactical correctness
A database schema is syntactically correct, if it doesn't violate syntactical and context-sensitive rules of the specification language used.

This level of correctness holds for both static and dynamic structures and can be easily checked by a parser.

Although satisfying syntactical correctness, a schema can, of course, violate semantical constraints. The first step to approach semantical correctness for database specifications is to detect contradictions in the schema, i.e., to check whether the schema is consistent. For this purpose, possible interpretations ("populations") of that schema have to be analyzed. Following logics, an interpretation of a specification is a collection of possible instances for every element of the specification (in case of EER diagrams: entities for every entity type, relationships for every relationship type, etc.). The size of this collection, i.e., how many instances each element may have, is determined by the structure of the schema. Specification languages contain rules which constrain the population of a schema. For example, cardinality intervals restrict the amount of possible instances of relationship types. We call these rules "**quantitative dependencies**". They numerically constrain the possible interpretations of a schema. To check satisfiability for quantitative dependencies, only one of all possible interpretations has to be constructed, thereby proving that instantiation of the specification is - in principle - possible.

Definition 2: Consistency
A database schema is consistent, if at least one interpretation exists that contains finite, non-empty sets of allowed instances for each element of the database schema.

In logics, consistency is a well-known correctness criteria for sets of formulas. As a database specification can be considered as such a collection, the criterion is equally applicable. In logics, empty or infinite interpretations are valid, too. However, they make no sense for data to be stored in a database. Hence, we adapted the logics' notion of consistency to meet database-specific correctness requirements.

That syntactical correctness does not necessarily imply consistency, can be seen in figure 2.

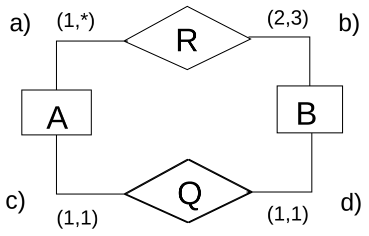


Figure 2: Abstract example of an inconsistent EER schema

Edge `c` and `d` imply that the number of entities of type `A` equals the number of entities of type `B`. That means, for each `b` exactly one `a` of type `A` is needed to establish the relationships q of type `Q` between `A` and `B`. On the other hand, relationship type `R` (edges `a` and `b`) forces the number of instances `a` of `A` to be at least twice as high as the number of instances `b` of entity type `B`, because each `b` consumes at least two `a`'s and at most three `a`'s. This is a contradiction to the association of `a`'s and `b`'s via `Q`. Thus, all conditions together lead to an inconsistent, unrealizable (although syntactically correct) specification.

Semantical correctness forms the highest level of correctness. It demands a database schema to reflect the desired portion of the problem domain. This level of correctness is, of course, a non-decidable quality [15] which has to be evaluated individually. This holds especially if the constraints of the problem domain have only been informally specified (as often the case).

Definition 3: Semantical correctness
A database schema is semantically correct in regard of the problem domain, if the database schema is consistent and its (possible) interpretations are equivalent to the (possible) interpretations of the problem domain.

To check for semantical correctness involves generating a lot of interpretations and "compare" them with the interpretations developers have in mind. This is often done in the scope of prototyping activities.

By translating our abstract example in figure 2 into an intuitive example (cf. fig. 3) we illustrate what simple kind of errors may cause inconsistent specifications. The EER specification (cf. fig. 3) is intended to model that

Students have to participate in at least two and at most three courses. A course (of our small world) has at least one participant but in general allows for an unrestricted number of participants. Students have to and are allowed to pass the exam for a certain course exactly for once.

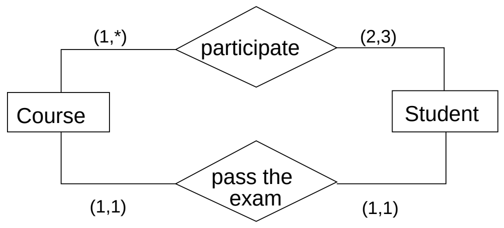


Figure 3: Concrete example of the inconsistent EER schema in figure 2

According to figure 2 and the calculations performed, this diagram is inconsistent. Such kind of inconsistency is often caused by erroneously used cardinality constraints, a well-known problem of using such constraints. In contrast to the above formulated intention, the relationship type `pass the exam` and its associated cardinalities model that a student passes the exam exactly for *one single* course in his/her life time. Correspondingly, a course is modeled to offer its exam only for *one single* student. The common pattern behind this mistake is that cardinality constraints are misused to couple *individual entities* via a concrete relationship. In contrast to that, cardinality constraints can only express *at how many* actual relationships an individual entity may participate without determining the other participating entities.

If such mis-interpretations directly result in inconsistent specifications, it has to be detected during the prototyping activities. For example, if we would have defined the cardinality interval $(0,*)$ for the course (edge `c`), the specification would have passed the consistency check. But validating the specification for semantical correctness would have revealed that only a small set of degenerated populations could have been generated - in this case exactly those which allow each student to pass only one exam during his/her life time. How to perform such a validation for high level database specifications will be shown in the next sections.

4 Validation Method: Principles and Theory

Experiences in the field of validation and testing show that a lot of testing effort is done in an unstructured, ad hoc manner. Although, such proceeding may suit for small specifications, it is inadequate for complex and large specifications. Thus, the validation effort itself has to be structured and defined in advance (cf. fig. 4). Afterwards, such a structural frame can be tailored to the actually desired validation process.

Structural frame

To start a validation effort, we have to identify the validation goals we want to achieve, i.e., choose the (special purpose) correctness criteria (cf. section 3). These criteria imply a set of **hypotheses** which reflect presumptions about the correctness of the considered specification. The hypotheses remain constant during the validation procedure and characterize the results which we expect at the end of the procedure. The next step is to

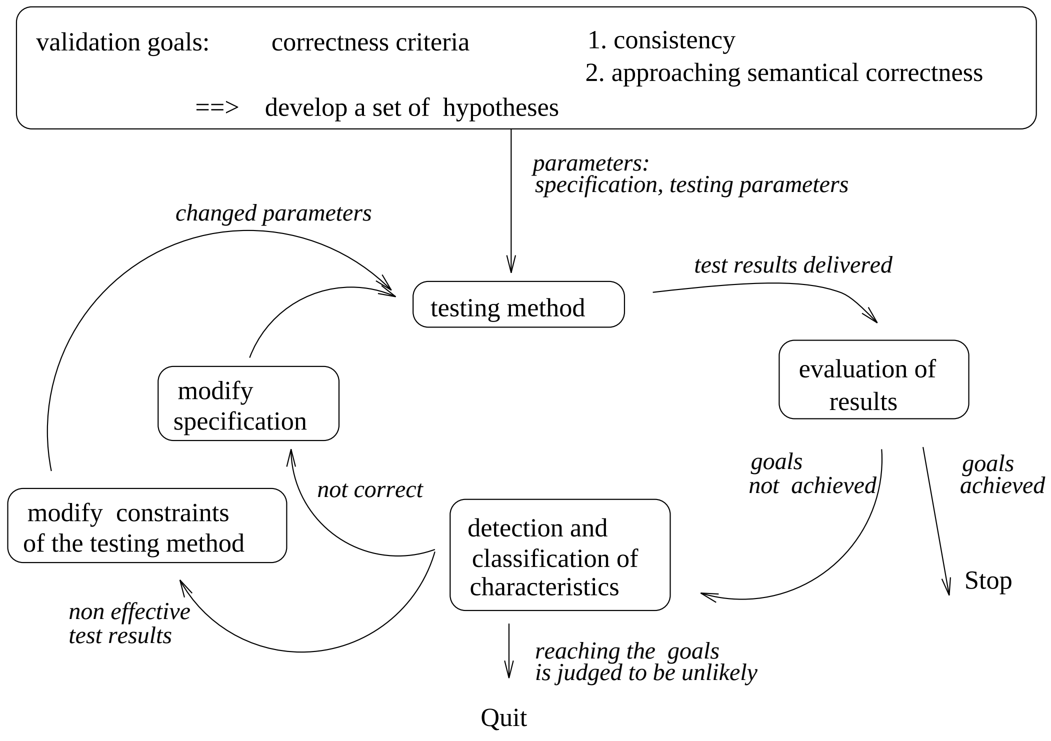

validation goals:
correctness criteria
1. consistency
2. approaching semantical correctness
==> develop a set of hypotheses
parameters: specification, testing parameters
changed parameters
testing method
test results delivered
evaluation of results
modify specification
goals not achieved
goals achieved
not correct
modify constraints of the testing method
detection and classification of characteristics
Stop
non effective test results
reaching the goals is judged to be unlikely
Quit

Figure 4: Structural frame for validation effort

execute a **testing method**. This method bases on collections of generated test data and should be planned to support or to reject the chosen hypotheses. After the execution of the testing method, its results are compared with the expected results. Through such an **evaluation**, we decide whether we achieved our validation goals and may terminate the validation procedure or have to continue because the goals have not been achieved. In the latter case, we detect and classify the **characteristics** of the failure from the delivered results. This classification leads to three different conclusions:

- The specification is not correct wrt. our set of hypotheses. Thus, we have to **modify the specification**.
- The delivered results are insufficient to evaluate our set of hypotheses, e.g., a hypothesis can neither supported nor rejected. Thus, the **parameters of the testing method** have to be modified.
- Reaching the goals, i.e., satisfying the hypotheses, is judged to be unlikely. In this case, the validation procedure should be aborted.

After modifying the incorrect specification or the unsuitable parameters, we reiterate the validation procedure with the same set of hypotheses until it either stops (goals achieved) or is aborted (human judgement).

To organize our validation approach concerning the quality of EER-diagrams, we have to instantiate this structural frame *twice*. Since our approach aims at two correctness levels: consistency and approaching semantical correctness via prototyping, we need one validation process for each criterion. Furthermore, we order both processes in a sequence. Validation for consistency is executed first followed by validation for semantical correctness because the latter requires consistent specifications. The principles and theoretical aspects of both these processes are discussed in the following.

Instantiating the frame: consistency checking

Instantiating the structural frame for consistency checking requires the following parameter's values:

validation goal:	check consistency for the specification
hypotheses:	a population exists which satisfies the specification
specification:	(a syntactically correct) EER diagram
additional testing parameter:	not needed
testing method:	(automatically) generate an arbitrary population for the specification

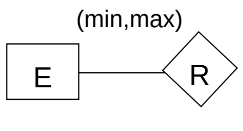
The **validation process for consistency** parametrized like above assumes the specification to be consistent. The according hypothesis has to be shown by generating a test population for the EER diagram. Since, it suffices to find one arbitrary population no additional testing parameters are required to characterize the data.

For our validation approach, we involve **a proof** which allows to decide whether an EER diagram is consistent or not. Originally, Lenzerini/Nobili [12] developed such a proof to decide consistency for classical ER specifications [4]. Basically, quantitative dependencies of an ER diagram, i.e., cardinality constraints, are mapped into a system of linear inequations. Then, the proof uses the so-called "Ellipsoid-Algorithm" of Papadimitriou/Steiglitz [20] to decide definitely about the existence of a general solution for the system of linear inequations. The general solution (if one exists) can be analyzed with the algorithm "fractional dual" [20] to determine definitely the existence of an integer solution. The integer solution of the system of linear inequations describes the size of the population, i.e., the number of instances for each modeling item (entity type, relationship type, etc.) used.

We extended this proof to be applicable to **systems of inequations reflecting EER diagrams** [23, 24]. Such specifications contain additional quantitative dependencies like those for type constructors and entity-valued key attributes. To capture an EER specification in terms of a system of inequations, we developed the following set of transformation rules:

Definition 4: System of inequations for EER diagrams

1. cardinality constraints



(a) for each cardinality constraint (`min`,`max`) with `min` > 0

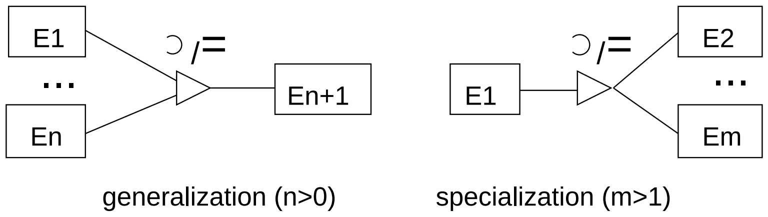
$$|R|^{\dagger} \geq \texttt{min} \cdot |E|$$

(b) for each cardinality constraint (`min`,`max`) with `max` $< \infty$

$$|R| \leq \texttt{max} \cdot |E|$$

The inequations describe the ratios between the number of instances of an entity type and a relationship type.

2. type constructions



(a) for each type construction

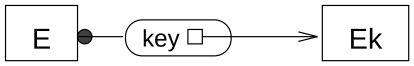
$$\Sigma_{i=1}^{n} \; |E_i| \geq \; \Sigma_{j=n+1}^{m} \; |E_j|$$

(b) additionally, for each **total** type construction

$$\Sigma_{i=1}^{n} \; |E_i| \leq \; \Sigma_{j=n+1}^{m} \; |E_j|$$

In general, this means that the number of instances of the output types may not exceed the number of instances of the input types. In case of total type construction, they are equal.

3. for each entity valued key attribute



$$|E_k| \geq \; \varepsilon, \qquad \varepsilon = |E|$$

E_k must provide as many instances that each entity of E can be uniquely identified. If E possesses more than one key attribute, ε may decrease because a combination of values is possible (for how to calculate ε, cf. [23]).

4. for each E, R contained in the EER specification

$$|E|, |R| \geq 1 \text{ and } |E|, |R| < \; \infty$$

to guarantee finite and non-empty populations.

In [24], we show that it is possible to transform arbitrary EER diagrams into such a system of inequations. Hence, the **consistency of EER schemata becomes a decidable quality**.

In the following, we illustrate our definition by a simple example. For the EER diagram given in figure 5, the transformation yields the following system of inequations:

$^{\dagger}|R|$ is a variable denoting the number of instances for R, analogously for E, etc.

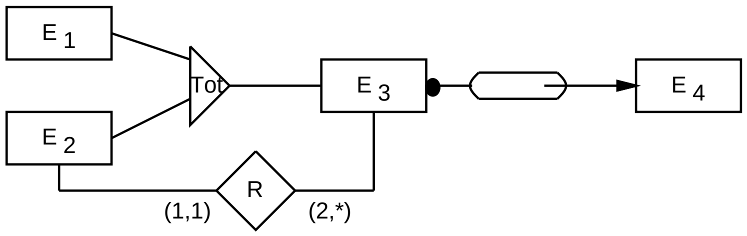


Figure 5: Example of quantitative dependencies in an EER schema

a)	$\lvert R\rvert \geq 2\cdot \lvert E_3\rvert$	cardinality constraint
b) – c)	$\lvert E_2\rvert = \lvert R\rvert$	cardinality constraint
	$(\Leftrightarrow \lvert E_2\rvert \leq \lvert R\rvert \quad \wedge \quad \lvert E_2\rvert \geq \lvert R\rvert)$	
d) – e)	$\lvert E_1\rvert + \lvert E_2\rvert = \lvert E_3\rvert$	type construction
	$(\Leftrightarrow \lvert E_1\rvert + \lvert E_2\rvert \leq \lvert E_3\rvert \quad \wedge$	
	$\lvert E_1\rvert + \lvert E_2\rvert \geq \lvert E_3\rvert)$	
f)	$\lvert E_4\rvert \geq \lvert E_3\rvert$	key attribute
g) – k)	$\lvert E_1\rvert \geq 1, \ldots, \lvert E_4\rvert \geq 1,\ \lvert R\rvert \geq 1$	no empty sets of instances
l) – p)	$\lvert E_1\rvert < \infty, \ldots, \lvert E_4\rvert < \infty,\ \lvert R\rvert < \infty$	only finite sets of instances

The solution of this system of inequations describes the size of the population of the schema. In our example, no solution exists because the inequations a) and b)–c) imply $\lvert E_2\rvert \geq 2\cdot \lvert E_3\rvert$ which means a contradiction to d)–e). Thus, the schema depicted in figure 5 is not consistent. The inconsistency is obviously caused by the "dependency cycle" [E_3, type constructor, E_2, R, E_3] in which every element quantitatively depends on itself through the cyclic path.

In general, for **non-cyclic parts of a schema**, it is always possible to find an (integer) solution for the number of instances. This holds because in a non-cyclic path no contradictions in the ratios of participating schema elements can occur. In terms of the corresponding system of inequations, this means that more variables than inequations exist [20]. Special care must only be taken for **cyclic dependencies** which cause the system of inequations to be (row) degenerated, i.e., more inequations than variables exist (cf. fig. 5: 5 variables, 16 inequations).

In principle, determining the existence of a (nonempty, finite) interpretation for an individual EER diagram corresponds to solving the related system of inequations. From a general solution, an integer solution can be calculated which describes the size of possible populations. The ability to decide the consistency of an EER specification provides our corresponding validation process with a formal criterion when to terminate - either a solution exists or not. With this certainty of consistency, the next level of correctness, the semantical correctness (cf. definition 3), can be tackled.

Instantiating the frame: approaching semantical correctness

Instantiating the structural frame to check for semantical correctness needs the following parameter's values:

validation goal:	approaching semantical correctness for the specification
hypotheses:	1. the specified test populations satisfy the specification
	2. the specification meets the user's intention
specification:	(a consistent) EER diagram
testing parameter:	1. desired quantities of instances for (part of) the specification, i.e., the size of the population
	2. collections of "real-world" (i.e., readable, pronounceable) basis-data
testing method:	1. (automatically) generate populations for the specification, according to the testing parameters
	2. develop queries to prototype the population

In fact, this set of parameters equals those necessary for **prototyping** activities. This is not surprising because approaching semantical correctness heavily involves user interaction. Concerning the hypotheses, we assume the specification to have passed the syntactical and consistency check and to reflect the desired portion of the problem domain and that our intended, i.e., specified, populations will satisfy it.

To support prototyping activities, the **population** should fulfill certain properties. Our approach offers to control the generation of populations in two ways (testing parameters). We can *restrict its size* and require the concrete data to be better *comprehensible*, i.e., pronounceable by a user. The latter is adopted from a former low level testing approach [17] and is part of the concrete algorithm [13].
Sizing the population means to modify the system of inequations for the EER specification. As stated in definition 4.4, the number of instances for each entity and relationship type has to be greater than 0 and less than ∞. Consequently, sizing the number of instances means to replace these inequations by such which give the desired number. Obviously, this modified system of inequations requires to be checked again for consistency. Thus, introducing testing parameter 1. might imply a second validation process for consistency (if the quantities haven't been calculated automatically).

Applying the testing method means to generate concrete test populations (cf. section 5) and to develop a collection of queries. These queries should reflect the expected workload of the system and should test for semantical meaning as well as boundary cases or coverage criteria. We are still investigating whether part of such queries can be automatically derived from an EER specification. With the generated populations and the collection of queries at hand, the validation process follows the structural frame.

5 Validation Method: Pragmatic Algorithm

In the last section, we introduced a proof-based method to check consistency and semantical correctness via prototyping. This method is only of theoretical benefit due to its enormous complexity which is measured in terms of the complexity of a standard linear optimization problem [20, 23]. Additionally, by solving the system of inequations, the information is lost where errors occur in the diagram. But this is necessary information to correct inconsistencies.

Hence, we need a **pragmatic validation method** which is applicable within reasonable time restrictions and which effectively helps to localize errors. To satisfy the latter, references to the structure of the specification have to be managed during the execution of the testing method. To reduce the complexity, the method has to be specially tailored to the needs of the underlying specification, thereby, avoiding the inherent complexity of general approaches. For this purpose, we developed and implemented [23] a concrete validation algorithm for EER diagrams. This algorithm bases on the transformation of an EER diagram into a so-called **cardinality graph** (cf. fig. 6). A cardinality graph describes an EER diagram in terms of its quantitative dependencies (cf. sec. 3) and additionally, keeps a relation between them and the structure of the EER diagram. Thus, our algorithm firstly constructs a cardinality graph from the EER diagram. We give the corresponding construction rules in definition 5.

Definition 5: Cardinality graph $\mathrm{cg}_{\mathrm{eer}} = \langle\ \mathcal{E},\ \mathcal{R},\ \mathcal{K}_1,\ \mathcal{K}_2\ \rangle$ for an EER diagram `eer`.

1. $\mathcal{E}$: a set of nodes.
   Each node represents an entity type of the EER diagram `eer`,
2. $\mathcal{R}$: a set of nodes.
   Each node represents a relationship type of the EER diagram `eer`,
3. $\mathcal{K}_1$: a set of non-directed, labeled edges.
   Each edge represents a cardinality constraint (`min`,`max`) of the EER diagram `eer` with (`min`,`max`) $\neq$ (0,$*$). An edge connects the two nodes in cg_{eer} representing the entity and relationship type for which the cardinality constraint is defined. The edge is labeled with this cardinality constraint interval.
4. $\mathcal{K}_2$: a set of directed, labeled edges.
   Each edge represents a cardinality constraint induced by a type construction or a entity valued key attribute (cf. definition 4.2 and 4.3). It is labeled twice:
   - For total type constructions with "=", for partial ones with "$\supseteq$", and for entity valued key attributes always with "$\supseteq$".
   - With an identifying number to distinguish different constructs.

   In case of entity valued key attributes, the edge points to the node representing the entity type to be identified and connects it with the node representing the key attribute's entity type. In case of generalization, the edge points from the nodes representing the input entity types to the node representing the output entity type. In case of specialization: vice versa.

To illustrate this definition, we transform an EER diagram into its corresponding cardinality graph (cf. fig. 6). Please note that connections between entity types and relationship types marked in the EER diagram with cardinality constraints (0,$*$) don't appear in the graph, e.g., see E_7, R_1, since they don't restrict the possible populations (cf. definition 5.3). Furthermore, multiple relationships like those between E_2 and R_1 are aggregated into one restricting edge labeled by the strongest constraint determined from the multiple relationships.

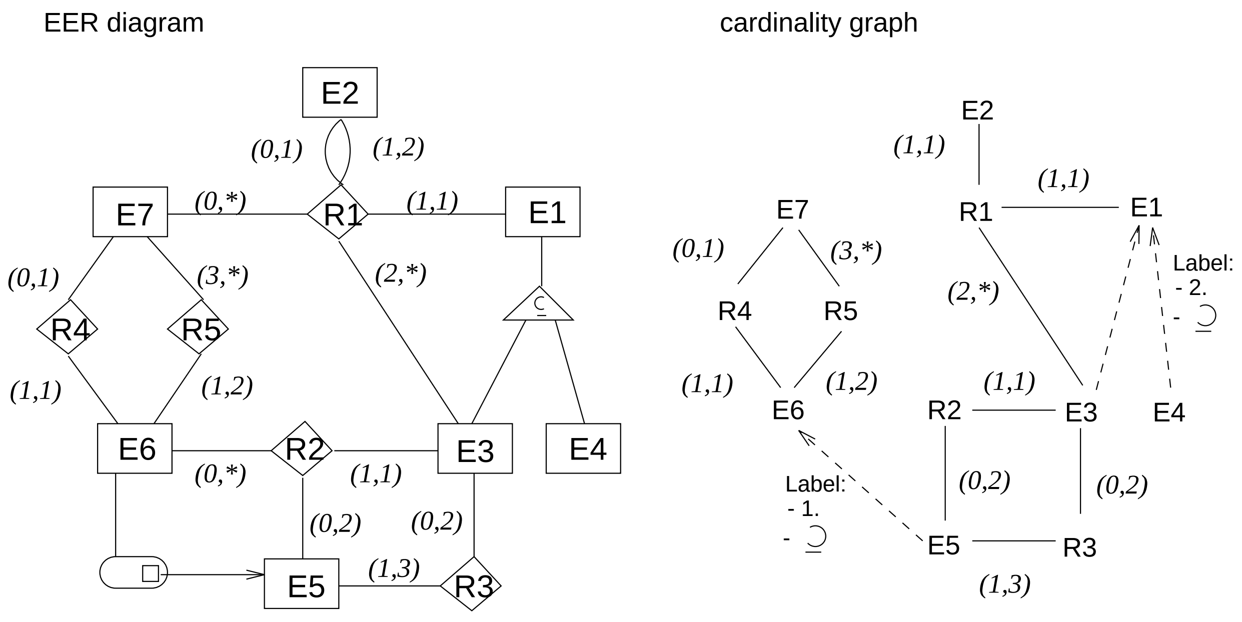


Figure 6: Example how to transform an EER diagram into a cardinality graph

On top of such a cardinality graph, it is now possible to detect **critical paths**. A critical path in the context of consistency checking is a cyclic path in the cardinality graph where entity types quantitatively depend on themselves. Such reflexive dependencies are the only quantitative dependencies which can (but must not) cause inconsistencies. Hence, detecting and checking critical paths forms the second step of our algorithm.

We distinguish two types of critical paths in the cardinality graph (cf. fig. 6): **pure entity/relationship cycles** like E_6, R_5, E_7, R_4, E_6 or **cycles containing type constructions** like E_3, E_1, R_1, E_3. Each such path and their combination represent a possible consistency conflict. Hence, each individual critical path and overlapping critical paths have to be checked. Non cyclic dependencies don't have to be checked since they don't imply consistency conflicts. As stated in section 4, it is always possible to generate instances for the related modeling concepts.

Checking pure entity/relationship cycles

To check pure entity/relationship cycles, we compute a so-called **cardinality condition** for each such cycle. For this purpose, we choose a direction how to circulate through a cycle. Due to the chosen direction and starting from one node, we calculate the dependency of the node from itself in terms of cardinality constraints. To achieve this, we have

to regard that depending on the chosen direction, an edge of the graph may describe a dependency from an entity to a relationship or vice versa.
This is reflected by the following, **direction-dependent quantitative dependencies**:

- The direction-dependent quantitative dependency of entity type E represented by node n_e to relationship type R represented by node n_r amounts to
  $\mathtt{min}\cdot |E| \leq |R| \leq \mathtt{max}\cdot |E|$
  with ($\mathtt{min}$,$\mathtt{max}$) the label of the edge connecting n_e with n_r.
- The direction dependent quantitative dependency of relationship type R represented by node n_r to entity type E represented by node n_e amounts to
  $\frac{1}{\mathtt{max}}\cdot |R| \leq |E| \leq \frac{1}{\mathtt{min}}\cdot |R|$
  with ($\mathtt{min}$,$\mathtt{max}$) the label of the edge connecting n_e with n_r and
  $\frac{1}{\mathtt{min}} = \infty$ for $\mathtt{min} = 0$ and $\frac{1}{\mathtt{max}} = 0$ for $\mathtt{max} = \infty$

Now, to calculate the dependency for a specification element `X` (represented by node $\mathtt{n}_x$) from itself, we use the above direction dependent quantitative dependencies. For a cycle in the cardinality graph with `p` nodes, we get `p` such dependencies with `p` different variables (one for each node). Now, we can eliminate `p-1` (node's) variables via substitution and, hence, express the reflexive quantitative dependency for the remaining node $\mathtt{n}_X$ as follows:

$$a_1 \cdot a_2 \cdot \ldots a_p \cdot |\mathtt{X}| \leq |\mathtt{X}| \leq b_1 \cdot b_2 \cdot \ldots b_p \cdot |\mathtt{X}|$$

Dividing this inequation by $|\mathtt{X}|$[‡] yields the **cardinality condition** for the considered cycle which has to be satisfied in case of consistency:

$$\prod_{i=1}^{p} a_i \leq 1 \leq \prod_{j=1}^{p} b_j$$

Such a cardinality condition is easy to compute and can be calculated independently from the starting node. Thus, it is possible to decide for each individual entity/relationship cycle whether it is consistent or not. Fortunately, it can be shown that even overlapping cycles (as those in figure 7) can be proven to be consistent by checking each individual cycle contained.

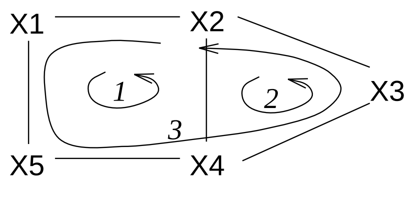


Figure 7: Overlapping cycles in cardinality graph

This holds because if each individual cardinality condition is satisfied (for example: for cycles *1, 2,* and *3* of fig. 7) then a strongest cardinality condition can be derived from

‡ $|\mathtt{X}| > 0 is obvious as we are looking for non-empty sets of instance.$

the set of individual ones which gives the smallest interval $[\prod_{i=1}^{p} a_i, \prod_{j=1}^{p} b_j]$ valid for all overlapping cycles.

Using our example graph in figure 5 for illustration purposes, we calculate the following cardinality condition for the two pure entity/relationship cycles:

$$\begin{array}{llllll}
1) & E_5, R_3, E_3, R_2, E_5: & 1 \cdot \frac{1}{2} \cdot 1 \cdot \frac{1}{2} \cdot \; |E_5| & \leq & |E_5| & \leq 3 \cdot \infty \cdot 1 \cdot \infty \cdot |E_5| \\
 & & \frac{1}{4} & \leq & 1 & \leq \; \infty \\
2) & E_6, R_5, E_7, R_4, E_6: & 1 \cdot 0 \cdot 0 \cdot 1 \cdot \; |E_6| & \leq & |E_6| & \leq 2 \cdot \frac{1}{3} \cdot 1 \cdot 1 \cdot |E_6| \\
 & & 0 & \leq & 1 & \leq \; \frac{2}{3}
\end{array}$$

The cardinality condition of the first cycle is satisfiable and, therefore, the corresponding specification part consistent whereas the second cardinality condition indicates inconsistency. Thus, the first part of our second algorithm step consists of detecting all pure entity/relationship cycles in the cardinality graph of the EER specification and to calculate and check the cardinality condition for each such cycle. Consequently, the rest of the second algorithm step deals with the remaining critical paths:

Checking cycles containing type constructions

To check **cycles containing type constructions**, we have to ensure that the quantitative dependencies between entities of input and output types (cf. definition 4.2) don't conflict with dependencies induced by cardinality constraints (cf. definition 4.1). Thus, we identify all paths in the cardinality graph from a node n_{out} representing an output type E_{out} to a node n_{in} representing an input type E_{in} of a certain type construction, e.g., E_1, R_1, E_3 in figure 6. For each such path, we calculate the dependency from the output type to the input type with the help of the direction-dependent quantitative dependencies introduced above. This results in a **cardinality condition for each path** ϕ with k the number of connected nodes:

$$\prod_{i=1}^{k} a_i \; | E_{out} | \; \leq \; | E_{in} | \; \leq \; \prod_{j=1}^{k} b_j \; | E_{out} |$$

Since, several paths may exist in the graph, leading from n_{out} to n_{in}, several cardinality conditions may exist. In order to guarantee consistency, we have to ensure the strongest quantitative restriction jointly induced by all paths ϕ. This means to find the **overlapping interval** for all cardinality conditions of the different paths:

$$\mathrm{MAX}_{\phi} \; (\prod_{i=1}^{k} a_i) \; | E_{out} | \; \leq \; | E_{in} | \; \leq \mathrm{MIN}_{\phi} \; (\prod_{j=1}^{k} b_j) \; | E_{out} |$$

$$\text{or for short: } A \cdot | E_{out} | \; \leq \; | E_{in} | \; \leq \; B \cdot | E_{out} |$$

Since, we may have sets of input types for one output type in case of generalization[§], the total number of all necessary output entities has to be calculated in order to derive the

[§]vice versa for specialization.

number of the necessary input entities (cf. definition 4.2). This results in the following expression:

$$(\Sigma_{E \in inputtypes} \mathrm{A}_E) \cdot \mid \mathrm{E}_{out} \mid \;\leq\; \mid \mathrm{E}_{in}^{+} \mid \;\leq\; (\Sigma_{E \in inputtypes} \mathrm{B}_E) \cdot \mid \mathrm{E}_{out} \mid$$

The expression above yields a relation between $(\Sigma_{E \in inputtypes}\ \mathrm{A}_E)$ and $(\Sigma_{E \in inputtypes}\ \mathrm{B}_E)$ which describes the quantitative dependencies between input and output types independent from the dependency induced by the type construction itself. In case of consistency, the relation must be numerically satisfiable and may not conflict with the quantitative dependency induced by the type construction. The interval above can easily be computed for each type construction and analyzed wrt. its inherent type construction dependencies[¶].

To illustrate this proceeding with an example, consider the type construction in figure 6. The input types E_3 and E_4 depend on the output type E_1 via the type construction. Additionally, E_3 depends on E_1 via the path E_3, R_1, E_1. For E_4, no additional dependency exists. The twofold dependency for E_3 has to be checked for consistency conflicts.
First, we calculate the cardinality condition for the detected path:

$$\mathrm{E}_1, \mathrm{R}_1, \mathrm{E}_3 : \; \frac{1}{2} \cdot 1 \cdot \mid \mathrm{E}_1 \mid \;\leq\; \mid \mathrm{E}_3 \mid \;\leq\; 1 \cdot \infty \cdot \mid \mathrm{E}_1 \mid$$

Since no other path has to be considered, we get $\mathrm{A}_{E_3} = \frac{1}{2}$ and $\mathrm{B}_{E_3} = \infty$. For E_4, the cardinality conditions are set to $\mathrm{A}_{E_4} = 0$ and $\mathrm{B}_{E_4} = \infty$ which means no restriction (default). This results into:

$$(\mathrm{A}_{E_3} + \mathrm{A}_{E_4}) \cdot \mid \mathrm{E}_1 \mid \;\leq\; \mid \mathrm{E}_{3,4}^{+} \mid \;\leq\; (\mathrm{B}_{E_3} + \mathrm{B}_{E_4}) \cdot \mid \mathrm{E}_1 \mid$$

$$\frac{1}{2} \cdot \mid \mathrm{E}_1 \mid \;\leq\; \mid \mathrm{E}_{3,4}^{+} \mid \;\leq\; \infty \cdot \mid \mathrm{E}_1 \mid$$

This expression gives the quantitative dependencies between the input and output elements of a type construction independent from its inherent dependency. This inherent dependency resulting from the type construction itself taken into account, implies (cf. definition 4.2):

$$\mid \mathrm{E}_3 \mid + \mid \mathrm{E}_4 \mid \;\geq\; \mid \mathrm{E}_1 \mid \;\Rightarrow\; \mid \mathrm{E}_{3,4}^{+} \mid \;\geq\; \mid \mathrm{E}_1 \mid$$

which doesn't conflict with the above expression.

Deciding consistency for such critical paths reduces to analyzing the relation of the A's and B's for a number of well-defined error cases [23]. With this check, the second step of our algorithm terminates. As result, consistency or inconsistency is delivered. In case of consistency, test data generation follows.

Test data generation

Now after the first two algorithm steps, consistency is determined. The calculated intervals for the critical paths determine the ratios between the number of instances for

[¶]Due to the two kinds of type constructions, we distinguish three fine grained error cases which we will not present here. They are discussed in detail in [23].

specification elements, i.e., entity and relationship types. We employ this information to actually generate test data, respectively, a test population.

In this step of the algorithm (step 3), an interactive process starts which expects proposals for instances' numbers from a user. Due to the (pre)calculated ratios, the proposed numbers are checked for satisfiability and either accepted or rejected. In case of rejection, proposals for *consistent quantities are offered by the algorithm* that can be accepted by the user. If no initializing numbers are given, a population of arbitrary size is generated.

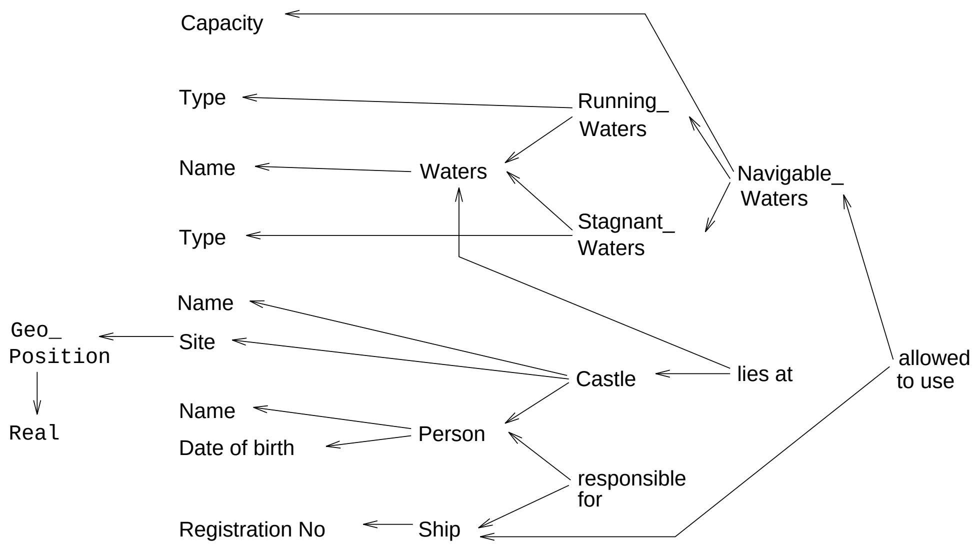


Figure 8: Example dependency graph for the EER diagram in fig. 1

In step 4 of the algorithm, the concrete test data generation starts. For this purpose, we transform the original EER diagram into a so-called **general dependency graph**. We briefly introduce such a graph by example (cf. [24] for a broader overview). Figure 8 shows such a general dependency graph for our example EER diagram in figure 1. Each node represents an element of the EER diagram, e.g., an entity or relationship type, an attribute, or a data type. Each edge expresses an **existential dependency** of the node the edge origins in from the node it points to. For example, an instance of a relationship type can only exist if the participating entities exist. In turn, each entity needs the values of its attributes, an entity of an output class requires the existence of a corresponding entity in the input class, and an attribute needs values of its range. In figure 8, `lies at` depends on a `Castle` entity and on a `Waters` entity, the `Castle` entity itself depends on its attribute values `Site` and `Name`. `Site` depends on values for the data type `Geo_Position` which in turn depends on its component data types latitude, longitude, and height, each of them ranging over `Real` numbers.

We use such a general dependency graph to describe an **execution order for the test data generation**. Test data is generated for each node of the dependency graph for which two conditions hold: Firstly, no data yet exists for the node and, secondly, the node is independent from any other node without test data. Due to this order, a traversal through the dependency graph is defined. Special care must be taken for cyclic dependencies but since the original specification is known to be consistent such cycles can be handled via backtracking steps.

The basic philosophy for our test data generation comprehends also a **pool concept**. In contrast to other approaches with complex rejection mechanisms, the certainty of a consistent specification allows to employ an optimistic constructive approach. We successively generate value pools for each specification element represented in the general dependency graph, constructing them from already completed pools of validated (sub-)pools. According to the execution order, the algorithm starts to generate data pools for the data types, e.g., `int, real, string`. From these basic pools, pools for the complex data types, e.g., `Geo_Position`, are constructed. On top of these data pools, values for attributes are chosen and assembled to values for entities, etc. Using values pools supports to

- explicitly control and limit (to the necessary minimum) the number of actual instances for a specification element,
- define specific characteristics for such pools resp. its values, e.g., statistical distributions, pronounceability, or the use of "real-world" data,
- to reuse former pools, e.g., for names of persons

After the successful construction of an entire population, the data is stored in the database prototype referring to the EER specification. There, it can be queried or represented by a browser [8].

6 Conclusions

We presented a validation approach which focusses on the validation of central database components of application software. Its original basis comprises a formal proof and a formal classification of special purpose correctness levels to be achieved. This solid, theoretical foundation enables to develop a pragmatic validation algorithm which

- definitely determines whether a database specification is consistent or not and, thus, yields a terminating but still pragmatic method to prove this property,
- uses only the specification and user information about the size of the population and the desired characteristics of the value pools to generate a population, i.e., test data,
- supports reuse of existing value pools in order to reduce the validation effort,
- supports high level prototyping of the database component as well as of associated functional specifications, because the data can easily be stored and queried from a corresponding database prototype.

Currently, we are investigating how this validation approach can be adapted to other specification types, e.g., object oriented specifications. Due to the similarities of structuring concepts of semantic data models and object structures, e.g., generalization, object valued attributes, etc, the approach seems to be applicable for object oriented specifications, too [11]. Additionally, the company TNO at Delft (The Netherlands) checks whether this approach is applicable for the validation of their data models (using the specification language EXPRESS). Furthermore, we are investigating to what extend queries for standard test cases, e.g., boundary cases, can be derived from an existing specification.

References

[1] B. Beizer. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold Company, 1984.

[2] G. Bernot. Testing Against Formal Specifications: a Theoretical View. Technical Report URA 1327 du CNRS, Departement de Mathematique et d'Informatique, Ecole Normale Superieure, Paris, 1991.

[3] F. Bry and R. Manthey. Checking Consistency of Database Constraints: A Logical Basis. In *Proc. 12th Int. Conf. Very Large Data Bases*, pages 13–20, 1986.

[4] P.P. Chen. The Entity-Relationship Model - Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[5] L. Delcambre and K. Davis. Automatic Validation of Object-Oriented Database Structures. In *Proc. 5th Int. Conf. Data Engineering*, pages 2–9. IEEE Computer Society, 1989.

[6] O. DeTroyer. *On Data Schema Transformation.* PhD thesis, University of Tilburg, 1993.

[7] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich. Conceptual Modelling of Database Applications Using an Extended ER Model. *Data & Knowledge Engineering*, 9(2):157–204, 1992.

[8] G. Engels and P. Löhr-Richter. CADDY: A Highly Integrated Environment to Support Conceptual Database Design. In G. Forte, N. Madhavji, and H. Müller, editors, *Proc. 5th Int. Workshop on Computer-Aided Software Engineering, Montreal, Kanada*, pages 19–22. IEEE Computer Society Press, 1992.

[9] U. Hohenstein. Automatic Transformation of Entity-Relationship Schemas into Relational Schemas. Technical Report 88-10, Technical University of Braunschweig, 1988.

[10] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, Research Issues. *ACM Computing Surveys*, 19(3):201–260, 1989.

[11] B. Hulvershorn. Similarities between the Generation of Test Data for EER Schemata and TROLL Schemata (in German). Project Thesis, Technical University of Braunschweig, 1993.

[12] M. Lenzerini and P. Nobili. On the Satisfiability of Dependency Constraints in Entity-Relationship Schemata. *Information Systems*, 15(4):453–461, 1990.

[13] P. Löhr. Automatic Generation of Pronouncable Strings (in German). Project Thesis, Technical University of Braunschweig, 1984.

[14] N. Lyons. An Automatic Data Generating System for Data Base Simulation and Testing. *Database*, 8(4), 1977.

[15] Z. Manna. *Mathematical Theory of Computation.* Mc Graw - Hill, USA, 1974.

[16] A. Neufeld, G. Moerkotte, and D. Lockemann. Generating Consistent Test Data: Restricting the Search Space by a Generator Formula. *The VLDB Journal*, 2(2):173–213, 1993.

[17] L. Neugebauer and K. Neumann. Schema directed Generation of Test Data for Relational Databases (in German). Technical Report 85-02, Technical University of Braunschweig, 1985.

[18] H. Noble. The Automatic Generation of Test Data for a Relational Database. *Information Systems*, 8(2):79–86, 1983.

[19] L. Osterweil and L.A. Clark. A Proposed Testing and Analysis Research Initiative. *IEEE Software*, 9(5):89–96, 1992.

[20] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization - Algorithms and Complexity.* Prentice-Hall, USA, 1982.

[21] J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.

[22] B. Thalheim. Fundamentals of Cardinality Constraints. In G. Pernul and A.M. Tjoa, editors, *Proc. 11th Int. Conf. on the Entity-Relationship Approach, LNCS 645*, pages 7–23. Springer, 1992.

[23] A. Zamperoni. A Conceptual Framework how to Generate Logical Models for EER Schemata. Diploma Thesis (in German), Technical University of Braunschweig, 1992.

[24] A. Zamperoni and P. Löhr-Richter. Enhancing the Quality of Conceptual Database Specifications through Validation. In R.A. Elmasri and V. Kouramajian, editors, *12th Intern. Conf. on the Entity-Relationship Approach*, pages 87–99, 1993.