Institut für Informatik der Universität Zürich

Towards an Integrity Model Specification Language

Stefan Scherrer
Klaus R. Dittrich

Technical Report Nr. 93.13 April 93

Towards an Integrity Model Specification Language

Stefan Scherrer, Klaus R. Dittrich

Forschungsbereich Datenbanktechnologie
Institut für Informatik, Universität Zürich

Abstract

Integrity management is one of the characteristic requirements for database management systems (DBMS). While part of the miniworld constraints are expressible in terms of data model or transaction model concepts and are thus enforced by the respective subsystems, others remain to be enforced by a general-purpose integrity management subsystem (IMS) which allows the definition of constraints in terms of a logical language and which enforces these constraints with respect to database state transitions.

The construction of an IMS is generally an awkward task, both in that the similarities between different IMS are substantial (but reusability techniques are rarely applied) and in that data and transaction management are quite often designed and implemented before integrity management is considered a serious requirement.

In this paper, we propose a specification language called IMSL whose primary goal is to rationalize the IMS-construction task and which moreover enables the description of existing IMS. It turns out that production rule concepts play an important role in IMS specification, both in that they serve as a metaphor to define the semantics of logical languages (for the definition of constraint conditions) and in that they allow the descriptive specification of state transitions.

1 Introduction

Integrity management is one of the characteristic requirements for DBMS. It is concerned with the consistency of database states and state transitions with respect to a given set of *constraints*. *Operational integrity* holds when different clients properly interact according to a transaction model. *Semantic integrity* holds when each database state satisfies a given set of (declarative) constraints which are formulated using some kind of *logical language*.

A database state is the result of a *state transition* (a sequence of *atomic actions*) which *starts* from an old database state and *ends* in a new one. At the start of such a state transition, integrity is assumed to hold, while the DBMS has to enforce that integrity holds for resulting database states as well. In cases where the transition attempts to end in an inconsistent state, the DBMS has to extend the current transition by triggering *reactions* to violated constraints.

The set of *possible* database states is defined by the concepts of the *data model*. Semantic integrity constraints restrict the set of possible database states to the set of *allowed* database states[1].

They can be primarily classified according to the language concepts which are provided to express them and according to the kinds of reactions that are triggered in case of their violation.

- *Inherent integrity constraints* are directly imposed by the data model itself. They are independent of any data schema. Their enforcement is ensured by the correct implementation of the atomic actions of the database system, i.e. by the DBMS implementor (DBI). For example, first normal form and uniqueness of tuples in relations are inherent constraints of the relational data model.
- *Implicit integrity constraints* are again captured by the data model but are schema dependent; i.e., they are identified during the database design phase and are expressed in the schema by the database administrator (DBA). Databases must correspond to their schema definitions, and this is (again) ensured by the correct implementation of the atomic actions. For example, uniqueness of relational attribute values is an implicit integrity constraint of the relational data model.
- *Explicit integrity constraints* subsume the constraints which are controlled by the DBMS but which are not defined in the schema. During a state transition, these constraints can be violated by atomic actions. At the end of a state transition (which may have been requested by the DBA), however, they are checked by the DBMS and, in case of constraint violations, reactions are triggered in order to produce a consistent database state. The duration of a state transition may either be defined by the DBI or explicitly indicated by the DBA. For example, referential integrity constraints or assertions are explicit integrity constraints in some relational systems.

  The constraint language for explicit constraints, the possible set of state transitions and the general IMS behavior explained above form an *integrity model* (in analogy to data and transaction models).
- *External integrity constraints* are the real world constraints which are not captured by inherent constraints and which cannot be defined by means of implicit and explicit constraints. Since they are specified and enforced outside of the DBMS (i.e., in the application system), they are not of further interest to us.

In other words, many kinds of integrity constraints exist, and DBMS-mechanisms are required to enforce them. Typically, these mechanisms are spread over the various subsystems of a DBMS. While the transaction management subsystem (TMS) is responsible for operational integrity enforcement, the data management subsystem (DMS) is responsible for inherent and implicit integrity constraint enforcement. The integrity management subsystem (IMS) enforces explicit integrity constraints. The interaction of these subsystems has to be carefully designed by the DBI at DBMS-construction time.

Many different DBMS emerged in the past and will emerge in the future. The construction of such systems “from scratch” is very resource intensive. Extensible database management systems (eDBMS) try to cure these symptoms by incorporating some kind of software reuse technique [Krueger 92]. Different eDBMS mainly differ in the applied reuse technique and the addressed DBMS aspects (e.g. data, transaction, integrity, and storage management).

The KIDS approach to DBMS construction [Dittrich 91] is an attempt to increase the *implementation efficiency* of *run-time efficient* DBMS. The general idea (sketched in Figure 1) is to

1. The set of *possible* transitions includes arbitrary transitions which result in a possible database state, while the set of *allowed* transitions is the subset of possible transitions which result in an allowed state.

specify the relevant properties of the subsystems and to automate their construction as far as possible rather than to *implement* them "from scratch" again and again. *Subsystem specifications* are checked on completeness/correctness and stored in *libraries* by *specification tools* and transformed into subsystems by *generators* which make use of extensible libraries of *primitives*. While some parts of the DBMS are assumed to be *fixed* for all DBMS (e.g. physical object server), others remain to be *implemented* by the DBI in a traditional manner (e.g. user interfaces).

In this paper, we concentrate on the construction of IMS within the KIDS-framework. The concepts of an IMS (the integrity model) can be specified using the *integrity model specification language* called IMSL which is independent of particular data models and transaction models. The *integrity model specification tool* checks specifications in terms of IMSL on completeness (correctness, resp.) and creates a parse tree which serves as input for the *integrity manager generator*. The generation itself results in an IMS in terms a class library which is then used by the DBI to create the user interface (refer to Figure 1 for an overview).

The construction of an application generator in general [Cleaveland 88] and IMS generator in particular can be (at least for didactical purposes) subdivided into seven steps:

1 *Recognizing the domain* of the generator involves the recognition of code, design, and architecture patterns being candidates for automated production. For example, constraint simplification [Jeusfeld 92] is an idea which applies to any kind of constraints.

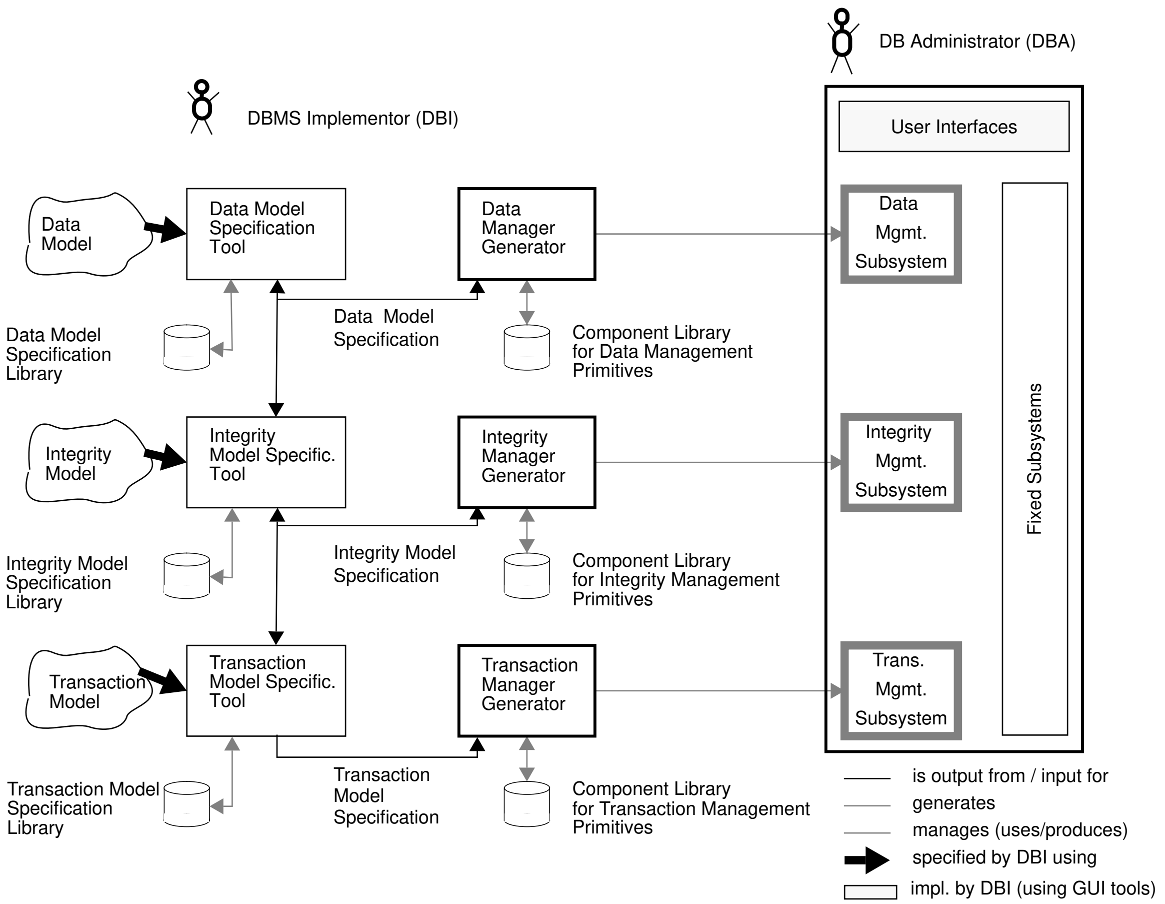

Figure 1 The overall approach ([Dittrich 91])

2 *Defining the domain boundaries* includes the definition of interfaces to the other aspects. In our context, the boundaries are defined by transaction and data management; i.e. inherent and implicit constraints are enforced by the DMS and operational integrity constraints are managed by the TMS such that the IMS enforces explicit constraints only.

3 *Defining the underlying model* helps to express the semantics of each generator feature. For example, mathematical logics and automata theory would be candidates to define the semantics of constraints and the principal behavior of IMS.

4 *Defining the variant* and *invariant parts* of the generator includes identifying the properties of IMS which can be specified by the DBI (variant properties) and the properties which are fixed (invariant properties) for all IMS.

5 *Defining the specification input* defines the form in which specification information is coded. We have chosen a textual representation.

6 *Defining the output* of a generator requires a generic software design that will meet the needs of all IMS. The definition of generator products becomes the easier the more consistent and complete the underlying model is.

7 *Implementing the generator* consists of writing a software system which translates IMS specifications into a concrete IMS. Compiler construction tools (e.g. lex and yacc) ease this task substantially.

In this paper, we will mainly concentrate on step 4, 5, and 6 by presenting the syntax and semantics of IMSL. It will turn out that concepts first developed in the area of *production rule systems* play a surprisingly integral part when specifying integrity models.

The paper is organized as follows. In section 2, related work is discussed. In section 3, the

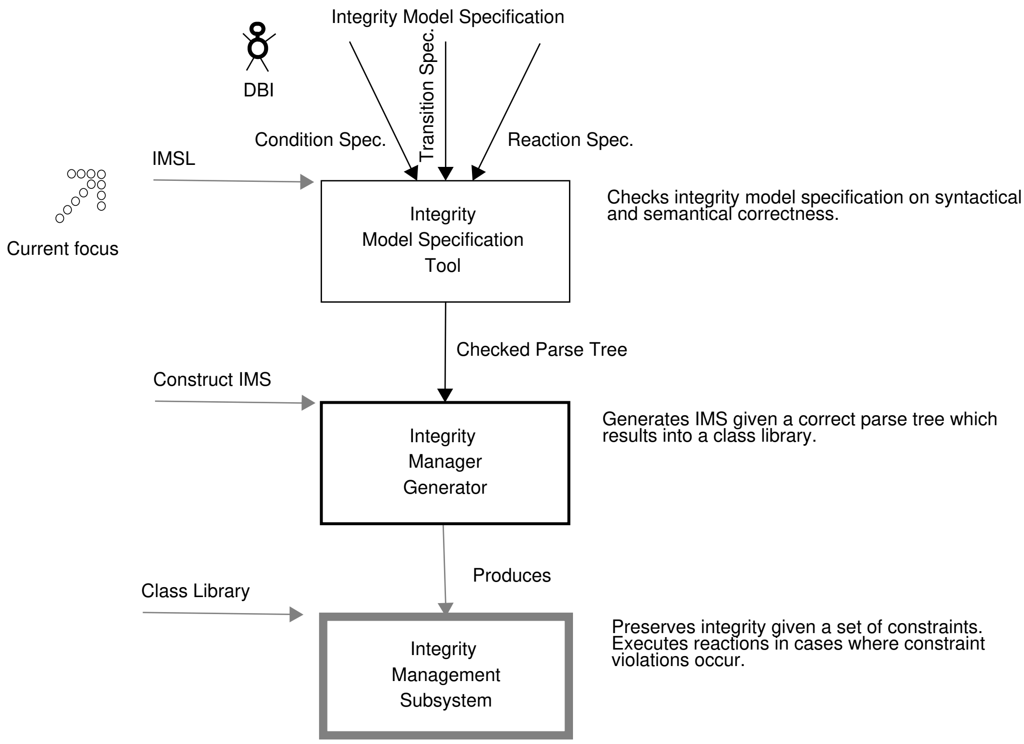

Figure 2 Overview of IMS construction

notion of integrity model is refined by introducing different subaspects of constraint management (logical language, possible reactions, state transitions). The subaspect logical language itself is then elaborated in section 4. Production rules are used to define the semantics of logical language elements. In section 5, language concepts are presented which allow the DBI to define the set of possible state transitions. Section 6 sketches the generic software architecture for explicit integrity constraint maintenance which is common to all IMS emerging as products of our approach. Remarks on future work are made in section 7.

2 Companion to Related Work

Considerable research effort has been spent to construct concrete integrity models for particular systems, for example O_2 [Medeiros 91], Starburst [Ceri 90], and ConceptBase [Jeusfeld 92]. In these approaches, an existing (data and transaction model dependent) production rule system is used to enforce integrity constraints; that is, constraints are (partially) automatically translated into sets of production rules which are then under control of a rule manager. It is often claimed that the respective integrity model can be easily adapted to other data models. However, this would require substantial re-implementation of the translator which maps declarative constraints to production rules. To the best of our knowledge, IMSL is the first specification language which attempts to factorize out this knowledge. Thus, the translation is *specified* using the production rules metaphor rather than manually *implemented*.

From an eDBMS point of view, we are not aware of any eDBMS ([Batory 88], [Carey 86], [Lohmann 91], [Maryanski 86], [Stonebraker 91], and [Thatte 91]) which explicitly supports the reuse of knowledge on IMS-construction, since they mostly concentrate on data and transaction management. Thus, the integration of the integrity management aspect into the eDBMS world by means of IMSL is another major contribution of this paper.

3 Integrity Models

An integrity model defines the set of possible explicit integrity constraints and their semantics that will be supported by the IMS of concern. Furthermore, it defines how the IMS perceives state transitions. The DBI specifies a particular integrity model in terms of the integrity model specification language IMSL. Similar to any specification languages, the expressiveness of IMSL reflects the design space of IMS which can be defined by our approach. Our design space in turn is the result of a careful domain analysis.

In the following, the various IMSL-constructs are introduced in a top-down manner. Figure 3 shows the top-level syntactical constructs for integrity model specification[1].

For each integrity model, a *logical language* has to be defined whose concepts can be used by

1. We make use of the typical EBNF-concepts to define language syntax. While all-capital words denote keywords, the others denote non-terminal symbols. The left-hand side of a production defines a non-terminal symbol. Language delimiters are quoted. Angle brackets “[ ]“define options (zero or once the contents) while curly brackets “{ }” define repetitions (zero or a finite number). Alternatives are separated by vertical slashes “|”. Round brackets “( )” are sometimes used to clarify the effect of alternatives. Non-terminals containing substring “name” usually refer to strings where not stated otherwise.

the DBA to define the *condition* parts of integrity constraints. Thus, IMSL is a meta language which allows the specification of logical languages. In IMSL, a logical language is specified in a purely functional approach by means of enumerating type symbol names [4] and function symbol names [6]. A function symbol name refers to a separate subspecification where mainly its signature and its semantics are specified. The according IMSL-constructs are treated in more detail in section 4. Assuming a given logical language (given by the set of type and function symbols), conditions can be recursively constructed by compositional application of function symbols either to existing conditions or to values. Conditions whose value is FALSE denote constraint violations.

Besides the logical language, *the set of possible reactions* in cases where a condition violation occurs is another characteristic of an integrity model. *Default* and *DBA-definable behavior* are distinguished. If a condition is violated and the respective constraint has not been attached with a DBA-defined reaction, the default reaction is applied. If a DBA-defined reaction exists, it replaces the default reaction. Thus, from a specification language point of view, while a default reaction must be specified [9], the specification of DBA-definable behavior is optional [8]. Specifying default reactions simply consists of triggering particular operations which are defined by the data model or transaction model; that is, these operations are provided with context-specific information (e.g. line [28]). On the other hand, if the DBI wants to let the DBA define own reactions, she has to enumerate the allowed operations. Reactions are not further elaborated in this paper.

The third characteristic of an integrity model is its perception of what *database states* and *state transitions* are. One invariant part of IMS specification is that state transitions have an explicit starting point (where the database state is assumed to be consistent) and an explicit termination point (where the IMS must ensure database state integrity). On the other hand, the situations which indicate the begin and the end of transitions may vary from IMS to IMS. The IMSL-means to specify such pairs of situations are illustrated in section 5.

At this place, we introduce the running example we are going to use for the demonstration of specifications in IMSL. We assume a specification RelDaMo for a "traditional" relational data model providing for the usual set-of-tuples structures and the usual operations for the selection, projection, and the cross-product of relations as well as insertion, deletion, and modification of tuples of relation instances. Moreover, operations to create new relation schemas and relation instances are assumed. Now, the DBI wants to define assertional and referential integrity con-

```
integrity-model-definition::=                                              [1]
    INTEGRITY MODEL integrity-model-name                                   [2]
            CONDITIONS                                                     [3]
                    TYPE SYMBOLS FROM data-model-name                      [4]
                            type-symbol-name { "," type-symbol-name }      [5]
                    FUNCTION SYMBOLS symbol-name { "," symbol-name }       [6]
            REACTIONS                                                      [7]
                    [ USER-DEFINED reaction-type { "," reaction-type } ]   [8]
                    DEFAULT reaction-type                                  [9]
            TRANSITIONS                                                    [10]
                    transition-type-name { "," transition-type-name }      [11]
            END                                                            [12]
```

Figure 3 The top-level language concepts

```
INTEGRITY MODEL Relinmo/* Relational integrity model */                                          [13]
    CONDITION                                                                                    [14]
        TYPE SYMBOLS FROM RelDaMo/* referencing the RelDaMo data model specification */          [15]
            Relation.RelationIid,      /* relation instances */                                  [16]
            Relation.RelationSid,      /* relation schemas */                                    [17]
            /* values */                                                                         [18]
            Integer.IntegerVid, Boolean.BooleanVid, String.StringVid, Tuple.TupleVid,            [19]
            /* domains */                                                                        [20]
            Integer.IntegerVSid, Boolean.BooleanVSid, String.StringVSid, Tuple.TupleVSid         [21]
        FUNCTION SYMBOLS    /* specified in section 4 */                                         [22]
            ∧, >, not,      /* basic predicates */                                               [23]
            ∀RelationI,     /* universal quantification on tuples in a relation instance */      [24]
            ∃RelationI,     /* existential quantification on tuple in a relation instance */     [25]
            ∀RelationS,     /* similar to ∀RelationI but for all instances of a relation schema *.[26]
            ric             /* referential constraint between two relation instances */          [27]
    REACTIONS DEFAULT FT.abort-transaction(VAL(TID))/* rollback of current transition */         [28]
    TRANSITIONS FlatTransactions  /* reference to transition specification explained in section 5 */  [29]
    END                                                                                          [30]
```

Figure 4 Application of the top-level language concepts

straints. State transitions shall correspond to flat transactions which are assumed to be specified in a transaction model specification called FT. In case of constraint violations detected at the end of the transition, the transition shall be rolled back by default. No user-defined reactions shall be allowed. Figure 4 demonstrates the application of the language concepts introduced so far to this example.

4 Logical Languages

In this section we demonstrate how a logical language can be specified using IMSL. Well-formed sentences of a logical language are called *conditions*. In contrast to existing approaches where the logical language is defined by means of concrete syntax and semantics (e.g. [Ceri 90]), we follow a more mathematical-logic approach [Andrews 86] in that we consider the design of concrete syntax as a user-interface design issue (Figure 1) and specify the essence of a logical language only which is in fact a collection of *function symbols* (e.g. lines [23] to [27]). The grammar rules for function symbol specification are summarized in Figure 6.

For each function symbol, the DBI has to enumerate the *free variables* [33] (and their respective *types*) and the *range* of the function [32]. *Sentences* of a logical language can be recursively composed in that free variables of functions are provided with *values*. Values either stem from invocations of *operations* [38] (provided by the data model specification), or from evaluations of *functions* (see below) whose ranges match the types of the free variables [42].

The types of the free variables together with the range denote the *signature* of a function symbol. Note that function signatures can serve as types of free variables as well [47]. The basic types are taken from the underlying data model specification (for an example, refer to line [16] to [21], see also Figure 5); full details of data model specification are provided in [Scherrer 92].

When defining a logical language, one typically would designate a subset of the possible sentences as well-formed sentences (conditions) by means of recursively defining formation rules. In our functional approach, conditions are simply sentences whose range is Boolean.BooleanVid, that is {TRUE, FALSE}.

Besides its signature, the DBI also has to define the *semantics* of a function symbol, that is (in our context), how the value of this function can be computed. The process of computing the value of a function is called *evaluation.* The DBI has to choose among two strategies. In the first strategy, the value is computed by invoking an operation (given by the data model) provided with actual parameters (which are indeed values again). In this strategy, the function is *completely evaluated*, i.e., in one step [34]. Depending on the invoked operation, this strategy can prove to be very expensive. Especially if one knows that the value of the function is initially *a* and can become *b* only if an operation *o* with specific parameters is invoked. Hence, the second strategy suggests an *incremental evaluation* of functions by means of triggers[1] [35].

A *trigger* defines the situations when the value of a function changes [40]. The situation in turn consists of an *event definition* and a *predicate* (yielding a value whose type must be { TRUE, FALSE }). For the sake of simplicity, we assume that for each operation provided by the data and transaction model specification, *events* are defined. They are signalled when the operation is executed. Actual event parameters describe the actual parameters and the results of operation executions. They can be used to compute the predicate. If this computation yields TRUE, the trigger fires; that is, the action part of the trigger is applied. *Actions* are in fact assignments which can either redefine the current value of the condition or redefine the current condition itself [44].

Due to the compositional nature of concrete condition definitions, conditions are not only *incrementally* evaluated when triggers fire, but also *completely* evaluated whenever the values of free variables change.

In conclusion, note the distinction between the operations given by a data model specification and the conditions being part of an integrity constraint. While the former are executed immediately (i.e., have a “snapshot” view of the database), the latter are (possibly partially) evaluated at the end of state transitions. Section 6 elaborates on details of the enforcement scheme.

In the following subsections, we give example specifications of function symbols.

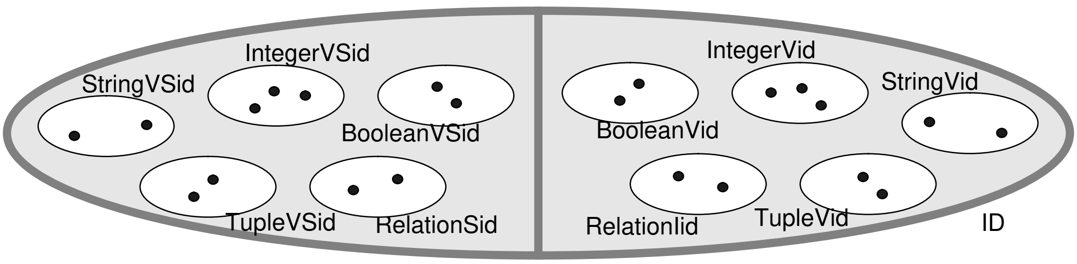

Figure 5 Database state as viewed as subsets of intensional and extensional items. Type names are subset names.

1. This closely corresponds to constraint simplification in [Jeusfeld 92].

```
fun-symbol-specification::= FUNCTION SYMBOL fun-symbol-name                                      [31]
    RANGE type-symbol-name                                                                       [32]
    FREE VARIABLES { var-symbol-declaration }                                                    [33]
    [ COMPLETE EVALUATION action ]                                                               [34]
    [ INCREMENTAL EVALUATION INITIAL action { trigger-definition } ]                             [35]
    END                                                                                          [36]
var-symbol-declaration::=var-symbol-name “:” type-symbol-name                                    [37]
op-call::= op“(“ [ value { “,” value } ] “)”                                                     [38]
op::= spec-item-name | value                                                                     [39]
trigger-definition::=EVENT event-definition WHERE value ACTION action                            [40]
event-definition::= spec-item-name “(“ var-symbol-declaration { “;” var-symbol-declaration } ”)” [41]
value::= op-call | VAL ”(“ ( condition | var-symbol-name )“)”                                    [42]
action::= { assignment } | ( COMPLETE EVALUATION )                                               [43]
assignment:= ( VAL ”(“ condition “)” “:=” value) | ( condition “:=” condition )                  [44]
condition::= fun-symbol-name [ “(“ [ substitution { “,” substitution } ] “)”] | THIS | var-symbol-name [45]
substitution::= value | condition                                                                [46]
type-symbol-name::= [ “(“ [ type-symbol-name { “,” type-symbol-name } ] ”)” “:” ] spec-item-name  [47]
spec-item-name::= spec-name “.” item-name                                                        [48]
var-symbol-name::= name | TID                                                                    [49]
fun-symbol-name::= name                                                                          [50]
```

Figure 6 Syntax necessary for function specification

4. 1 Examples: Basic Functions

In Figure 7, the logical symbol $\wedge$ is specified whose purpose is to connect two conditions with the usual boolean *and* semantics. Complete evaluation is used; i.e., the value of the function is obtained by means of passing the values of the free variables to the and operation which is defined in the Boolean specification (which is a part of the data model specification). The variable symbol THIS is predefined and denotes the function which is currently under consideration. On the left-hand side of an assignment, VAL(var-symbol) resets the evaluation of var-symbol-name. VAL(var-symbol-name) on the right-hand side returns the current value of var-symbol-name.

Of course, all necessary simple function symbols can be specified in this way. In our running example, >, not (see [23]) would have to be defined analogously.

4. 2 Examples: Assertional and Referential Integrity Conditions

For our Relinmo example, we assume assertions to be universal (existential) quantors ranging over relation instances. Such conditions evaluate to TRUE if for all tuples (for at least one tuple)

```
FUNCTION SYMBOL ∧                  /* is referenced in [23] */     [51]
RANGE Boolean.BooleanVid                                           [52]
FREE VARIABLES                                                     [53]
    first: Boolean.BooleanVid                                      [54]
    second: Boolean.BooleanVid                                     [55]
COMPLETE EVALUATION                                                [56]
    VAL(THIS):= Boolean.and(VAL(first), VAL(second))               [57]
END                                                                [58]
```

Figure 7 A logical connector

```
FUNCTION SYMBOL ∀RelationI                     /* referenced in [24] */                              [59]
RANGE Boolean.BooleanVid                                                                              [60]
FREE VARIABLES                                                                                        [61]
    rid: Relation.RelationIid                  /* id of a relation instance */                        [62]
    p: (Tuple.TupleVid): Boolean.BooleanVid    /* mapping from tuples to boolean */                   [63]
COMPLETE EVALUATION                                                                                   [64]
    VAL(THIS):= Integer.zero-p(Integer.diff(Relation.card(VAL(rid)),                                  [65]
                                 Relation.card(Relation.select(VAL(rid), VAL(p)))))                   [66]
INCREMENTAL EVALUATION                                                                                [67]
    INITIAL COMPLETE EVALUATION                /* determine initial value by a complete evaluation*/  [68]
    EVENT   Relation.insert(ri: Relation.RelationIid; tv: Tuple.TupleVid)                             [69]
    WHERE EAV.identicID(VAL(rid), VAL(ri))                                                            [70]
    ACTION VAL(THIS):= VAL(p)(VAL(tv))                                                                [71]
END                                                                                                   [72]
```

Figure 8 Assertional condition

of a set, a specific predicate evaluates to TRUE. Otherwise, these conditions evaluate to FALSE. In our rather radically functional view, quantors can be modeled by function symbols[1] as well as basic predicates. Figure 8 is a specification of the function symbol $\forall_{RelationI}$, whose (truth) value denotes whether a unary predicate p holds for all tuples of a particular relation instance rid. Both, rid and p are free variables. While the former accepts as value identifiers of relation instances, the latter accepts operation names representing mappings from Tuple.TupleVid to Boolean.BooleanVid.

Complete evaluation ([64]-[66]) is realized using data model specific operations like Integer.zero-p (testing its argument on zero), Integer.diff (yields difference of first and second parameter), Relation.card (yields the number of tuples of the argument relation), and Relation.select (yielding the subrelation of the argument relation whose tuples satisfy the predicate given as the second parameter). Given that a $\forall_{RelationI}$-condition evaluates to TRUE, only a single situation may lead to its falsification. Whenever a tuple VAL(tv) is inserted into the relation instance VAL(rid)[2], the new evaluation is given by the application of VAL(p) to VAL(tv). Note that the deletion of tuples does not affect the evaluation.

An existential quantifier $\exists_{RelationI}$ with signature (Relation.RelationIid, (Tuple.TupleVid): Boolean.BooleanVid): Boolean.BooleanVid could be defined analogously. In the following, however, we demonstrate how to specify the function symbol ric whose application leads to referential integrity conditions.

Here, complete evaluation is mainly based on the data model operation Relation.sub-relation-p which returns TRUE if the relation denoted by the first parameter contains at least the tuples of the relation denoted by the second actual parameter (and FALSE otherwise). Two situations may falsify a concrete ric condition with respect to incremental evaluation. It may evaluate to FALSE if a tuple with a specific attribute value of attribute with name VAL(foreign-name) is inserted into the relation instance VAL(foreign-rid) that has no corresponding tuple attribute value in the relation instance VAL(primary-rid). On the other hand, falsification may occur when a primary key value is deleted which has been referenced by at least one foreign key value. In both cases, this is

1. Specification of quantors that quantify over sequences of database states will be investigated in the future.
2. The (predefined) data model operation EAV.identicID allows to check whether two identifiers are equal.

```
FUNCTION SYMBOL ric                          /* referenced in [27] */
RANGE Boolean.BooleanVid
FREE VARIABLES
    primary-rid: Relation.RelationIid        /* id of a relation instance containing primary key */
    foreign-rid: Relation.RelationIid        /* id of a relation instance containing foreign key */
    primary-name: String.StringVid           /* attribute name of primary key */
    foreign-name: String.StringVid           /* attribute name of foreign key */
COMPLETE EVALUATION
    VAL(THIS):= Relation.sub-relation-p(Relation.project(VAL(primary-rid), VAL(primary-name)),
                                  Relation.project(VAL(foreign-rid), VAL(foreign-name)))
INCREMENTAL EVALUATION
    INITIAL COMPLETE VALUATION
    EVENT Relation.insert(f: Relation.RelationIid; t: Tuple.TupleVid)
    WHERE EAV.identicID(VAL(f), VAL(foreign-id))
    ACTION VAL(THIS):= Relation.subrelation-p(
            Relation.createRelationIofTupleV(Tuple.project(VAL(t), VAL(foreign-name))),
            Relation.project(VAL(primary-rid), VAL(primary-name)))
    EVENT Relation.delete(p: Relation.RelationIid; t: Tuple.TupleVid)
    WHERE EAV.identicID(VAL(p), VAL(primary-id))
    ACTION VAL(THIS):= Boolean.not(Relation.subrelation-p(
            Relation.createRelationIofTupleV(Tuple.project(VAL(t), VAL(primary-name))),
            Relation.project(VAL(foreign-rid), VAL(foreign-name))))
END
```

Figure 9 Referential condition

again expressed using the data model operation Relation.sub-relation-p. The operation Relation.createRelationIofTupleV is used to create a singleton relation given a tuple.

4. 3 Examples: Conditions on Intensional Level

The function symbols introduced so far enable the DBA to define conditions on objects, e.g., relation instances. However, from a DBA point of view, conditions with a "for all relation instances corresponding to a particular relation schema, condition A has to hold" are highly desirable as well. Hence, similar condition types have to be provided by the DBI. The following example shows that this is possible in IMSL.

Since we assume a data model specification which does not provide an operation which delivers the existing relation instances given a relation schema identifier, no complete evaluation can be specified. As another consequence of this fact, the initial evaluation value with respect to incremental valuation is set to TRUE. Thus, a $\forall_{RelationS}$-condition monitors newly defined relation instances corresponding the specific relation schema only. Whenever such an instance is created (VAL(rid)), the current condition (THIS) is replace by a disjunction ($\wedge$) of the current condition and an condition ($\forall_{RelationI}$) with relation instance (VAL(rid)) and an operation (VAL(p)) as substitutions for its free variables. Of course, if a deletion of a relation instance with schema VAL(rid) occurred, THIS would become corrupted. Currently, IMSL provides no means to automatically modify conditions (e.g., dropping a conjunction given a series of conjunctions). This is still matter of further investigations.

```
FUNCTION SYMBOL∀RelationS                  /* referenced in [26] */                [96]
RANGE Boolean.BooleanVid                                                           [97]
FREE VARIABLES                                                                     [98]
    rsid: Relation.RelationSid             /* id of a relation instance */         [99]
    p: (Tuple.TupleVid): Boolean.BooleanVid   /* mapping from tuples to boolean */ [100]
INCREMENTAL EVALUATION                                                             [101]
    INITIAL VAL(THIS):= Boolean.TRUE()     /* assigning the constant TRUE */       [102]
    EVENT  Relation.createRelationI(rs: Relation.RelationSid; rid: Relation.RelationIid)  [103]
    WHERE EAV.identicID(VAL(rsid), VAL(rs))                                        [104]
    ACTION THIS:= ∧(THIS, ∀RelationI(VAL(rid), VAL(p)))                            [105]
END                                                                                [106]
```

Figure 10 A relation schema quantor

4. 4 Examples: The DBA View of Conditions

In this subsection, we develop a scenario which describes how the DBA might perceive a specific IMS with respect to conditions. Assume that Relinmo has previously been specified (Figure 4). Moreover, assume that both, the specification has been checked and an appropriate IMS has been generated (Figure 1).

In this scenario, the DBA is provided by a user interface which is implemented by the DBI using interface building tools (Figure 1) in the form of a language that supports the creation of:

- *Variables*: Variables are typed with a type symbol introduced in Relinmo. ":=" denotes assignment of a value to a variable. Values shall be either evaluations of conditions or results of relational data model operation call (refer to Figure 11 for examples).
- *New function symbols*: The DBA is provided with a sublanguage of IMSL which enables the definition of new function symbols. Of course, the DBA would have to provide for the symbol name, the range, the free variables, and a complete evaluation specification (refer to Figure 12 for an example).
- *Concrete conditions*: Condition names are defined similar to variable names. The constructor names correspond to the names of the function symbol specifications referenced by Relinmo and variables (refer to Figure 13 for two examples).

Of course, the DBA-defined variable and function symbol definition facilities will be offered by means of appropriate library functions, which finally represent the programmers interface to the integrity management subsystem. The same holds for the construction, destruction and modification of conditions.

5 State Transitions

As already mentioned in the introductory section, the transaction model cannot be assumed to be fixed in the KIDS-context. From an integrity enforcement point of view, the transitions from one consistent database state to another are of particular interest, hence we need a notion to define what state transitions are in the context of our integrity model of consideration. While some integrity models assume that such a transition starts with something like a begin-transaction and ends with a commit-transaction, others may start with begin-transaction again but end with some-

```
VARIABLE prs, frs RANGE Relation.RelationSid/* primary and foreign relation schema identifier */
VARIABLE pri, fri  RANGE Relation.RelationIid /* primary and foreign relation instance identifiers */

/* the schema of prs shall (simplified) be: Emp(Emp#, EmpName, Sal, Dept#,....) */
/* the schema of frs shall (simplified) be: Dept(Dept#, DeptName,...)

pri:= Relation.createRelationIofSchema(prs) /* create a relation instance according to schema prs */
fri:= Relation.createRelationIofSchema(frs) /* create a relation instance according to schema frs */
```

Figure 11 Sample schema definition

```
FUNCTION SYMBOL salary-ok-p
RANGE Boolean.BooleanVid          /* value of salary attribute respects the salary limit*/
FREE VARIABLES
    person: Tuple.TupleVid        /* the tuple whose salary attribute is to be checked */
COMPLETE EVALUATION
    VAL(THIS):= Integer.greater-p(10000, Tuple.project(VAL(person), "Sal"))
END
```

Figure 12 Sample function definition

```
/*  create a concrete referential integrity condition empDept checking referential integrity
    among Emp.Dept# and Dept.Dept# */
CONDITION empDept ric(VAL(pri), VAL(fri), "Dept#", "Dept#")

/*  create a concrete assertional condition empSalary on attribute Emp.Sal */
CONDITION empSalaryCheck ∀RelationS(VAL(pri), salary-ok-p)
```

Figure 13 Sample condition definition

thing like check-transaction which explicitly forces the system to check the integrity of the transition carried out so far, and to make the further execution of the transaction dependent on the checking result. Hence, it is up to the integrity model definer (the DBI) to specify the intervals where integrity is possibly violated.

The following language construct enables this kind of specification.

```
transition-type::= TRANSITION TYPE transition-type-name
                START event-definition
                END event-definition
                INTERMEDIATE event-definition
                WHERE value
                TID value
    END
```

Figure 14 Syntax necessary for transition type specification

Figure 15 is an example of a transition type specification. It assumes an existing transaction model specification FT which defines the events begin-transaction and end-transaction. It is up to the IMSL-specification tool to verify that these events exist[1] and match parameter lists. TID represents a predefined variable of IMSL in order to enable references from reaction specifications to the transition whose correctness is currently checked.

```
TRANSITION TYPE FlatTransaction                                   [133]
    START FT.begin-transaction(t1: TMS.Tid)                        [134]
    END FT.end-transaction(t2: TMS.Tid)                            [135]
    INTERMEDIATE FT.check-transaction(t2: TMS.Tid)                 [136]
    WHERE FT.equal(VAL(t1), VAL(t2))                               [137]
    TID VAL(t1)                                                    [138]
END                                                                [139]
```

Figure 15 A transition type example

6 Enforcement

Integrity enforcement basically consists of *checking conditions* before the end of state transitions and (in cases where constraint violations occur) of *executing reactions*.

As usual, condition checking is driven by two major requirements. First, the number of conditions to be evaluated have to be minimized. Only those conditions whose "interpretation" changes during the state transition should be checked. Consequently, at the end of a state transition, the enforcement mechanism must know what actually happened in between the interval borders. This information is typically kept in some kind of *log*[1]. The log records are assumed to contain the event occurrences (including their actual parameters) raised by the data and transaction management subsystem. Triggers will be generated for each kind of condition. They fire when corresponding events (read from the log at the end of a state transition) match their event and where clause. Hence, each condition attached to such a trigger has to be evaluated. Second, the evaluation of the conditions actually under consideration at the end of a transaction has to be as efficient as possible. The incremental evaluation scheme addresses this requirement. Whenever a trigger fires, the value of the THIS can be determined in that a simpler condition is evaluated (according to the ACTION-clause of the trigger). If THIS has been assigned to a free variable of another condition c, c is completely re-evaluated. Thus, the firing of a specific trigger may transitively cause the propagation of complete condition evaluation according to the concrete composition of conditions.

During this checking process, the integrity constraints whose conditions have evaluated to FALSE are collected. On completion of the checking process, the collection is examined; if it is empty, the actions executed during the state transition have preserved database integrity, otherwise, database integrity is violated and the execution of reactions is required in order to return to a consistent database state. Reactions are executed until the collection of violated integrity constraints turns out to be empty. Of course, certain combinations of conditions and reactions may provoke non-termination of integrity enforcement. Two strategies to circumvent this situation exist. First, the DBA could define a time-out interval; if it expires, the default reaction is applied (typically backward recovery of the transition). Second, we could assume that the

1. With respect to DMS and TMS specification, we assume here that for each provided operation an event exists whose formal parameters correspond to the formal parameters and the result of the operation.
1. Currently it is not yet clear whether the log of the transaction management can be used for this purpose or whether the integrity management has to keep a log of its own. For the following considerations, this issue is not important.

DBA-specified reactions always bring the database back into a consistent state. Consequently, re-checking of the constraint after the execution of its action would not be necessary, and the constraint enforcement would terminate in any case.

7 Future Work

From an IMSL-*implementation* point of view, the definition of the output format of the constructor will be the next major step to be taken. This includes the selection of a programming language as target language as well as an appropriate IMS architecture (refer again to Figure 1).

From an IMSL-*language design* point of view, we admit that static conditions in this paper might have been over-emphasized; i.e., constraints to be valid in any database state. Dynamic constraints, in contrast, possibly refer to several database states [Lipeck 87]. We plan to extend our framework of static condition type specification in order to capture dynamic conditions as well in the near future. Dynamic conditions typically involve temporal quantifiers and temporal connectors which in turn are special function symbols. Roughly speaking, using a minimal set of quantifiers and connectors, dynamic conditions can be represented as *normalized transition graphs* [Lipeck 87] whose nodes represent temporal condition states and whose edges are labeled with non-temporal conditions to be checked during the next state transition. Whatever checking strategy is used (monitor, transaction modification, or generation of triggers [Gertz 92]), transition graphs represent a uniform means to capture the semantics of dynamic conditions. Different application domains may require different temporal quantors and connectors. The DBI will define their semantics in terms of the initial set of quantors and connectors.

Future language design work will also include refinements in order to increase/clarify the reusability of integrity model specifications themselves.

Acknowledgments

The authors wish to thank P. Baumann, A. Kotz-Dittrich, S. Gatziu, and A. Geppert for invaluable discussions about mathematical foundations, production rule systems, event languages, and eDBMS in general and their careful reading of earlier drafts of this paper.

Bibliography

[Andrews 86] Andrews P. B.; An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof; Academic Press Inc., Orlando FL, USA, 86;

[Batory 88] Batory D. S., Barnett J. R., Garza J. F., Smith K. P., Tsukuda K., Twichell B. C., Wise T. E.; Genesis: An Extensible Database Management System; IEEE TOSE 14 (11), November 88; pp. 1711-1730; and in: Zdonik S. B., Maier D.; Readings in Object-Oriented Database Systems; Morgan Kaufman Publishers Inc., San Mateo CA, USA, 90; pp. 500-518;

[Carey 86] Carey M. J., DeWitt D. J., Frank D., Graefe G., Muralikrishnan M., Richardson J. E., Shekita E. J.; The Architecture of the EXODUS Extensible

DBMS; In: Proc. 1st Intl. IEEE Workshop on Object-Oriented Database Systems; IEEE, Pacific Grove, Asilomar, CA, USA, 86; pp. 52-65;

[Ceri 90] Ceri S., Widom J.; Deriving Production Rules for Constraint Maintenance; In: Proc. 16th Intl. Conf. on Very Large Databases, Brisbane, AU, August 90; pp. 577-589;

[Cleaveland 88] Cleaveland J. C.; Building Application Generators; IEEE Software, 5 (4), July 88; pp. 25-33;

[Dittrich 91] Dittrich K. R., Geppert A., Goebel V., Nittel S., Scherrer S.; KIDS: A Declarative Approach to DBMS Construction; In: Proc. Conf. on Database Research in Switzerland, Lausanne, October 91; pp. 176-186;

[Gertz 92] Gertz M.; Deriving Integrity Constraint Maintaining Triggers from Transition Graphs; Proc. 4th GI Workshop “Grundlagen von Datenbanken”, Barsinghausen, FRG, June 92; pp. 40-44;

[Jeusfeld 92] Jeusfeld M., Jarke M.; From Relational to Object-Oriented Integrity Simplification; Proc. 2nd Intl. Conf. on Deductive and Object-Oriented Databases, Munich, FRG, December 91; pp. 460-477;

[Kotz 88] Kotz A., Dittrich K. R., Mülle J.; Supporting Semantic Rules by a Generalized Event/Trigger Mechanism; In: Proc. Intl. Conf. on Extending Database Technology, Venice, 88; LNCS 303, Springer, Berlin, FRG, 88; pp. 22-36;

[Krueger 92] Krueger C. W.; Software Reuse; ACM Computing Surveys, 24 (2), June 92; pp. 131-185;

[Lipeck 87] Lipeck U., Saake G.; Monitoring Dynamic Integrity Constraints Based on Temporal Logic; Information Systems, 12 (3), 87; pp. 255-269;

[Lohmann 91] Lohman G. M., Lindsay B., Pirahesh H., Schiefer K. B.; Extensions to Starburst: Objects, Types, Functions and Rules; CACM 34 (10), October 91; pp. 95-109;

[Maryanski 86] Maryanski F., Bedell J., Hoelscher S., Hong S., McDonald L., Peckham J., Stock D.; The Data Model Compiler: A Tool for Generating Object-Oriented Database Systems; In: Proc. 1st Intl. IEEE Workshop on Object-Oriented Database Systems; IEEE, Pacific Grove, Asilomar, CA, USA, 86; pp. 73-83;

[Medeiros 91] Medeiros C., Pfeffer P.; Object Integrity Using Rules; In: Proc. European Conf. on Object-Oriented Programming, Geneva 91; LNCS 512, Springer, Berlin, FRG, 91; pp. 219-230;

[Scherrer 92] Scherrer S.; The KIDS Data Model Specification Language; Technical Report, Forschungsbereich Datenbanktechnologie, Institut für Informatik, Universität Zürich; December 92;

[Stonebraker 90] Stonebraker M.; On Rules, Procedures, Caching and Views in Data Base Systems; In: Proc. ACM SIGMOD 1990 Intl. Conf. on Management of Data, Atlantic City NJ, USA, May 90; ACM, New York NY, USA, 90; pp. 281-290;

[Stonebraker 91] Stonebraker M., Kemnitz G.; The POSTGRES Next Generation Database Management System; CACM, 34 (10), October 91; pp. 64-78;

[Thatte 91] Thatte S. M.; A Modular and Open Object-Oriented Database System; SIGMOD RECORD, 20 (1), March 91; pp. 47-53;